CS32 Winter 2022
Project #4: Unhinged

Due: 11 PM, Thursday, March 10

Make sure to read the entire document (especially Requirements and Other Thoughts) before starting your project.

Introduction

Before writing a single line of code or reading the rest of this document, you MUST first read AND THEN RE-READ the Requirements and Other Thoughts section. Print out this page, tape it to your wall or on the mattress above your bunk bed, etc. And read it over and over.

The NachenSmall Software Corporation, which has traditionally only built software for running senior-citizen bingo games, has decided to pivot into a new area. They've decided to disrupt the online matchmaking market and build a new dating app called… Unhinged.

The evil co-CEOs, Carey and David (not to be confused with Harry and David, the purveyors of yummy gourmet gift baskets) have become increasingly frustrated with today's superficial, picture-filled dating apps and have decided to create a text-only dating app to help people focus on what's really most important—compatibility. They want their app to have these basic features:

- It must be able to support up to 100k members
- Each member can have dozens of *attribute-value pairs*. For example, for the pairs "hobby" → "nose hair braiding" and "occupation" → "pet stylist", "hobby" would be an attribute, and "nose hair braiding" would be its corresponding value, etc.
- To identify compatible members, the Unhinged app needs to have a way to translate a member's attribute-value pairs to a set of compatible attribute-value pairs. For example, if a member searching for dates has an attribute-value pair of "hobby" → "nose hair

braiding”, then this might be translated to a compatible attribute-value pair of “physical attribute” → “long nose hair”, since they’d likely want someone who has long nose hair as a partner.
- The ability to rank order all potential matches for a member by identifying the people with the most compatible attribute-value pairs to that member.

Given their eccentric ways, Carey and David initially asked a student at UC Berkeley to build their new dating app. The student was able to complete a set of interface definitions for the project, but then resigned to go on a six-month Bikram Yoga and farming retreat in the Santa Cruz mountains to grow hydroponic hemp (you know those UCB undergrads).

Since the student was unable to complete the actual implementation of these interfaces, Carey and David will be providing you with the design so far for you to build off of.

Your job in this year’s Project #4 is to build the five classes required to complete the dating app:
- **PersonProfile**: A class that represents a person and all of their attribute-value pairs
- **MemberDatabase**: A class that stores all the members’ profiles (name, email, attributes for each member)
- **AttributeTranslator**: A class capable of translating an input attribute (e.g., “hobby” → “eating”) into a set of compatible attributes (e.g., “job” → “chef”, or “hobby” → “cooking”) for potential matches
- **MatchMaker**: A class that identifies and rank-orders matches to other member profiles for a given member
- **RadixTree**: A class that implements a templated map using the radix tree[1] data structure

Here’s what you might see when you run the program:

```
Enter the member's email for whom you want to find matches:
sm0lbirg@hotmail.com
The member has the following attributes:
hobby --> coding
job --> professor
gender --> male
hobby --> baking
favorite food --> b'stilla
How many shared attributes must matches have? 5
```

[1] https://bruinlearn.ucla.edu/courses/109755/pages/prefix-trees-trie-and-radix-tree?module_item_id=4793722 (27-min video)
https://en.wikipedia.org/wiki/Radix_tree

```
The following members were good matches:
Jamie Lai at jamie_lai24143@hotmail.com with 13 matches!
Stephen Li at slee1724@gmail.com with 11 matches!
Emile Gin at egin9389@xfinity.com with 10 matches!
Jame Buoy at jamebuoy2008@gmail.com with 6 matches!
Karrie Wong at kw4224@aol.com with 6 matches!
```

What Do You Need to Do?

Question: So, at a high level, what do you need to build to complete Project #4?

Answer: You'll be building five complete classes, detailed below:

Class #1: You need to build a class called **PersonProfile** that holds a person's profile (i.e., their name, email, and attributes).

