Project 4
Cyber Spider

Time due: 9 PM Thursday, March 10

Before writing a single line of code, you MUST first read AND THEN RE-READ the *Requirements and Other Thoughts* section.

Introduction

Didn't you read the text in red on the previous page?

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 that 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 has been contacted by the U.S. National Cyber Command (NCC), the national cybersecurity defense service, about creating a software program to discover latent computer attack campaigns.

If you're not familiar with cyberattacks, most of them come in the form of malicious files (e.g., *badfile.exe* or *malicious.pdf*). A user is either tricked into downloading such a malicious file, or the file is injected into the victim's machine automatically (and invisibly) when the user browses a malicious website (e.g., www.badwebsite.com). Once the malicious program has been downloaded and executed on the victim's computer, it may install other malicious programs (e.g., *badfile.exe* creates and runs *spyware.exe*), gather intelligence, or hunt for other computers to attack. Occasionally, such a malicious program will also communicate back to its master "Command and Control" server on the Internet (e.g., *badfile.exe* connects to www.masterbadsrvr.com) to obtain new directives from the attacker.

According to the NCC, due to an earlier classified government program, every single government-owned computer already monitors and collects exhaustive log data on each of the following activities:

- Every time a software file F is downloaded from a particular website W to a particular computer C, the following is logged: {C, W, F}
- Every time a software file F creates another software file G on a particular computer C, the following is logged: {C, F, G}
- Every time a software file F contacts a website W on a particular computer C, the following is logged: {C, F, W}

So for example, if government computer m1001 downloaded *foo.exe* from website www.trash.com, then the government security software running on that computer would detect this activity and add a new entry to the end of its local log data file detailing the activity:

{m1001, www.trash.com, foo.exe}

Each government computer then submits its collected logs, once per hour, to NCC headquarters where they are stored in aggregate. The government now has trillions of lines of telemetry[1] log data of the type described above, and has a hypothesis that they can search through this data to identify as yet undiscovered cyberattacks.

The NCC believes that by "spidering" or "crawling" through these trillions of lines of collected telemetry log data, and correlating this data with known threat indicators (e.g., file *F175.exe* is known to be spyware, or website www.xyz.com is known to be a malicious website), that they will be able to discover unknown or latent malicious files or websites that are somehow connected/related to these known indicators.

But how?

Well, imagine that the government has collected the following three lines of activity logs from its millions of computers (along with millions of other lines, depicted by the … below):

> *{m701, www.xyz.com, F62.exe}*
> *{m987, F62.exe, F7883.exe}*
> *{m043, F7883.exe, www.somerandomwebsite.com}*
> …

The first log line indicates that the file *F62.exe* was observed being downloaded from website www.xyz.com on government computer m701. The second line indicates that file *F62.exe* was observed creating a new file called *F7883.exe* on government computer m987. Finally, the third line indicates that file F7883.exe was observed initiating a connection to a website called www.somerandomwebsite.com on government computer m043.

Now let's assume that the NCC happens to already know that website www.xyz.com is malicious (perhaps they discovered this fact during a previous investigation), and they want to discover other as yet unknown malicious files and websites that are somehow connected with this known malicious website. How might they do so?

Well, they could search through their millions of lines of log data (including the three lines above) for www.xyz.com to discover other malicious entities that are somehow related to this single known malicious entity. For example, by searching through the millions of lines of logs, they will eventually stumble upon the log line:

> *{m701, www.xyz.com, F62.exe}*

which details that file *F62.exe* was seen being downloaded from www.xyz.com. This will lead them to conclude that file *F62.exe* is likely to be malicious too, since malicious websites are known to host malicious executable files.

Similarly, once the government discovers that file *F62.exe* is likely malicious, they can continue searching through the logs. They'll eventually find this line:

[1] *Telemetry* refers to the fact that the data is gathered remotely and sent to the NCC for processing.

```
{m987, F62.exe, F7883.exe}
```

where they see file *F62.exe* creating another file called *F7883.exe* on government machine m987. They may thus infer that *F7883.exe* is likely to be malicious too, since malicious files generally create other malicious files (a spyware installer might install a spyware program, for example).

Further, once they discover that file *F7883.exe* is malicious, they can continue searching through their logs for connections to this new file. They will soon discover this log line:

```
{m043, F7883.exe, www.somerandomwebsite.com}
```

This log line indicates that file *F7883.exe* has connected to a website called www.somerandomwebsite.com on government computer m043. They may thus infer that this website is likely to be malicious as well, since malicious software often contacts malicious websites for new attacker commands.

So by starting with a single malicious entity (www.xyz.com) and repeatedly hunting for log lines that detail direct or indirect relationships with this indicator amongst millions of lines of logs, the government hopes to discover all related unknown malicious entities!

Of course, things are never so simple. What if the log line in RED were also observed amongst the government-collected log data:

```
{m701, www.xyz.com, F62.exe}
{m987, F62.exe, F7883.exe}
{m043, F7883.exe, www.somerandomwebsite.com}
{m043, F7883.exe, www.google.com}
…
```

A naïve attack discovery algorithm might infer that website www.google.com is malicious too! Why? Since we have discovered that file *F7883.exe* is malicious, and since we've observed it connecting to website www.google.com, that site must be malicious by association! Of course, this would be a false alarm that would make the results of the hunt useless: Any program that contacted www.google.com, like a browser, would be inferred to be malicious, and then any website that browser visited would be, etc. The real malicious sites and programs would be lost among the many false alarms.

To combat the problem, the government has one additional criterion that they want to use during this discovery process to reduce the possibility of such false alarms. Specifically, if an entity (e.g., www.google.com, or *chrome.exe*) is highly prevalent (e.g., found on many machines), then it's almost certainly likely to be legitimate and not an advanced attack. Why? Advanced attacks tend to have very limited distribution – to a handful or maybe dozens of victim computers – unlike legitimate software that's found on thousands or millions of computers. So the NCC has indicated that entities with high prevalence (e.g., *chrome.exe*) should never be classified as malicious, even if they have a direct or indirect connection to known malicious entities. Such a rule would exclude www.google.com from being classified as a malicious entity even though it's directly connected with a discovered malicious entity, *F7883.exe*.

That's the basic idea of what the NCC wants you to do. Starting with one or more known malicious entities, they want you to *efficiently* search through millions of lines of log data for relationships between these known entities and other entities to discover as-yet unknown malicious entities, repeating this process over and over until the entire web of an attack is discovered.

So, in your final CS32 project for Winter 2016, your goal is to build a set of C++ classes that can be used to implement an unknown attack detection system… a Cyber Spider that crawls the attack web. Your classes MUST be able to identify unknown threats (files and URLs) by discovering relationships between them and known threat indicators (known bad files and known bad URLs) through the analysis of log data of the types described above ({C,W,F}, {C,F,G}, and {C,F,W}). However, rather than requiring you to analyze trillions of lines of log data, your program will have to analyze only hundreds of thousands to millions of log lines to prove that the general approach works. Lucky you – this makes the problem much simpler[2]!

If you're able to prove to NachenSmall's reclusive and bizarre CEO, Carey Nachenberg, that you have the programming skills to build the unknown attack detection tool described in this specification, he'll hire you to build the complete project, and you'll be famous… at least among the hundred or so people who comprise the top secret National Cyber Command security community.

Malicious Entity Discovery

The NCC has specified that the following rules MUST be used to discover unknown malicious files and websites:

Given the following inputs:

1. An aggregated set of millions of lines of log data of the form described above
2. A set of known malicious entities (e.g., a list of one or more bad files and/or websites)

[2] It's impossible for a single computer to hold, or for that matter, efficiently search through trillions or more pieces of data. So in modern "big-data" analytics systems, we split up and distribute such data across hundreds or even thousands of separate computers, then use distributed analysis tools to analyze the data in a coordinated way across all of these computers. If you're interested in how this might be done at scale, search for "hadoop" and "cassandra" – these are two distributed big-data systems that would be useful in implementing a full-scale Cyber Spider system like the one described above (capable of analyzing trillions or quadrillions of pieces of data). Or even better, join Carey at Symantec - we do this kind of thing! ☺

3. A prevalence threshold value P_{good} (e.g., $P_{good} = 12$) that can be used to determine whether an entity is too popular to be considered malicious: If the number of log lines that refer to entity E is greater than or equal to P_{good}, then E is considered legitimate because it's too popular to be an advanced attack[3].

