**Bugs/Banned STL uses/Missing implementations:**

There are no major bugs in the code. No banned stl’s were used. No implementations are missing.

**Testing:**

**RadixTree.h:**

The RadixTree is tested in a separate file with an almost identical implementation of the RadixTree – the only difference being that the RadixTree is not a template class and only takes int’s as values. The radix tree is tested with the following code:

RadixTree rt;

rt.insert("Cards", 5);

rt.insert("Coin", 7);

rt.insert("Cats", 6);

rt.insert("Bash", 1);

rt.insert("Cause", 3);

rt.insert("Coit", 6);

rt.insert("Car", 2);

rt.insert("Cash", 6);

rt.insert("Dog", 3);

rt.insert("D", 5);

rt.insert("Dogs", 9);

rt.insert("Doggy", 8);

rt.insert("Donald", 1);

rt.insert("Donnel", 5);

rt.insert("Basher", 8);

rt.insert("Catty", 9);

rt.insert("Com", 1);

rt.insert("Bucky", 8);

rt.insert("Ca", 3);

rt.insert("Casher", 5);

rt.insert("Coi", 2);

rt.insert("Casher", 7);

rt.insert("Coi", 5);

rt.insert("Bash", 3);

rt.insert("Car", 9);

preOrderTraversal(rt.getHead());

cout << endl;

assert(\*(rt.search("Cards")) == 5);

assert(\*(rt.search("Coin")) == 7);

assert(\*(rt.search("D")) == 5);

assert(\*(rt.search("Cause")) == 3);

assert(\*(rt.search("Donald")) == 1);

assert(\*(rt.search("Donnel")) == 5);

assert(\*(rt.search("Catty")) == 9);

assert(\*(rt.search("Bucky")) == 8);

assert(rt.search("Cashers") == nullptr);

assert(rt.search("Cat") == nullptr);

assert(rt.search("Don") == nullptr);

assert(rt.search("E") == nullptr);

The preOrderTraversal function should have the following output:

B

ash 3

er 8

ucky 8

C

a 3

r 9

ds 5

sh 6

er 7

t

s 6

ty 9

use 3

o

i 5

n 7

t 6

m 1

D 5

o

g 3

gy 8

s 9

n

ald 1

nel 5

The radix tree should not have any memory leaks.

**PersonProfile:**

PersonProfile is tested with the following code:

PersonProfile p("Eric", "eliu0120@gmail.com");

AttValPair attVal1("Hobby", "Gameing");

AttValPair attVal2("Hobby", "Hiking"); // Test same attribute different value

p.AddAttValPair(attVal1);

p.AddAttValPair(attVal2);

AttValPair attVal3("Occupation", "student");

AttValPair attVal4("Gender", "male");

p.AddAttValPair(attVal3); // Tests identical AVP pair #1

p.AddAttValPair(attVal4); // Tests identical AVP pair #2

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

AttValPair av;

p.GetAttVal(i, av);

cout << av.attribute << "->" << av.value << endl;

}

The correct output is:

Hobby->Gameing

Hobby->Hiking

Occupation->student

Gender->male

**AttributeTranslator:**

AttributeTranslator is tested with the following code:

AttributeTranslator at;

at.Load("translator.txt");

string srcAtt = "";

string srcVal = "";

cout << "Input source attribute: ";

getline(cin, srcAtt);

cout << "Input source value: ";

getline(cin, srcVal);

cout << endl;

AttValPair src(srcAtt, srcVal);

vector<AttValPair> compPairs = at.FindCompatibleAttValPairs(src);

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

cout << structToString(compPairs[i]) << endl;

The translator.txt file has extra newlines to test that AttributeTranslator works despite empty lines existing in the file. The code takes in an input attribute and an input value, and outputs compatible attribute value pairs. For example, inputting “trait” and “sly” should give the following outputs:

job,engineer

trait,disputatious

job,salesperson

**MemberDatabase:**

Loading the member database and FindMatchingMembers of the MemberDatabase is tested with the following code:

MemberDatabase md;

md.LoadDatabase("testMembers.txt");

string att = "";

string val = "";

cout << "Input attribute: ";

getline(cin, att);

cout << "Input value: ";

getline(cin, val);

AttValPair pair(att, val);

vector<string> emails = md.FindMatchingMembers(pair);

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

cout << emails[i] << endl;

testMembers.txt is a text file containing only a portion of the database in members.txt. The code takes in an input attribute and an input value, and outputs emails with the matching attribute value pair. For example, inputting “trait” and “sly” should give the following outputs (not necessarily in the same order):

ReAguir47586@icloud.com

AbFow2483@charter.net

AnHolma15222@me.com

GetMemberByEmail is tested with the following code:

string email = "";

cout << "Input email: ";

getline(cin, email);

const PersonProfile\* pp = md.GetMemberByEmail(email);

cout << pp->GetEmail() << " " << pp->GetName() << endl;

The code takes in an input email and outputs the email and name associated with the email. For example, inputting “AbFow2483@charter.net” gives the following output:

AbFow2483@charter.net Abdullah Fowler

**MatchMaker:**

MatchMaker is tested with the provided testing framework itself, which asks for an input email, outputs the email’s associated attribute value pairs, asks for an input thresholds, and outputs compatible emails. For example, inputting “AbFow2483@charter.net” should give the following output:

The member has the following attributes:

job --> architect

hobby --> canyoneering

hobby --> slacklining

hobby --> painting

trait --> sly

trait --> incurious

trait --> excitable

trait --> unself-critical

trait --> vacuous

trait --> discouraging

trait --> disloyal

trait --> aloof

Inputting 7 should give the following output:

The following members were good matches:

Hakeem Gould at HakeemGould24@juno.com with 7 matches!

Kyle Bradshaw at KyleBra@earthlink.net with 7 matches!

Nathan Ortega at NathO6056@ymail.com with 7 matches!

Both output should match the output of the sample executable file given. The compatible emails outputted should be listed in decreasing number of matches, followed by alphabetical order of the emails.