Class #2: You need to build a class called **MemberDatabase** that can store at least 100k person profiles and lets you obtain a profile based on a member's email address, as well as search for the set of members that have a particular attribute-value pair in their profile.

Class #3: You need to build a class called **AttributeTranslator** that can translate from an input attribute to a set of compatible attributes.

Class #4: You need to build a class called **MatchMaker** that can find all relevant matches for a given member based on their attribute-value pairs, the attribute-value pairs of the other members, and a threshold indicating the minimum set of matching attributes.

Class #5: You need to build a class template called **RadixTree** that implements a radix tree-based map, capable of mapping std::strings to any data type.

You will find all of the gory details below in the section called Details.

What We Will Provide

We will provide the following for your use:

- A ***main.cpp*** file that has a main() function that lets you run/test out your match making classes. You may modify this file for testing purposes, **BUT YOU WILL NOT TURN IT IN**

WITH YOUR SOLUTION, so any changes must not be required for proper functioning of your solution**.**

- A ***provided.h*** file which contains a couple of struct definitions that are used in the interfaces of some of the classes you will write. **YOU MUST NOT MODIFY THIS FILE AS YOU WILL NOT TURN IT IN WITH YOUR SOLUTION.**
- A member database file, called ***members.txt*** which contains a list of members and their attribute-value pairs
- An attribute translator data file, called ***translator.txt***, which contains a list of attributes and their translations

You will write .h and .cpp files for each of the classes **PersonProfile**, **MemberDatabase**, **AttributeTranslator**, and **MatchMaker**, and a .h file for **RadixTree**. If you declare and implement your classes correctly, they should work perfectly with our main() driver and you'll create your own awesome dating app!

Details

PersonProfile Class

You must build a class called PersonProfile which implements a member's profile. A profile for a given member includes the following items:

- A person's name (e.g., David Sm0lbirg)
- A person's email address (e.g., sm0lbirg@hotmail.com)
- One or more attribute-value pairs that describe the member, e.g., ("hobby", "coding")

Your PersonProfile class:

- **MUST** be able to add and retrieve attribute-value pairs in better than O(N) time where N is the number of attribute-value pairs stored in the object. So for example, $O(\log_2 N)$ would be acceptable. For big-O analysis purposes, you may assume that there's a constant that all attribute lengths are less than.
- **MUST** use your RadixTree class to map attributes to values (for full credit)
- **MUST NOT** use the STL map, unordered_map, multimap, or unordered_multimap types
- **MUST NOT** add any new public member functions or variables
- MAY use the STL list, vector, set, and unordered_set classes
- MAY have any private member functions or variables you choose to add

Your PersonProfile class must have the following methods:

PersonProfile(std::string name, std::string email)

This constructs a PersonProfile object, specifying the member’s name and email address.

~PersonProfile()

You may define a destructor for PersonProfile if you need one.

std::string GetName() const

The GetName method returns the member’s name.

std::string GetEmail() const

The GetEmail method returns the member’s email address.

void AddAttValPair(const AttValPair& attval)

The AddAttValPair method is used to add a new attribute-value pair to the member’s profile. If the person’s profile already has an attribute-value pair with the same attribute and value as the attval parameter, then this method should do nothing. More than one attribute-value pair in the map can have the same attribute, as long as their corresponding values are different. We place no requirements on the order that you must store your attribute-value pairs.

AttValPair is a struct we give you in provided.h.