A new entity E can be determined to be malicious based on the following criteria:

1. If there exists a "download" relationship where a file F is downloaded from a website W, and website W has been identified as a malicious entity, and the downloaded file F has a prevalence lower than P_{good}, then the downloaded file MUST also be designated a malicious entity.

   For example, if www.bad.com is currently known to be malicious[4], file *foo.exe* is referenced fewer than P_{good} times within the logs, and a log line like the following exists that indicates that file *foo.exe* was downloaded from website www.bad.com on machine m1234:

   {m1234, www.bad.com, foo.exe}

   then *foo.exe* will be determined to be malicious as well.

2. If there exists a "download" relationship where a file F is downloaded from a website W, and file F has been identified as a malicious entity, and website W (from which F was downloaded) has a prevalence lower than P_{good}, then website W MUST also be designated a malicious entity.

   For example, if *bar.exe* is currently known to be malicious, and www.barbar.com occurs fewer than P_{good} times within the logs, and a log line like the following exists (which indicates that file *bar.exe* was downloaded from www.barbar.com on government computer m640):

   {m640, www.barbar.com, bar.exe}

   then www.barbar.com will be determined to be malicious as well.

3. If there exists a "creation" relationship where a file F creates a file G on a particular machine, and file F has been identified as a malicious entity, and file G has a prevalence lower than P_{good}, then the created file G MUST be designated as a malicious entity.

[3] Advanced attacks are, as a rule, of limited prevalence, since the attacker doesn't want to infect hundreds or thousands of computers and increase the odds that their threat will be discovered.

[4] The website might be known to be malicious either (a) because it was provided in the set of known malicious entities, or (b) it was subsequently discovered to be malicious by these rules.

For example, if *bar.exe* is currently known to be malicious, and *bletch.exe* occurs fewer than P_{good} times within the logs, and a log line like the following exists (which details the creation of file bletch.exe by file bar.exe on computer m989):

{m989, bar.exe, bletch.exe}

then *bletch.exe* will be determined to be malicious as well.

4. If there exists a "creation" relationship where a file F creates a file G on a particular machine, and file G has been identified as a malicious entity, and file F has a prevalence lower than P_{good}, then the creating file F MUST be designated as a malicious entity.

   For example, if *bar.exe* is currently known to be malicious, and *my_installer.exe* occurs fewer than P_{good} times within the logs, and a log line like the following exists (indicating that file *bar.exe* was created by file *my_installer.exe* on machine m721):

   {m721, my_installer.exe, bar.exe}

   then *my_installer.exe* will be determined to be malicious as well.

5. If there exists a "connect" relationship where a file F connects to a website W, and website W has been identified as a malicious entity, and the connecting file F has a prevalence lower than P_{good}, then the connecting file MUST also be designated a malicious entity.

   For example, if www.bad.com is currently known to be malicious, file *goo.exe* is referenced fewer than P_{good} times within the logs, and a log line like the following exists (indicating that goo.exe connected to www.bad.com on machine m6542):

   {m6542, goo.exe, www.bad.com}

   then *goo.exe* will be determined to be malicious as well.

6. If there exists a "connect" relationship where a file F connects to a website W, and file F has been identified as a malicious entity, and website W has a prevalence lower than P_{good}, then website W MUST also be designated a malicious entity.

   For example, if *bar.exe* is currently known to be malicious, and www.unknown.com occurs fewer than P_{good} times within the logs, and a log line like the following exists (indicating that file bar.exe connected to www.unknown.com on computer m897340):

   {m897340, bar.exe, www.unknown.com}

then website www.unknown.com MUST be determined to be malicious as well.

Based on the NCC's requirements, these six rules MUST be applied transitively. This means that every time a new entity E has been determined to be malicious, the six rules above MUST be re-applied to all relevant entities connected to E, and so on.

For example, given the following log lines collected from millions of government computers:

```
{m562, a.exe, b.exe}
{m109, c.exe, b.exe}
{m109, explorer.exe, d.exe}
{m562, c.exe, www.attacker.com}
{m1174, q.exe, www.attacker.com}
{m3455, c.exe, www.google.com}
{m3455, www.google.com, a.exe}
...
```

And given the following entity prevalence values taken from across the entire set of logs:

P(a.exe) = 2   P(b.exe) = 2   P(c.exe) = 3   P(d.exe) = 4   P(q.exe) = 1
P(www.attacker.com) = 2   P(www.google.com) = 1,352,980

And given the initial knowledge that file *a.exe* is known to be malicious, and that we want to treat all entities with prevalence larger than or equal to P_{good} = 10 as legitimate, then we may subsequently infer:

1. That b.exe is bad (through rule 3, above, *a.exe* taints *b.exe*)
2. That c.exe is bad (through rule 4, above, *b.exe* taints *c.exe*)
3. That www.attacker.com is bad (through rule 6, above, *c.exe* taints www.attacker.com)
4. That *q.exe* is bad (through rule 5, above, www.attacker.com taints *q.exe*)

This transitivity quality ensures that we discover new malicious entities that aren't necessarily directly connected to known-malicious entities. For example, in step 4 above we have determined that *q.exe* is malicious, even though there is no direct link between *q.exe* and our only originally-known bad file *a.exe*.

Also note that we do not infer that www.google.com is bad, even though it is connected to *a.exe* and *c.exe*, and these files are known or were discovered to be malicious. This is because Google's prevalence P(www.google.com) within the logs is greater than our threshold value of 10. Similarly, we don't infer that *d.exe* is malicious, since it has no direct connection to a known or inferred malicious entity.

So, based on the principle of transitivity, once we discover a new malicious entity through a connection with an existing, known malicious entity, we can use this newly-classified malicious entity to discover other, more distant malicious entities. This process continues until we've run out of new connections to/from the expanding group of all known malicious entities. In practice, since advanced attacks tend to be of limited size – targeting dozens or fewer victims, leveraging perhaps hundreds of malicious files and websites – this transitive discovery process can be performed very quickly, even when hunting across literally trillions of lines of log data… that is, if you use the right data structures and algorithms!

Organizing the Data For Efficient Discovery of Malicious Entities

So how might you go about searching through millions of lines of logs (of the form {Machine#, First Entity, Second Entity}, e.g., {m1234, foo.exe, www.foobar.com}) to discover unknown malicious entities? Well, if you wanted to use a really slow but simple approach, you could just repeatedly search through the entire collection of logs a line at a time, something like this:

Inputs:

1. A set of known-bad entities S_{bad}
2. A prevalence threshold, P_{good}
3. A log containing millions of lines, each of the form:
   {MachineID, Entity1, Entity2}
4. A prevalence function P(e) that returns the prevalence of entity e across all of the logs (i.e., if e occurs on 5 log lines, then P(e) = 5)

Algorithm:

```
Do the following:
    Starting at the beginning of the log, iterate through each log line L:
        If L.Entity1 is in set $S_{bad}$, and $P(L.entity2) < P_{good}$ then
            Add the L.Entity2 to $S_{bad}$
        If L.Entity2 is in set $S_{bad}$, and $P(L.entity1) < P_{good}$ then
            Add the L.Entity1 to $S_{bad}$
  while $S_{bad}$ has had at least one entity added during the previous loop
```

As you can see, our pseudocode's outer loop will run over and over and over, adding newly discovered entities to its S_{bad} set during each pass through the log lines. Only once the function completes a full pass through the log lines *without* discovering a new bad entity will it stop looping and return its list of discovered bad entities.

