1) There are no known bugs in my program.

2)

**DiskMultiMap**

I implemented the **DiskMultiMap** class with the hash table data structure. Specifically, I implemented the open hashing method, where keys are stored in lists attached to buckets, or the cells, of the hash table. I designed a struct Head that acts as the head of each bucket. I also designed a struct Bucket that has an integer that holds information about the beginning of the bucket. I also designed a struct Node that holds information about the key, value, and context of each item in the table. A node can be default constructed, or be constructed with a MultiMapTuple parameter. In this case, the key, value, and context members of the node are initialized with the key, value, and context of the MultiMapTuple. The Node struct also has a next private data member that “points” to the next item in the list. By “point,” I mean that it essentially acts like a pointer to the next item, but really is just an integer that holds information about the location of the next item. It is not actually a pointer. I also included some private data members in the DiskMultiMap class, such as a string that holds the file name of the file I’m working with and a BinaryFile object. I also have some helper functions and implementations with pseudocode listed below:

**size\_t generateHash(const string s)**

create a hash structure of type string

return hash(s) // returns a value that essentially hashes the string parameter

**int updateHash(const size\_t hT)**

initialize a Head

if the binary file cannot read the head at the offset of 0, cerr an error message

return the new hash value after updating the hash

**bool updateHead, updateBucket, and updateNode**

BinaryFile writes in the respective type with a given offset for each of these

Here are a couple of non-trivial implementations of the public member functions:

**bool insert(const string& key, const string& value, const string& context)**

return false if strings exceed size of 120 or if the BinaryFile is not open

Initialize a new MultiMapTuple with values key, value, and context

Initialize a node with the created MultiMapTuple object as the parameter

generate a hash for this key

Remember the bucket index by setting it equal to the generated hash

Initialize a Bucket

If BinaryFile can’t read it at the bucket index

write an error

Initialize a Head

If BinaryFile can’t read the head at offset 0

write an error

If the head didn’t delete, node index is head’s last, last gets pushed back by size of Node,

add.next is the bucket’s beginning, and the bucket’s beginning is node Index

otherwise, read in the node, index gets set to the deleted head, next’s index is the

beginning of the bucket, the beginning of the bucket becomes node’s old index

if head or bucket doesn’t get updated

return false

return updateNode, and pass in the added node and the index of new node

I also implemented the **IntelWeb** class which includes two **DiskMultiMaps** as private data members (oldmap and newmap). I also have a string that holds the current file name. Here is some pseudo code for my non-trivial implementations of methods in IntelWeb:

**bool createNew(const string& filePrefix, unsigned int maxDataItems)**

close both old and new maps

make a bucketNumber equal to the maxDataItems / loadFactor

if you can create new files in each map successfully, set filename equal to filePrefix

return true

return false

**bool openExisting(const string& filePrefix)**

close both old and new maps

if both maps can be successfully opened by the DiskMultiMap class

pass in the filePrefix into the data member that holds the file name and return true

otherwise, return false

**bool ingest(const string& telemetryFile)**

cerr an error if the teleMetryFile cannot be opened

while getline of the ifstream of the telemetryFile and of a string is true

If the ordering is incorrect, cerr an error and continue

insert the key, value, and context into the old map

insert the value, key, context into the new map

return true

**unsigned int crawl**

clear the badEntitiesFound parameter

create a queue that holds strings, an unordered set that holds strings, and a count variable

create an unordered\_map and an unordered\_set to hold the bad items and their prevalence

create a set that orders all of its InteractionTuples based on a comparator function

Make two iterators that search for the indicators in both maps. If either iterator is valid,

Search through old and new map and push indicators into queue and the set

Increment the count of malicious items

while the queue isn’t empty

remember and pop the first value off of the queue

search through the both maps for the popped value

Malicious count is added on by calling the inspect function for both maps

set badEntitiesFound equal to the vector we made inside of the method

set badInteractions equal to the vector we made inside of the method

sort the badEntitiesFound

erase all duplicates in the badInteractions vector

return the count of malicious items

**bool purge(const string& entity)**

Create an Iterator of DiskMultiMap, a MultiMapTuple, and a counter int

search for the entity in the new map using the iterator

while the iterator is valid, increment the count by calling the new map’s erase

search for the entity in the old map using the iterator

while the iterator is valid, increment the count by calling the old map’s erase

if the count of the number of items purged is greater than 0, return true

otherwise, if the count is less than or equal to 0, return false

3) I met all the Big O requirements.