Here’s how this might be used:

```
void makePersonARockClimber(PersonProfile& p) {
  AttValPair av(“hobby”,”rock climbing”)
  p.AddAttValPair(av);
}
```

int GetNumAttValPairs() const

This method returns the total number of distinct attribute-value pairs associated with this member.

bool GetAttVal(int attribute_num, AttValPair& attval) const

This method gets the attribute-value pair specified by attribute_num (where 0 <= attribute_num < GetNumAttValPairs()) and places it in the attval parameter. The method returns true if it successfully retrieves an attribute; otherwise, it returns false and leaves attval unchanged. If you write a loop like this

```
PersonProfile pp("Carey Nachenberg", "climberkip@gmail.com");
... // Add some attribute-value pairs to pp
for (int k = 0; k != pp.GetNumAttValPairs(); k++) {
    AttValPair av;
    GetAttVal(k, av);
    std::cout << av.attribute << " -> " << av.value << std::endl;
}
```

this spec imposes no requirement on the order in which GetAttVal provides attribute-value pairs.

AttributeTranslator Class

The AttributeTranslator class is responsible for identifying compatible attribute-value pairs for a specified input pair. For example, let's imagine that Carey Nachenberg has the following attribute-value pair in his profile:

favorite_food,del taco

The AttributeTranslator class could be used to translate the above pair into the following attribute-value pairs, each of which would increase a match's compatibility:

favorite_food,del taco   // if the other person also likes del taco, that's a good sign
favorite_food,mexican // folks that like Mexican food might also be compatible
occupation,del taco employee // Del Taco employees can get free food for me!

As you can see, a given attribute-value pair may be translated into not just one, but potentially many different attribute-value pairs that might be associated with compatible matches.

To decide what input attribute-value pairs to translate to what output attribute-value pairs, we provide a data file for your use, called ***translator.txt***. This file consists of one or more lines with the following comma-separated format:

source_attribute,source_value,compatible_attribute,compatible_value
source_attribute,source_value,compatible_attribute,compatible_value
…

For example:

```
favorite_food,del taco,favorite_food,del taco
favorite_food,del taco,favorite_food,mexican
favorite_food,del taco,occupation,del taco employee
favorite_food,taco bell,favorite_food,del taco
…
```

The above indicates that someone who’s favorite food is Del Taco has compatibility with folks who either:

- Also have a favorite food which is Del Taco
- Have indicated that Mexican food is their favorite type of food
- Have indicated that they work for Del Taco (since they can get us free tacos)

It also says that folks whose favorite food is Taco Bell, also probably like Del Taco (but given the entries in the file above, not the other way around!).

This file represent the idea “Someone with a source_attribute,source_value will probably like someone with compatible_attribute,compatible_value.” The more compatible attribute-values another person has, the better of a match they’re likely to be.

You may edit our ***translator.txt*** file if you like for testing purposes. You will not be submitting this file as part of your solution, however. Your solution must work with our provided version of the file (and, of course, with any other properly-formed file).

Your AttributeTranslator class:

- **MUST** be able to retrieve all related attribute-value pairs for a specified source attribute-value pair in better than O(N) time where N is the number of source attribute-value pairs stored in the object. So for example, O($\log_2$ N) would be acceptable (although you can do much better). For big-O analysis purposes, you may assume that there's a constant that all attribute lengths are less than. You may also assume that the number of compatible pairs any attribute-value pair translates to is bounded by a constant (e.g., < 10 pairs).
- **MUST** use your RadixTree class to map source attribute-value information to compatible attribute-values (for full credit)
- **MUST NOT** use the STL map, unordered_map, multimap, or unordered_multimap types
- **MUST NOT** add any new public member functions or variables
- MAY use the STL list, vector, set, and unordered_set classes
- MAY have any private member functions or variables you choose to add

Your AttributeTranslator has the following methods:

AttributeTranslator()

This is the AttributeTranslator constructor. It must take no arguments.

~AttributeTranslator()

You may define a destructor for AttributeTranslator if you need one to free any dynamically allocated memory used by your object.

bool Load(std::string filename)

This method loads the attribute-value translation data from the data file specified by the filename parameter. The method must load the data into a data structure that enables efficient translation of attribute-value pairs (meeting the big-O requirements at the top of this section). The method must return true if the file was successfully loaded and false otherwise.