If you remember anything from our big-O lectures, you'll notice that the above function is extremely slow. For instance, consider a log file with five billion lines, and which contains the following five lines spread out amongst those five billions lines:

```
m99793, splat.exe, www.cNc.com    // file splat.exe contacted website www.cNc.com
m02238, pfft.exe, www.cNc.com     // file pfft.exe contacted website www.cNc.com
m52902, hmm.exe , pfft.exe        // file hmm.exe created file pfft.exe
m52902, hmm.exe, www.google.com   // file hmm.exe contacts website www.google.com
m00001, www.virus.com , hmm.exe   // file hmm.exe was downloaded from www.virus.com
```

Imagine if we were to run our above algorithm, starting with an S_{bad} set that includes only {www.virus.com}:

1. The first iteration of the loop would iterate through all five billion log lines and discover that file *hmm.exe* is malicious (due to its direct connection with known-bad website www.virus.com), adding it to the S_{bad} set.
2. The second iteration of the loop would iterate again through all five billion log lines and discover that file *pfft.exe* is malicious (due to its connection with newly-discovered bad file *hmm.exe*), adding it to the S_{bad} set. It would also see the connection between hmm.exe to www.google.com, but it would not add www.google.com to the S_{bad} set because of www.google.com's high prevalence.
3. The third iteration of the loop would iterate yet again through all five billion log lines and discover that website www.cNc.com is malicious (due to its connection with newly-discovered malicious file *pfft.exe*), adding it to the S_{bad} set.
4. The fourth iteration of the loop would iterate yet again through all five billion items and discover that file *splat.exe* is malicious (due to its connection with newly-discovered malicious website www.cNC.com), adding it to the S_{bad} set.
5. The fifth iteration of the loop would iterate yet again through all five billion items and discover no new malicious entities.
6. Finally, detecting that no new malicious entities were discovered during that iteration of the loop, the function would return the set of malicious items in the S_{bad} set:

   hmm.exe
   pfft.exe
   splat.exe
   www.cNc.com
   www.virus.com

Clearly, we can do this more efficiently! For example, rather than repeatedly iterating through the billions of lines of logs, could we somehow leverage a hash table or binary search tree to speed things up? Of course, the answer is yes!

For example, we could perform a single pass through our five billion log lines and construct a hash table-based *map* that associates each first entity Entity1 to its associated second entity Entity2 for all five billion log lines. Amongst the other lines in the hash table, it would contain:

www.virus.com → hmm.exe
hmm.exe → pfft.exe, www.google.com

pfft.exe → www.cNc.com
splat.exe → www.cNc.com
…

Now, once our hash table was fully constructed, if we want to discover all entities associated with a given entity X (e.g., *hmm.exe*), we can simply look up X in our hash table and find all related entities {S1, S2, …} that were seen associated with X.

So, referring back to the previous five-billion line log example, starting with the knowledge that www.virus.com is malicious, we could look it up in our hash table (in constant time), and discover that it's associated with *hmm.exe*. We could then look up *hmm.exe* in our hash table (again, in constant time) and discover that it's associated with *pfft.exe* (and google.exe, which we'd ignore due to high prevalence). We could then look up *pfft.exe* in our hash table (in constant time) and discover that it's associated with www.cNc.com.

In just a handful of steps, we could identify almost every malicious entity related in some way to our initial malicious entity www.virus.com. All we had to do was perform a single (slow) pass through our five-billion lines of logs to insert them into a hash table, and then all further processing is extremely fast.

But wait, we have a small problem! While we've discovered most of our malicious entities, our improved algorithm failed to discover *splat.exe*, because there was no mapping between www.cNc.com and *splat.exe* in our hash table (since it only maps from source entities to target entities). OK, so our solution isn't perfect, but it's a good start. Perhaps you can think about how you'd update your data structure (or create a secondary data structure) to enable detection of both forward and reverse associations!

So as you can see, by leveraging a hash table (or perhaps multiple hash tables), we can drastically speed up the search process to discover unknown malicious entities.

One of your challenges in this assignment is to figure out an efficient set of data structures and algorithms to digest the log data in order to facilitate an extremely efficient discovery of all malicious indicators. The idea is that you'll process your input log lines once and insert them into an efficient set of data structures. Then, you can repeatedly search through these efficient data structures to detect new attacks, without having to continually slog through the original log data.

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 two complete classes, detailed below:

Class #1: You need to build a multimap class called *DiskMultiMap* that implements a disk-based, open hash table-based multimap:

1. You need to create a new disk-based multimap class based on an *open hash table*. By disk-based, we mean that the hash table's data is stored entirely in a disk file rather than in your computer's RAM. We'll explain how to do this in the sections below.
2. Your multimap MUST be able to associate each key string with one or more *pairs* of string values, where each pair is of the form {S1, S2}. So for example, it MUST be able to associate "hmm.exe" with the pair of values {"pfft.exe", "m52902"}.
3. Since this is a multimap, it MUST support multiple associations with the same key. So adding:

   "hmm.exe" → {"pfft.exe", "m52902"}
   "hmm.exe" → {"pfft.exe", "m52902"}
   "hmm.exe" → {"pfft.exe", "m10001"}
   "blah.exe" → {"bletch.exe", "m0003"}

   MUST result in your multi-map holding the following mappings:

   "hmm.exe" → {"pfft.exe", "m52902"}, {"pfft.exe", "m52902"},
   {"pfft.exe", "m10001"}
   "blah.exe" → {"bletch.exe", "m0003"}

   So in the above, the string "hmm.exe" is associated with three different pairs.
4. You need to be able to efficiently search in the hash table given a key, and use your own hand-written *iterators* to iterate through discovered associations.
5. You need to be able to efficiently delete all items matching a particular {key, S1, S2} combination. For example, if the user issued a command to delete all mappings that exactly match *"hmm.exe" →{"pfft.exe", "m52902"}*, from the table above, the resulting hash table would look like this:

   "hmm.exe" →{"pfft.exe", "m10001"}
   "blah.exe" → {"bletch.exe", "m0003"}

Class #2: You need to build a class called *IntelWeb*:

1. This class MUST be able to "digest" the entire contents of one or more log data files and organize this information into an efficient disk-based data structure that can be "crawled" to discover malicious entities (the discovery process should be an improved version of the one described in the section above).
2. Given a set of known malicious entities as input, this class MUST be able to "crawl" through your previously built, disk-based data structure to discover the presence of these known malicious entities, as well as associated, unknown malicious entities within the log data. It MUST then return all discovered

malicious entities as well as context around all references to the discovered malicious entities (i.e., one or more log entries, {M#, E1, E2}, that detail each discovered malicious entity's relationship(s) with other entities, malicious or legitimate).

What Will We Provide?

1. We'll provide a header file, *InteractionTuple.h* that contains a definition for the *InteractionTuple* struct. You MUST NOT modify this file in any way, but you may include it as necessary in your other header and cpp files. **You will not submit this header file as part of your assignment – we will use our original version to test your program.**

2. We'll provide a header file, *MultiMapTuple.h* that contains a definition for the *MultiMapTuple* struct. You MUST NOT modify this file in any way, but you may include it as necessary in your other header and cpp files. **You will not submit this header file as part of your assignment – we will use our original version to test your program.**

3. We'll provide a class called *BinaryFile.h* that allows you to read and write data to a binary data file. We'll explain more about what a binary data file is in the *BinaryFile* section below. You MUST NOT modify this header file in any way, but you may include it and use it as necessary in your other header and cpp files. **You will not submit this header file as part of your assignment – we will use our original version to test your program.**

4. We'll provide you with two additional header files that define skeleton class definitions for your *DiskMultiMap* and *IntelWeb* classes:

   DiskMultiMap.h which holds our provided ***DiskMultiMap*** skeleton class.
   IntelWeb.h which holds our provided ***IntelWeb*** skeleton class.

   You will need to modify these header files as part of your assignment:

   1. **You MUST NOT add any new public member functions or public data members to these classes. Doing so will result in a ZERO score on this project.**
   2. **You MUST implement proper working versions of our provided skeleton public functions in these classes.**
   3. **You may add new private data members and private member functions to these classes, as required, to support your member function implementations.**

5. We'll provide a simple *main.cpp* file that lets you test your overall Cyber Spider implementation. You can compile this *main.cpp* file (and our other provided

header files) with your own source files to build a complete working test program. You can then run this program from a Windows command line interpreter (cmd.exe) or from a Mac OS X Terminal window. You can read all about our provide main.cpp program in the *Test Harness* Section of this document. We suggest reading the *Test Harness* section ***after*** reading the rest of the document, since it will make a lot more sense then.

6. We'll provide you with a test data generator program, called *p4gen*, that produces synthetic telemetry log data files that you can use to test your implementation. To use the *p4gen* program to create some test data, use the following command line:

```
p4gen sources.dat malicious.dat numEevents numMachines outputlog.dat
```

sources.dat is a file that we will provide that contains a list of the top thousand or so legitimate website domains on the internet (e.g., www.cbs.com, www.yahoo.com, etc.). These will be used to generate random log lines that include URLs.

The file *malicious.dat* is a file that you must create that includes any malicious telemetry log lines that you wish to have dispersed throughout your generated test log files. For example, you might include the following lines in *malicious.dat*:

```
m918471 www.virus.com/downloads virusinstaller.exe
m918471 virusinstaller.exe spyware.exe
m918471 spyware.exe www.attackercontrolnetwork.com
m1002973 unknown.exe www.attackercontrolnetwork.com
```

numEvents specifies how many total synthetic log lines (not including your malicious lines) will be produced by the *p4gen* program.

numMachines specifies how many different machines should be represented in the synthetic logs.

The *p4gen* program then generates a set of random log data lines and outputs them to the file *outputlog.dat*.

So, for example, if you ran *p4gen* like this (using our provided *sources.dat* and the *malicious.dat* file above):

```
p4gen.exe sources.dat malicious.dat 5 3 testlog1.dat
```

it might produce the following *testlog1.dat* file, which includes both the randomly generated log lines and your specified malicious log lines:

```
m918471 www.virus.com/downloads virusinstaller.exe
m1 http://delta.com/eay/ hq.exe
```

```
m2 http://nicovideo.jp/cvsc/ fd.exe
m2 fd.exe bwk.exe
m2 fd.exe dux.exe
m2 dux.exe fozv.exe
m918471 virusinstaller.exe spyware.exe
m1 http://taboola.com/ggxr/nrvy/t/ kj.exe
m918471 spyware.exe www.attackercontrolnetwork.com
m1 http://softonic.com/q/ y.exe
m0 http://feedly.com/z/ ik.exe
m1002973 unknown.exe www.attackercontrolnetwork.com
m0 ik.exe kka.exe
m0 kka.exe s.exe
m0 kka.exe http://jrj.com.cn/ycx/xt/
m0 s.exe gyp.exe
```

You can use the *p4gen* tool to generate large log files with hundreds of thousands of entries to test your program with.

Deep-dive: The BinaryFile Class

As mentioned above, we are providing you with a class named *BinaryFile* that you can use to write data to and read data from a file on disk. The data will be stored on disk in binary form as opposed to a newline-delimited text form.

You can think of a binary disk file as an analog of standard C++ vector<char>, except that the data is stored on disk instead of in RAM. The disk file starts out empty (just like a vector), but can be expanded to any size up to the capacity of your hard drive when you write data into the file.

Here’s how you might create a new binary file:

```
#include "BinaryFile.h"

int main()
{
      BinaryFile bf;                              // create a BinaryFile object

      bool success = bf.createNew("myfile.dat");  // create a new file
      if (!success)
            cout << "Error! Unable to create myfile.dat\n";
      else
            cout << "Successfully created file myfile.dat\n";
      ...
}  // bf's destructor closes the file
```

Note: If a file of the given name already exists, *createNew()* will wipe out the contents of that file, leaving it empty (0 bytes long) and opened ready for use.

Here's how you would open an existing binary data file that was created earlier (without wiping out its contents upon opening it):

```
#include "BinaryFile.h"

int main()
{
	BinaryFile bf;

	bool success = bf.openExisting("myfile.dat");
	if (!success)
		cout << "Error! Unable to find myfile.dat\n";
	else
		cout << "Successfully opened existing file myfile.dat\n";
	...
}  // bf's destructor closes the file
```

Once you have opened a binary data file, you can write data into the file at any offset you want. (The offset is the number of bytes from the beginning of the file, which is at offset 0.) To write some data, you use one of the two forms of the *BinaryFile*::*write()* method:

```
	bool write(const char* s, size_t length, BinaryFile::Offset toOffset);
	bool write(const SomeType& x, BinaryFile::Offset toOffset);
```

In both forms, the last argument is an integer, the number of bytes from the start of the file at which to start writing data. (BinaryFile::Offset is a typedef for a large integer type.) In the first form, the data written will be `length` number of characters starting with the character pointed to by `s`. In the second form, *`SomeType`* may be one of many possible types (e.g., int, double, BinaryFile::Offset, a struct consisting of an int and two character arrays, and many others); this call writes the value of `x` to the disk file (in internal binary form) .

For example, the following code writes a 5-byte string "David" into offsets 0-4 of the binary file and then writes an int whose value is 987654321 into offsets 7-10 of the binary file (7-10 because on our machine, an int is 4 bytes long in internal binary form:

```
int main()
{
	BinaryFile bf;
	if (bf.createNew("myfile.dat"))
	{
		bool success;
		success = bf.write("David", 5, 0);  // write 5 chars from "David"
		if (!success)
			cout << "Error writing 'David' to slots 0-4 of file!\n";

		int i = 987654321;
		success = bf.write(i, 7);
		if (!success)
			cout << "Error writing integer i to slot 7-10 of file!\n";
	}
}
```

The resulting binary data file would be exactly 11 bytes long and contain the following data. For clarity, the top row shows the offset where each piece of data can be found in the binary data file (but this is only for illustration; the offsets would not be stored in the file, of course).

Offsets	0	1	2	3	4	5	6	7	8	9	10
Values	‘D’	‘a’	‘v’	‘i’	‘d’	??	??	987654321			

Notice that the code above wrote five bytes of data to the start of the data file (between offsets 0-4), then wrote the value of the integer variable i at offsets 7-10. Since the program did not explicitly write any data to offsets 5 and 6 of the binary data file, these bytes in the file are unknown and their values could be anything (this is shown by the ??s). Notice that we did not ask to write a sixth character from the string, so no zero byte was written to offset 5. We are not showing exactly what byte values are at each of offsets 7 through 10, because the internal binary form of a 4-byte int may be different on different machines (if you're curious about the details, which you don't need to know for this project, see http://en.wikipedia.org/wiki/Endianness). Just know that if we read back an int starting from offset 7, that int's value will be 987654321.

You may write over existing data or append new data to an existing data file as well. For example, suppose we ran the following code after running the example just above.

```
int main()
{
        BinaryFile bf;
        if (bf.openExisting("myfile.dat")) // open previously-created data file
        {
                if ( ! bf.write("Carey", 5, 2))
                        cout << "Error writing string to file!\n";

                if ( ! bf.write("Dog", 3, 11))
                        cout << "Error writing string to file!\n";
        }
}
```

Upon completion of this second piece of code, the resulting binary data file would contain the following data:

Offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Values	‘D’	‘a’	‘C’	‘a’	‘r’	‘e’	‘y’	987654321				‘D’	‘o’	‘g’

As you can see, the string “Carey” overwrites the last three characters of “David” and replaced the two unknown characters in byte slots 5 and 6 of the file. Moreover, we expanded the file by three bytes by writing “Dog” in slots 11-13.