As described above, the text file consists of one or more lines (likely thousands of lines) with the following format:

    source_attribute,source_value,compatible_attribute,compatible_value

You must ignore all empty lines. You may assume that there are no extraneous spaces before or after any comma or at the beginning or end of any line, and that neither attributes nor values will have any commas in them. You may assume that the file is all lower-case and you do not need to do case-insensitive checks.

std::vector<AttValPair> FindCompatibleAttValPairs(
const AttValPair& source) const

This method must identify all compatible attribute-value pairs for the specified source attribute-value pair in an efficient manner (meeting the requirements at the top of this section) and return a vector containing them. If there are no compatible pairs, the vector returned must be empty. There is no particular order required for the AttValPairs in the vector returned. The vector returned must not contain two attribute-value pairs with the same attributes and values (i.e., no duplicates).

Given a line from the sample translation file above, e.g.,:

```
favorite_food,del taco,favorite_food,mexican
```

The first two terms represent the source attribute and source value. The last two terms represent the compatible attribute and compatible value that we want to look for in a match.

So given the data file shown in the section above, searching for a source attribute value pair of favorite_food,del_taco should return a vector containing

- favorite_food,del taco
- favorite_food,mexican
- occupation,del taco employee

since all of the above have favorite_food,del taco as their source attribute-value pair. However, the vector would not contain

- favorite_food,taco bell

because in the line in the data file shown in the section above with favorite_food,taco bell as the source attribute-value pair, favorite_food,del_taco is not the source attribute-value pair; it's the target attribute-value pair.

Consider the following example. If the following function were called with an AttributeTranslator that has loaded the data file shown in the Load section

```
void listCompatiblePairs(const AttributeTranslator& translator) {
      AttValPair att("favorite_food", "del taco");
      std::vector<AttValPair> result =
                          translator.FindCompatibleAttValPairs(att);
      if (!results.empty()) {
          std::cout << "Compatible attributes and values:" << std::endl;
          for (const auto& p: results)
              std::cout << p.attribute << " -> " << p.value << std::endl;
      }
}
```

then it should print something like the following (the order of the three lines may vary):

```
Compatible attributes and values:
favorite_food -> del taco
occupation -> del taco employee
favorite_food -> mexican
```

MemberDatabase Class

The MemberDatabase class is responsible for loading and keeping track of all of Unhinged's members (their names, email addresses, and attribute-value pairs) and making them easily searchable.

Your MemberDatabase class:

- **MUST** meet the following big-O requirements:

    - When asked to find the email addresses associated with members who have a given attribute-value pair, it must be able to deliver all members in **better** than O(P+M) time where P is the total number of distinct attribute-value pairs across the entire member population, and M is the number of members that have the searched-for attribute-value pair. So, for example $O(\log_2 P+M)$ would be acceptable.
    - When asked to get a member's PersonProfile by searching for their email address, it must be able to deliver the member's information in **better** than O(N) time where N is the total number of members in the member database.
- **MUST** be case-sensitive for all attribute-value pair lookups
- **MUST** be able to accommodate a large number of members (our provided ***members.txt*** data file has 100k members)
- **MUST** use your RadixTree class to map attribute-value pairs to email addresses
- **MUST** use your RadixTree class to map email addresses to member profiles
- **MUST NOT** use the STL map, unordered_map, multimap, or unordered_multimap types
- **MUST NOT** add any new public member functions or variables
- MAY use the STL list and vector classes
- MAY have any private member functions or variables you choose to add

Your MemberDatabase class has the following methods:

MemberDatabase()

The member database constructor.

~MemberDatabase()

You may define a destructor for MemberDatabase if you need one to free any dynamically allocated memory used by your object.

bool LoadDatabase(std::string filename)

This method loads the member database from the data file specified by the filename parameter, e.g., ***members.txt***. The method must load the data into data structures that enable efficient retrieval of email addresses (meeting the big-O requirements at the top of this section). The method must return true if the file was successfully loaded and false otherwise. If two members in the data file have the same email address, this method returns false.

The members data file is a text file with the following format:

```
Person 1's name
Person 1's email address
```

```
Count of number of attribute-value pairs for person 1
attr1,value1
attr2,value2
…
attrN,valueN

Person 2’s name
Person 2’s email address
Count of number of attribute-value pairs for person 2
attr1,value1
attr2,value2
…
attrN,valueP
```

Each member record separated by a single blank line.

For example:

```
Carey Nachenberg
climberkip@gmail.com
4
hobby,rock climbing
hobby,teaching
occupation,professor
favorite_food,del taco

David Sm0lbirg
sm0lbirg@hotmail.com
3
favorite_food,pan-fried dumplings
occupation,professor
hobby,pigeon racing
```

You may assume that there are no extraneous spaces at the beginning or ending of any line, or before or after any commas. The file may contain uppercase and lowercase letters. You may assume that there’s a single empty line separating each member in the file. You can look at our synthetically-generated ***members.txt*** file for an example of what you will have to parse.

std::vector<std::string> FindMatchingMembers(const AttValPair& input) const

This method must identify all members that have the specified input attribute-value pair in an efficient manner (meeting the requirements at the top of this section) and return a vector containing their email addresses. If there are no such members, the vector returned must be empty. There is no particular order required for the email addresses in the vector returned. The vector returned must not contain duplicate email addresses.

A key challenge here is designing a set of data structures that allow you to efficiently find all members that have a specified attribute-value pair. Give some thought to this.

For instance, for the little example above, if the caller were to call your FindMatchingMembers method with occupation,professor then the contents of the vector returned would be "climberkip@gmail.com" and "sm0lbirg@hotmail.com", in either order.

As another example, if the caller were to call your FindMatchingMembers method with hobby,pigeon racing, then the contents of the vector returned would be "sm0lbirg@hotmail.com".

const PersonProfile* GetMemberByEmail(std::string email) const

Given an email address, this method must determine if a member exists in the database with that email address, and if so, a pointer to that member's PersonProfile that is held in your MemberDatabase object; if there is no such member, this method returns nullptr. Here's how it might be called:

```
void findMemberByEmail(const MemberDatabase& md,
                       std::string member_email) {
    PersonProfile* ptr = md.GetMemberByEmail(member_email);
    if (ptr != nullptr)
       std::cout << "Found info for member: " << ptr->GetName() << std::endl;
    else
       std::cout << "No member has address " << member_email << std::endl;
}
```

A key challenge here is designing a set of data structures that allow you to efficiently find a member that has a specified email address. Give some thought to this.

MatchMaker Class

The MatchMaker class is the workhorse of this project. Given a member's email address, it must look up the attribute-value pairs associated with that member (using the MemberDatabase and PersonProfile classes), identify compatible attribute-value pairs (using the AttributeTranslator class), and then identify and rank-order compatible members for the original member that have at least the threshold number of compatible attributes. Finally, it must output these matching members.

Your MatchMaker class:

- **MUST** run as efficiently as possible - we will not state any exact big-O requirements, but try to make your code as efficient as possible - avoid O(N) algorithms where at all possible. If you code things correctly, you should be able to find matches in a fraction of a second even across hundreds of thousands of member profiles!
- **MUST NOT** add any new public member functions or variables
- MAY use ANY STL containers you like
- MAY have any private member functions or variables you choose to add

Your MatchMaker class has the following methods:

MatchMaker(const MemberDatabase& mdb, const AttributeTranslator& at);

This constructs a MatchMaker object with the indicated parameters.