In addition to writing simple C strings and integers to the data file, you can also write objects of other basic types (e.g., bool, char, int, long, unsigned int, float, double, BinaryFile::Offset). In addition, you can write arrays and structs/classes consisting of these types, which can themselves contain arrays or structs/classes of these types, etc., subject to the following restrictions:

- You must not write pointers or arrays or structs/classes containing pointers.
- You must not write objects of a struct/class type containing any virtual functions.

- You must not write objects of a struct/class type containing **both** at least one public and at least one private data member. It's OK if **all** data members are public (as is typically done with a C-like struct) or **all** data members are private (as is typically done with a C++ class with interesting behavior).
- You must not write objects of a struct/class type with a destructor, copy constructor, or assignment operator that was declared and implemented by the author of the class, not the compiler. In particular, you must not write C++ strings, vectors, lists, etc.; they do not have compiler-generated destructors, for example.

Simple C-like structs not containing pointers can be written. For example, the following code saves a Student struct to a binary data file:

```
struct Student
{
        char first[7+1];    // first name up to 7 chars long; 1 char for '\0'
        int studentID;
        float GPA;
};

int main()
{
        BinaryFile bf;

        if (bf.createNew("student.dat"))
        {
                Student s;
                strcpy(s.first, "Carey"); // this is the way to copy a C string
                s.studentID = 989105343;
                s.GPA = 3.62;

                if ( ! bf.write(s, 0))
                        cout << "Error writing Student struct to file!\n";
                else
                        assert(bf.fileLength() == sizeof(Student));
        }
}
```

The assertion should never fail; the number of bytes in the binary data file, which is what the *fileLength* method returns, should be the number of bytes in a Student object, because all we ever wrote was one Student object starting at offset 0. On most machines, this length would be 16, and the file would contain the following data. For clarity, the top row shows the offset where each piece of data can be found in the binary data file, and the second row shows which field in the struct each piece of data is associated with. Neither of the top two rows would actually be stored in the binary data file; they are for illustration only. Only the third row, labeled Values, would be stored in the data file. Note that the strcpy did not touch element 6 and 7 of first. The exact values of each byte at offsets 8 through 15 depend on our machine's internal representation of ints and floats.

Offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Field	first								studentID				GPA			
Values	‘C’	‘a’	‘r’	‘e’	‘y’	‘\0’	??	??	989105343				3.62			

Once you have written data to a binary data file, you can read data from the file. To read previously-written data, you use the *BinaryFile::read()* method, which takes one of two forms:

```
        bool read(char* s, size_t length, BinaryFile::Offset toOffset);
        bool read(SomeType& x, BinaryFile::Offset toOffset);
```

In both forms, the last argument is an integer, the number of bytes from the start of the file at which to start reading data. In the first form, `length` number of characters will be read into storage starting at the character pointed to by `s`. In the second form, *`SomeType`* may be any type that is allowed to be written); this call reads a value from the disk file (in internal binary form) and stores it in `x`.

Here's an example program that opens the data file created in the previous example and reads several of the previously-saved values:

```
int main()
{
        BinaryFile bf;
        if (bf.openExisting("student.dat"))
        {
                Student s;
                if ( ! bf.read(s, 0))
                        cout << "Error reading Student struct from file!\n";
                else
                {
                        cout << "First name: " << s.first << endl;
                        cout << "Student ID: " << s.studentID << endl;
                        cout << "GPA       : " << s.GPA << endl;
                }
        }
}
```

And here's another example that just reads in the *studentID* value that had been previously stored at offsets 8-11:

```
int main()
{
        BinaryFile bf;
        if (bf.openExisting("student.dat"))
        {
                int studID;
                if ( ! bf.read(studID, 8))
                        cout << "Error reading integer from file!\n";
                else
                        cout << "Student ID: " << studID << endl;
        }
}
```

Notice how we are able to read in just a piece of the larger struct (just the integer studentID value) as long as we knew its offset (8) and its type (int). As far as the binary data file is concerned, it just holds a bunch of bytes, so you can read any data from any offset in the file you like (but the results might be nonsense if you don't pay attention to offest and type; bf.read(studID, 5) would put an unusual int value in studID, both because the bytes starting at offset 5 were not written as an int, and because of the unknown values at offsets 6 and 7).

If you attempt to read data that does not yet exist in the file (i.e., past the end of the file), the *read()* method will return false. For example, what if we tried to read the student's GPA from offsets 13-16 rather than from 12-15 where it is stored:

```
int main()
{
        BinaryFile bf;
        if (bf.openExisting("student.dat"))
        {
                float GPA;
                if ( ! bf.read(GPA, 13))
                        cout << "Error reading float from file!\n";
                else
                        cout << "GPA: " << GPA << endl;
        }
}
```

The above would write the error message, since it attempts to read 4 bytes of data from locations 13-16 in the data file. However, our previously-created data file only has a total of 16 bytes (numbered 0 through 15):

Offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Fields	First								studentID				GPA			
Values	‘C’	‘a’	‘r’	‘e’	‘y’	‘\0’	??	??	989105343				3.62			

Therefore, the *read()* method would fail, since it would only be able to read 3 of the 4 bytes it requested (bytes 13,14, and 15). Byte 16 does not exist in the current data file.

When a BinaryFile is destroyed, the file is closed, which ensures that the data written to the file is indeed safely saved. If you wish to close a BinaryFile earlier, perhaps to use the same object when opening the file later, you can call *BinaryFile::close()*, as in this silly example:

```
void f(BinaryFile& bf, string s)
{
          // c_str() is a string method that returns a const char*
        assert(bf.write(s.c_str(), s.size(), 10));  //  offsets 0 to 9 unknown
        if (s.size() >= 4)
        {
                char buffer[4+1];
                assert(bf.read(buffer, 4, 11);
                buffer[4] = '\0';
                if (strcmp(buffer, "avid") == 0))
                        bf.close();
        }
}

int main()
{
        BinaryFile testf;
        if (testf.createNew("test.dat"))
```

```
                f(testf, "David");
        if (testf.isOpen())
                assert(testf.write("Hello", 5, 0));
        else
        {
                if (testf.openExisting("test2.dat"))
                        cout << testf.fileLength() << endl;
        }
}
```

Notice that the method *BinaryFile::isOpen()* lets you test whether the file is open.

If a *read()* or *write()* call on a BinaryFile returns false, then most further operations on that BinaryFile will return false until you close it.

What can you do with binary files?

Using the *BinaryFile* class you can implement a disk-based version of virtually any data structure that can be stored in RAM. There are two differences between RAM-based and disk-based data structures:

1. ***You*** must explicitly decide where to store each data structure in the binary data file. In contrast, when you create a new variable (or use a *new* expression to allocate a dynamic object) in RAM, C++ decides where in memory to put the object.
2. You can't use pointers in a binary data file. Instead, you must use offset values (of type BinaryFile::Offset) to specify where in the data file a piece of data is located.

So, for example, here's how we might implement a simple linked list of nodes (called *DiskNode*s) in a binary data file. In this example, we assume we're using a machine where a BinaryFile::Offset value is 4 bytes long and a DiskNode is 8 bytes long. We'll store our head pointer as a BinaryFile::Offset variable in bytes 0-3 of the binary file. We'll store our first DiskNode in bytes 4-11 of the binary file. Finally, we'll store our second DiskNode in bytes 12-19 of the binary file.