~MemberDatabase()

You may define a destructor for MemberDatabase if you need one to free any dynamically allocated memory used by your object.

std::vector<EmailCount> IdentifyRankedMatches(std::string email, int threshold) const

The IdentifyRankedMatches method is responsible for:

- Taking as input an email address for a member, and a matching threshold (indicating how many compatible attribute-value pairs another member must have to be considered a good match)
- Using the provided email address to obtain the member’s attribute-value pairs (e.g., “hobby”,”eating”, etc.)
- Converting this collection of attribute-value pairs into a collection of **unique** compatible attribute-value pairs that we want to find in other members (e.g., for “hobby”,”eating” might translate to “hobby”,”cooking” as well as “occupation”,”chef”)
- Discovering the collection of members that have each such compatible attribute-value pair (Joe and Mary have “hobby”,”cooking”; Mary and Sue have “occupation”,”chef”)
- Identifying the collection of members that have at least the threshold number of compatible attribute-value pairs in common with the member we’re trying to match for

- Returning a vector of EmailCount objects (a struct we provide in provided.h), each holding the email address of a matching member that has at least the threshold number of compatible attribute-value pairs, along with the number of such compatible pairs. The EmailCount objects in the returned vector are ordered primarily in descending order of the number of compatible attribute-value pairs, with ties broken by a secondary ordering in ascending alphabetical order by email address (so if two members both have the same number of matching attribute-value pairs, then we'd order their email addresses alphabetically in the output)

Here's how your method might be called.

```
void findMatches(const MatchMaker& mm,
                 const std::string& member_email,
                 int threshold) {
    std::vector<EmailCount> results =
                mm.IdentifyRankedMatches("sm0lbirg@hotmail.com", 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;
       }
    }
}
```

One important thing to consider: If we're searching for matches on behalf of member X, it is possible that two (or more) of member X's attribute-value pairs translate into the same compatible attribute-value pair. For example:

If Carey has just two attribute-value pairs:

```
hobby,rock climbing
hobby,hiking
```

And our attribute translator had the following two translation rules:

```
hobby,rock climbing,enjoys,outdoors
hobby,hiking,enjoys,outdoors
```

Then this would translate Carey's first attribute:

```
hobby,rock climbing
```

Into:

```
    enjoys,outdoors
```

And would **also** translate Carey's second attribute:

```
    hobby,hiking
```

Into:

```
    enjoys,outdoors
```

So we'd end up with two identical compatible attributes to search for in a mate:

```
    enjoys,outdoors
    enjoys,outdoors
```

In such a case your IdentifyRankedMatches method **must** consolidate these two duplicates into a single enjoys,outdoors item before attempting to search for compatible mates. So for example, if another member, Jessie, had a single attribute-value pair of:

```
    enjoys,outdoors
```

Then this would count as a single matching attribute-value pair with Carey, not two matches.

Here is an overall example. Assuming we have the following five members:

Anisha (anisha@gmail.com) has the following attribute-value pairs:

- likes,cookies
- likes,brownies
- likes,coding
- occupation,software engineer

Tjader (tjader@gmail.com) has the following attribute value pairs:

- likes,baking
- hobby,weight training
- occupation,salesperson

Hercumur (hercumur@gmail.com) has the following attribute value pairs:

- likes,baking
- likes,reviewing code
- occupation,QA engineer

Angus (angus@gmail.com) has the following attribute value pairs:

- likes,baking
- occupation,QA engineer

Andrea (andrea@gmail.com) has the following attribute value pairs:

- hobby,mahjong
- likes,reviewing code
- occupation,QA engineer

And assuming we had the following attribute translations:

likes,cookies,likes,baking
likes,brownies,likes,baking
likes,coding,likes,reviewing code
occupation,software engineer,occupation,QA engineer
…

And we wanted to find all matching members for Anisha with a threshold of at least 2 compatible attribute-value pairs, we'd do the following:

1. Translate Anisha's attribute-value pairs into a set of three compatible attribute-value pairs:
    a. likes,baking (both likes,cookies and likes,brownies translate to likes,baking, but we consider likes,baking only once)
    b. likes,reviewing code
    c. occupation,QA engineer
2. Identify members that match likes,baking and find Tjader, Hercumur, and Angus
3. Identify members that match likes,reviewing code and find Hercumur, and Andrea
4. Identify members that match occupation,QA engineer and find Hercumur, Angus, and Andrea
5. So Hercumur has 3 compatible attribute-value pairs, Angus has 2 compatible pairs, Andrea has 2 compatible pairs, and Tjader has 1 compatible pair
6. We'd then output the following member emails and counts in this order:
    a. hercumur@gmail.com, 3
    b. andrea@gmail.com, 2
    c. angus@gmail.com, 2

Note: Andrea comes before Cyril because of "andrea@gmail.com" < "angus@gmail.com" lexicographically. Also notice that Tjader is not in the output, since he only has 1 match which is less than the threshold of 2. Final note: This method must never return the member who searched for matches as a match for themselves!

RadixTree Class

You MUST write a templated class named *RadixTree* that implements a Radix Tree data structure that can efficiently map strings to values of any templated type (it's a type of map). This video[2] explains how a Radix Tree works. Your RadixTree class must support both inserting a new item and searching for an item, but not deleting an individual item.

Here's the interface that you MUST implement in your RadixTree class:

```
template <typename ValueType>
class RadixTree {
public:
  RadixTree();
  ~RadixTree();
  void insert(std::string key, const ValueType& value);
  ValueType* search(std::string key) const;
};
```

Building a radix tree is actually a bit tricky (there are lots of edge conditions), so we recommend that you start by implementing the guts of your RadixTree class using C++'s std::map or std::unordered map. Once you get all of your other classes working using this fake version of RadixTree, then you can fully implement the RadixTree class. In the worst case (if you can't finish implementing your Radix Tree correctly), we can test your other classes with our own RadixTree implementation and you can still get much of the credit for the project.

Your RadixTree class:

- **MUST** be a class template, implemented fully in **RadixTree.h**.
- **MUST** hold a number of nodes that is proportional to the number of unique key-value pairs inserted in the Radix Tree, NOT a number of nodes that is proportional to the number of keys times the average key length.
- **MUST** have a big-O for insertion of O(K) where K is the maximum key length of a new item being inserted into the Radix Tree, and for searches, O(K) where K is the maximum key length of the items in the Radix Tree.
- **MUST** be case-sensitive for all searches
- **MUST** be able to accommodate any number of key-value pairs
- **MUST NOT** use the STL map, unordered_map, multimap, or unordered_multimap types (in your final submission)
- **MUST NOT** add any new public member functions or variables

[2] https://bruinlearn.ucla.edu/courses/109755/pages/prefix-trees-trie-and-radix-tree?module_item_id=4793722 (27-min video)
https://en.wikipedia.org/wiki/Radix_tree

- MAY avoid dealing with an empty key string
- MAY use the STL set, list and vector classes
- MAY have any private member functions or variables you choose to add

Here are the specs for your RadixTree methods:

RadixTree()

The RadixTree constructor.

~RadixTree()

You may define a destructor for RadixTree if you need one to free any dynamically allocated memory used by your object.

void insert(std::string key, const `ValueType`& value)

The insert method must update the Radix Tree to associate the specified key string with a **copy** of the passed-in value. Inserting the same item twice should simply replace the original value with the new value. The insert method needs to consider a number of special cases and edge conditions; here are a few to consider (this is not an exhaustive list of edge conditions).

- What happens if we insert “he” → value1, followed by inserting “hello” → value2
- What happens if we insert “byte” → value3, followed by inserting “by” → value4

For this assignment, you may assume that the characters in a key string can be any character encoded by an integer between 1 and 127, inclusive. You will not be expected to deal with characters in your key strings whose encoding is outside that range. (Hint: It’s OK for each of the nodes in your Radix Tree to have an array of roughly 128 pointers.)