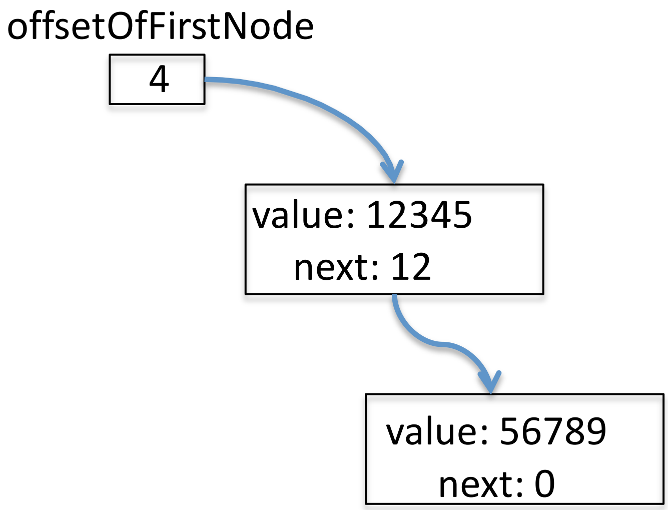
Normally we wouldn't use hard-coded offsets like 0, 4 and 12 to specify locations in the file. However, for the purposes of this example, we'll do so. In more carefully written code, `sizeof(BinaryFile::Offset)` and `sizeof(DiskNode)` would be the way we'd talk about the number of bytes occupied by objects of the indicated types.

```
struct DiskNode
{
        DiskNode(int v, BinaryFile::Offset n) : value(v), next(n) {}
        int value;
        BinaryFile::Offset next;   // instead of a DiskNode *
};

int main()
{
       BinaryFile bf;
       if (bf.createNew("linkedlist.dat"))
       {
                // First, we write the head "pointer" at the start of the file.
                // this indicates that our first node is at offset 4 in the
                // binary data file
              BinaryFile::Offset offsetOfFirstNode = 4;   // like a head pointer
              bf.write(offsetOfFirstNode, 0);

                // Save the first node at offset 4 in the binary file.
                // Note that the head "pointer" at the start of the file
                // will "point" to this node.  Also note that we specify that
                // the next node is at offset 12 in the file.
              DiskNode firstNode(12345, 12);
              bf.write(firstNode, 4);

              // Write the next node at offset 12 in the file.  Note that
              // we use a next value of zero.  This is our choice for the
              // equivalent of nullptr to indicate the end of the linked list.
              // The value zero as our choice implies we never will put
              // a DiskNode itself at offset 0 if it's part of a linked list.
              DiskNode secondNode(56789, 0);
              bf.write(secondNode, 12);
       }
}
```

This would result in the following being saved to the data file:

Offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Fields	offsetOfFirstNode				firstNode.value				firstNode.next				secondNode.value				secondNode.next			
Values	4				12345				12				56789				0			

And would represent the following logical data structure:



So as you can see, you can implement complex data structures directly within disk files. You'll need to use a technique like this to implement your own disk-based open hash table for this project!

Details: The Classes You MUST Write

You MUST write correct versions of the following classes to obtain full credit on this project. Each class MUST work correctly with our provided code, requiring no modifications to the public interfaces of the other class or to our provided structs or our provided BinaryFile.h or main.cpp files to make them work with your code. Doing so will result in a **zero score** on that part of the project.

DiskMultiMap Class

You MUST write a class named *DiskMultiMap* that lets the user:

1. Efficiently associate a given string *key* with one or more *pairs of string values*, where each pair contains a *value string* and a *context string*. That means that each association is between a single key, e.g. “foo.exe” and one or more pairs of data, where each pair holds two strings, e.g. {“bar.exe”, “m1234”}, or {“www.yahoo.com”, “m5678”}.
2. Efficiently look up items by a key string, and receive an *iterator* to enumerate all matching associations.
3. Efficiently delete existing associations with a given key.

All strings that will be added to the *DiskMultiMap* should be no more than 120 characters long (and thus representable as a C string in storage 121 bytes long). You should check for this in your code, and return an error if this is not the case.

Your implementation **MUST** use an *open hash table,* and the hash table data structure MUST be stored a binary disk file rather than in RAM.

Here’s the interface that you MUST implement in your *DiskMultiMap* class:

```
class DiskMultiMap
{
public:
      // You must implement this public nested DiskMultiMap::Iterator type
    class Iterator
    {
    public:
        Iterator();  // You must have a default constructor
        Iterator(/* You may have any parameters you like here */);
        bool isValid() const;
        Iterator& operator++();
        MultiMapTuple operator*();
    };

    DiskMultiMap();
    ~DiskMultiMap();
    bool createNew(const std::string& filename, unsigned int numBuckets);
    bool openExisting(const std::string& filename);
    void close();
    bool insert(const std::string& key, const std::string& value,
                const std::string& context);
    Iterator search(const std::string& key);
    int erase(const std::string& key, const std::string& value,
              const std::string& context);
};
```

You must implement all of this class's public member functions. You may add any private data members and private member functions you like to help you implement this class. However, you MUST NOT add any additional **public** member functions or data members to this class. Doing so will result in a score of ZERO on the project.

DiskMultiMap() and ~DiskMultiMap()

You MUST implement a constructor and destructor for your *DiskMultiMap*. The constructor MUST initialize your object. When the destructor returns, the disk file associated with the hash table must have been closed (either previously by a client calling *DiskMultiMap::close()* or by the destructor itself).

bool createNew(const std::string& filename, unsigned int numBuckets)

You MUST implement the *createNew()* method. This method MUST create an empty, open hash table in a binary disk file with the specified filename, with the specified number of empty buckets. If there is already an existing disk file with the same filename at the time that the *createNew()* method is called, then your *createNew()* method MUST overwrite the original file with a brand-new file (our *BinaryFile::createNew()* method will automatically perform this overwriting for you). Once the hash table has been created, you may then proceed to add new associations, search through it, or delete associations with the object's other methods.

Hint: An on-disk hash table typically has a header structure (that holds the number of buckets in the hash table, and perhaps information on how to locate previously-deleted nodes for reuse when the user adds a new association to the table), then an "array" of N bucket structures that each contain a file offset leading to the bucket's list of respective nodes, followed by the actual node structures themselves that actually hold the associations. Remember the limitations on what you can store in a binary disk file (see p. 18): You can not to store C++ strings in your disk-based hash table, so don't try! This means that any time you want to store a C++ string in your hash table, you'll be required to produce a C string from that string (easily done with the *c_str* member function).

If your *DiskMultiMap* object already has been used to open or create a disk-based hash table, and the *createNew()* method is called, your method MUST first close the currently-open data file, and then proceed with creating the newly specified data file.

If your method fails for any reason, it MUST return false. Otherwise, if it succeeds it MUST return true.

This method MUST run in O(B) time where B is the number of buckets in the hash table.

bool openExisting(const std::string& filename)

You MUST implement the *openExisting()* method. This method MUST open a previously-created disk-based hash table with the specified filename. If the specified disk file does not exist on the hard drive, then your method MUST return false. If your method succeeds in opening the specified file, then it MUST return true. Once the hash table has been opened, you may then proceed to search through it, add new associations, or delete associations.

If your *DiskMultiMap* object already has been used to open or create a disk-based hash table, and the *openExisting()* method is called, your method MUST first close the currently-open data file, and then proceed with opening the newly specified data file.

This method MUST run in O(1) time.

void close()

You MUST implement the *close()* method. This method is used to ensure that all data written to your hash table is safely saved to the binary disk file, and the binary disk file is closed. If a client doesn't call this method, then your *DiskMultiMap* destructor MUST close the binary disk file. Obviously, the client should only call the *close()* method if they first called the *createNew()* or *openExisting()* methods to create/open a hash table data file. Otherwise, the *close()* method should do nothing, as it should is called on an already-closed file.