`ValueType`* search(std::string key) const

The search method is responsible for searching your Radix Tree for the specified key. If the key is found, then the search method must return a pointer to the value associated with the key. If the specified key was not found, the method must return nullptr. If this method returns a non-null pointer, the caller is free to modify the value held within the Radix Tree, e.g.:

```
RadixTree<AttValPair> rt;
... // insert a bunch of stuff into the radix tree rt
```

```
AttValPair* ptr = rt.search("Carey Nachenberg");
ptr->attribute = "hobby";
```

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

- Back up your code to Google Drive, iCloud or some other cloud service or to a removable device every time you make progress. WE WILL NOT ACCEPT CRASHED COMPUTERS/LOST FILES AS AN EXCUSE.
- If you use the Visual Studio or XCode debugger, you will probably shave about 50% of your development time off this project. So use the debugger.
- In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project / Properties / Configuration Properties / General / Character Set
- The entire project can be completed in under 600 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
- You must not modify any of the code in the files we provide you, as you will not turn them in. Thus, if you modify them, we will not see those changes. We will incorporate the required files that you submit into a project with special test versions of the other files.
- You must not add any public member variables/functions to your derived classes. You may add private member variables/helper methods.
- Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
- For your RadixTree class, try figuring out all the special cases first and draw diagrams showing what your tree will look like in each case. Doing so will dramatically reduce the number of bugs you have (and save you hours of implementation time).
- Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
- Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
- To get full credit, you may use only those STL containers that are explicitly permitted for each class. However, if you're having trouble building a data structure from scratch, feel free to use other STL containers to help you make progress. Using banned STL containers will result in a point deduction, potentially taking your score to zero on the violating class.

If you don't think you'll be able to finish this project, then take some shortcuts. For example, if you can't get your RadixTree class working with a hand-built tree, use the STL map or unordered_map class to temporarily implement your RadixTree class so that you can proceed with implementing other classes, and go back to fixing your RadixTree class later.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., AttributeTranslator), we will provide a correct version of that class and test it with the rest of your program (e.g., we'll test our correct AttributeTranslator class with your MatchMaker class). If you implemented the rest of the program, our version of the AttributeTranslator class should work perfectly with your version of the MatchMaker class and we can give you credit for those parts of the project you completed.

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors under both g32 and either Visual Studio or clang++!

What to Turn In

You must turn in ten to twelve files. The three header files and doc are required:

PersonProfile.h	Contains your declaration of PersonProfile
PersonProfile.cpp	Contains your implementation of PersonProfile
AttributeTranslator.h	Contains your declaration of AttributeTranslator
AttributeTranslator.cpp	Contains your implementation of AttributeTranslator
MemberDatabase.h	Contains your declaration of MemberDatabase
MemberDatabase.cpp	Contains your implementation of MemberDatabase
MatchMaker.h	Contains your declaration of MatchMaker
MatchMaker.cpp	Contains your implementation of MatchMaker
RadixTree.h	Contains your Radix Tree class template and its implementation
utility.h	(optional) Contains utility function prototypes
utility.cpp	(optional) Contains utility function implementations
report.docx, report.doc, or report.txt	Contains your report

If you write a support function used by only one class, put it in that class's .cpp file. If you write a support function used by more than one class (e.g., operator<), put its prototype in utility.h and its implementation in utility.cpp. Comment any complicated part of your code.

You must submit a brief (You're welcome!) report that details what parts of the project:

- you were unable to finish
- use banned STL components (since you didn't have time to finish an implementation that doesn't use them)
- have bugs that you have not yet been able to find/fix

The report must also document how you tested your various classes. Either a paragraph about how you tested each method or a list of test cases is fine. For example, you might provide a list of items like this: “I inserted *car* then *carey*, then searched to make sure both were found in the Radix Tree.”

Grading

- 95% of your grade will be determined by the correctness of your solution and its conformance to the performance requirements
- 5% of your grade will be based on your report

Good luck!