This method MUST run in O(1) time.

bool insert(const std::string& key, const std::string& value, const std::string& context)

You MUST implement the *insert()* method. This method adds a new association to the disk-based hash table associating the specified *key*, e.g. “foo.exe”, with the associated *value* and *context* strings, e.g. “www.google.com” and “m12345”. Since you’re implementing a multimap, your insert function MUST be able to properly store duplicate associations. Consider this code:

```
int main()
{
        DiskMultiMap x;
        x.createNew("myhashtable.dat",100); // empty, with 100 buckets

        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m10001");
        x.insert("blah.exe", "bletch.exe", "m0003");
}
```

The above would result in your hash table holding the following associations (although not necessarily in the ordering shown below. We place no restrictions on what order your associations are stored in your hash table):

> "hmm.exe" → {"pfft.exe", "m52902"}, {"pfft.exe", "m52902"},
> {"pfft.exe", "m10001"}
> "blah.exe" → {"bletch.exe", "m0003"}

Upon successful insertion, this method returns true. Each *key*, *value* and *context* field MUST be NO MORE THAN 120 characters long, meaning that each of these fields can safely fit within a 121-byte traditional C string. Your *insert()* method MUST return false if the user tries to insert any key, value or context field with more than 120 or more characters.

Your insert method MUST first attempt to reuse storage for previously deleted nodes to add new associations, growing the disk file to add a new node to the hash table only if no previously-deleted node storage is available for reuse. Therefore, your class must keep track of where all deleted nodes are within the disk file and ensure there's a mechanism to locate these deleted nodes efficiently. When adding new data in the space that was used by a previously deleted node, you'll overwrite the spot in the disk file that held the previous node's data with your new node's data.

You MUST NOT use an in-memory data structure to keep track of the previously-deleted nodes to be reused (e.g., a vector of offsets); the information that lets you efficiently locate an available node must be stored in the disk file. (This is because (1) there might be a tremendous number of them, and (2) if our program finishes, so the *DiskMultiMap* object goes away but the disk file remains, then another program can create a *DiskMultiMap*, open it using the existing disk file, and have that *DiskMultiMap* be able to continue using the on-disk multimap, including being able to efficiently locate the available nodes that became available during the execution of the previous program.)

You may use any valid hash function you like to decide in what bucket to place each association. One convenient hash function you can use is C++'s built-in *hash* template that is defined in the C++ <functional> header file. You can search online for how to use this template to hash your strings (Stack Overflow can help). You may also use a hash function like FNV-1a. Just remember, if you define your own hash function from scratch, make sure it distributes your items uniformly across the buckets! It might make sense to do some experiments to verify such even distribution before assuming your self-defined hash function is appropriate.

Remember, this method MUST insert the new association directly into your hash table's disk file, NOT to an in-memory hash table! This means that every valid insert call will result in one or more modifications to your hash table's on-disk data structures!

This method MUST run in O(N/B) time assuming your hash table holds a generally diverse set of N items and has B buckets. If you are inserting an association with a particular key X, and the hash table already holds K associations with key X, then this method may run in O(K) time.

Iterator search(const std::string& key)

You MUST implement the *search()* method. This method is used to find all associations in the hash table that match the specified *key* string. Your method MUST return a *DiskMultiMap::Iterator* object, which is analogous to a C++ iterator. The user can then use this iterator to enumerate all associations that matched the specified *key* string. If no associations matched the specified key string, then the Iterator returned must be invalid (i.e., one for which *DiskMultiMap::Iterator::isValid()* returns false). Here's how you might use the *search()* method – look for the **bold** parts:

```
int main()
{
        DiskMultiMap x;

        x.createNew("myhashtable.dat",100); // empty, with 100 buckets

        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m10001");
        x.insert("blah.exe", "bletch.exe", "m0003");

        DiskMultiMap::Iterator it = x.search("hmm.exe");
        if (it.isValid())
        {
                cout << "I found at least 1 item with a key of hmm.exe\n";
                do
                {
                        MultiMapTuple m = *it;  // get the association
                        cout << "The key is: " << m.key << endl;
                        cout << "The value is: " << m.value << endl;
                        cout << "The context is: " << m.context << endl;
                        cout << endl;

                        ++it;  // advance iterator to the next matching item
                } while (it.isValid());
        }
}
```

The above code might print:

```
I found at least 1 item with a key of hmm.exe
The key was: hmm.exe
The value was: pfft.exe
The context was: m52902

The key was: hmm.exe
The value was: pfft.exe
The context was: m52902

The key was: hmm.exe
The value was: pfft.exe
The context was: m00001
```

Here’s another example:

```
int main()
{
        DiskMultiMap x;

        x.createNew("myhashtable.dat",100); // empty, with 100 buckets

        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m10001");
        x.insert("blah.exe", "bletch.exe", "m0003");

        DiskMultiMap::Iterator it = x.search("goober.exe");
        if ( ! it.isValid())
                cout << "I couldn’t find goober.exe\n";
}
```

The above code would print:

```
I couldn’t find goober.exe
```

As you can see, your *DiskMultiMap’s search()* method (and the returned Iterator) works similarly to the C++ *find()* methods provided by the STL associative container classes!

This method MUST run in O(N/B) time assuming your hash table holds a generally diverse set of N items and has B buckets. If you are searching for an item with a particular key X, and the hash table holds K associations with key X, then this method may run in O(K) time.

int erase(const std::string& key, const std::string& value, const std::string& context)

You MUST implement the *erase()* method. This method MUST remove **all** associations in the hash table that **exactly** match the specified *key*, *value*, and *context* strings passed in and return the number of associations removed. Your hash table MUST somehow track the location of all deleted nodes, so these nodes can be reused should one or more new associations later be added to the hash table after deletion. Here’s an example how your *erase()* method might be used:

```
int main()
{
        DiskMultiMap x;
        x.createNew("myhashtable.dat",100); // empty, with 100 buckets

        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m52902");
        x.insert("hmm.exe", "pfft.exe", "m10001");
        x.insert("blah.exe", "bletch.exe", "m0003");

        // line 1
        if (x.erase("hmm.exe", "pfft.exe", "m52902") == 2)
               cout << "Just erased 2 items from the table!\n";

        // line 2
        if (x.erase("hmm.exe", "pfft.exe", "m10001") > 0)
               cout << "Just erased at least 1 item from the table!\n";

        // line 3
        if (x.erase("blah.exe", "bletch.exe", "m66666") == 0)
               cout << "I didn't erase this item cause it wasn't there\n";
 }
```

After the insertions, the hash table contains the following associations:

“hmm.exe” → {“pfft.exe”, “m52902”}, {“pfft.exe”, “m52902”},
{“pfft.exe”, “m10001”}
“blah.exe” → {“bletch.exe”, “m0003”}

After line 1 executes, the hash table is left with the following entries:

“hmm.exe” → {“pfft.exe”, “m10001”}
“blah.exe” → {“bletch.exe”, “m0003”}

After line 2 executes, the hash table is be left with the following entry:

“blah.exe” → {“bletch.exe”, “m0003”}

After line 3 executes, the hash table is unchanged, and still contains:

“blah.exe” → {“bletch.exe”, “m0003”}

The hash table is unchanged because line 3’s *erase()* call specified a non-matching *context* value of m66666.

This method MUST run in O(N/B) time assuming your hash table holds a generally diverse set of N items and has B buckets. If you are erasing items with a particular key X, and the hash table holds K associations with key X, then this method may run in O(K) time.

The Nested DiskMultiMap::Iterator Class

An Iterator object is used to enumerate through the items in a *DiskMultiMap* after you have searched for an item using the *DiskMultiMap::search()* method. A general example of how to use the Iterator class is shown in the *DiskMultiMap::search()* section above. An Iterator pointing into a *DiskMultiMap* is conceptually pointing to an association in that *DiskMultiMap*. Incrementing that Iterator makes it point to the next association with the same key, if there is one.

If an Iterator object points into a *DiskMultiMap* when a member function other than *search()* is called on that *DiskMultiMap*, then the behavior of further operations on that Iterator, except for assigning to it or destroying it, is not defined by this spec. Roughly speaking, if a *DiskMultiMap*'s contents change, you can't assume any Iterators currently being used with it are still reliable to use. Notice that since this spec leaves the behavior undefined in this case, your implementation may do whatever it likes in this case, even crashing. Typically, you don't write any special code to detect such a situation (which is often impossible or expensive to do), so you just allow your normal code to do what it does, letting the chips fall where they may.

You must implement all of this class's public member functions. You may add any private data members and private member functions you like to help you implement this class. However, with two exceptions, you MUST NOT add any additional **public** member functions or data members to this class. Doing so will result in a score of ZERO on the project. The exceptions are (1) if you wish, you may add Iterator constructor(s) with whatever parameters you like, and (2) if the compiler-generated destructor, copy constructor, and assignment operator for Iterator don't behave correctly, you must declare and implement them.

For the descriptions below, we talk about an Iterator being in a valid or invalid state. An iterator is in a valid state if it points to an association in a DiskMultiMap; otherwise, it is in an invalid state.

Iterator()

The default constructor must create an Iterator in an invalid state. This constructor must run in O(1) time.

Other Iterator constructors

You may write other Iterator constructors with whatever parameters you like. It is your choice whether the Iterator created by any such constructor is in a valid or invalid state.

If the Iterator created is in a valid state, then it must Any such constructor MUST run in O(N/B) time or better, assuming that is being initialized to point into a hash table holding a generally diverse set of N items, with B buckets.

Iterator destructor, copy constructor, and assignment operator

The Iterator class must have a public destructor, copy constructor and assignment operator, either declared and implemented by you or left unmentioned so that the compiler will generate them for you. If you design your class well, the compiler-generated versions of these operations will do the right thing. Each of these operations must run in O(1) time.

bool isValid() const

This method MUST return true if the iterator is in a valid state, and false otherwise. This method MUST always run in O(1) time.

Iterator& operator++()

You MUST implement a prefix increment operator (but not a decrement operator or a postfix increment operator) for the Iterator class. This operator MUST do nothing if the Iterator it's called on is invalid. Otherwise, the ++ operator MUST advance the Iterator to the next association in the DiskMultiMap with the same key as the association the Iterator currently points to, if there is one; if there is no next association with the same key, then the ++ operator MUST change the Iterator's state to invalid. The method returns a reference to the Iterator it's called on.

This method MUST run in O(N/B) time or less, assuming that is called on an Iterator pointing into a hash table holding a generally diverse set of N items, with B total buckets.

MultiMapTuple operator*()

You MUST implement the unary * operator for your Iterator class – this is the dereference operator, and it allows you to examine an association pointed to by a valid Iterator, using syntax akin to that of dereferencing a C++ pointer or an STL iterator. The * operator MUST return an object of type *MultiMapTuple*:

```
struct MultiMapTuple
{
	std::string key;
	std::string value;
	std::string context;
};
```

If the Iterator is in an invalid state, each string in the *MultiMapTuple* returned is the empty string. Otherwise, the key, value and context strings of the *MultiMapTuple* returned have values equal to the key, value, and context components of the association the Iterator points to.

This method MUST run in O(1) time.

Note: Reading data from a disk file is literally thousands or millions of times slower than reading data from your computer's RAM, so consider the performance consequences of the following code:

```
void someFunc()
{
        DiskMultiMap x;
        …
        DiskMultiMap::Iterator myIt = x.search("foobar");
        if (myIt.isValid())
        {
                  // This first access results in a slow disk read, since
                  // we have to get the data from the disk
                string v = (*myIt).value;

                  // This next access retrieves the same data as the line
                  // above, so ideally should not read from the disk again,
                  // but instead use data saved inside the Iterator from
                  // the previous call
                string c = (*myIt).context;
                ...
                  // Now advance the Iterator
                ++myIt;
                if (myIt.isValid())
                {
                          // Since the Iterator points to a different
                          // association, this use of * will result in
                          // another disk read
                        v = (*myIt).value;
                }
        }
}
```

If *operator*()* stores the disk data read in private data members, then subsequent calls to *operator*()* on the same Iterator value can retireve the information from those data members instead of doing another disk read to the same node.

Of course, the programmer could have avoided the second read by coding this way:

```
                …
                MultiMapTuple m = *myIt;
                string v = m.value;
                string c = m.context;
```

Still, as the implementer of this class, we might want to shield our users from having to concern themselves with this big performance difference, especially since there is no equivalent difference with other types that use the same syntax: C++ pointers and STL itertators.

You are not required to implement this type of caching in your Iterator, but if you have time, you're encouraged to try to do so. That said, it's easy to get this kind of thing wrong, so first save a backup of your program once you get it working without caching. And if you do implement caching, be VERY careful and extensively test your implementation before turning it in!

Caching is a critical feature of much modern software, so it's helpful to understand how it works!

IntelWeb Class

The *IntelWeb* class is responsible for loading all of the log data from one or more telemetry log data files, placing this data into a disk-based data structure that can be efficiently searched by your attack discovery algorithm, and discovering new attacks by searching through this data structure. It also allows the user to purge specific telemetry data items from its data structures.

Unlike a traditional class that holds its data in RAM-based data structures and variables, your *IntelWeb* class is required to store all of its data in binary disk-based data files. This is because your class may need to hold huge amounts of data that won't fit into RAM, so you'll store this data on disk instead.

One cool thing about this approach (of saving data to disk) is that you can create an *IntelWeb* object, submit a bunch of data to it (which it'll save in an efficient way to disk), then destroy the object (perhaps because the program finished). Even though the object has been destroyed, its fundamental data structures still exist on disk, and they can be rehydrated by simply re-instantiating the class and opening the existing file (perhaps in a run of a different program or the same program launched later).

For example, a week later, you can re-instantiate your class, tell it to operate on all of its previously saved data (which is stored in the binary data file), and then you can continue using it just as you would have during the first run. All of its saved data will be available.

Your *IntelWeb* class will be implemented using your *DiskMultiMap* class.

```
class IntelWeb
{
public:
    IntelWeb();
    ~IntelWeb();
    bool createNew(const std::string& filePrefix, unsigned int maxDataItems);
    bool openExisting(const std::string& filePrefix);
    void close();
    bool ingest(const std::string& telemetryFile);
    unsigned int crawl(const std::vector<std::string>& indicators,
                       std::vector<std::string>& badEntitiesFound,
                       std::vector<InteractionTuple>& badInteractions,
                       unsigned int minPrevalenceToBeGood
                      );
    bool purge(const std::string& entity);
};
```

You must implement all of this class's public member functions. You may add any private data members and private member functions you like to help you implement this class. However, you MUST NOT add any additional **public** member functions or data members to this class. Doing so will result in a score of ZERO on the project.

IntelWeb()

The *IntelWeb* constructor MUST initialize each new *IntelWeb* object. It probably won't need to do too much work.

~IntelWeb()

The *IntelWeb* destructor MUST cause any open disk files to be closed and must free any dynamically allocated memory that hasn't already been freed.

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

The *createNew()* method MUST create a set of empty disk-based data structures to hold information about all of the telemetry log data that you'll be loading from one or more telemetry log data files. Your disk-based data structures MUST consist of one or more *DiskMultiMap*s, and you MUST not use any other disk files.

An *IntelWeb* object can organize and store its data in one or more *DiskMultiMap* files as you see fit – your goal is to design a data structure that makes searching through the data structures to discover attacks (in your *crawl()* method) as efficient as possible.

The *createNew()* method has two parameters. The first parameter is a filename prefix: All of the disk files created for this *IntelWeb* object MUST have names starting with the same prefix. For example, the call

```
IntelWeb x;
x.createNew("north-american-data", 2000000);
```

indicates that the disk-based files MUST start with the prefix *north-american-data*. If your implementation of IntelWeb uses three *DiskMultiMap*s, then *createNew()* might choose these names:

north-american-data-file-download-hash-table.dat
north-american-data-file-create-hash-table.dat
north-american-data-file-contact-hash-table.dat

or perhaps these names:

north-american-dataforward.xyz
north-american-data.reverse.pdq
north-american-data

We place no restrictions on how many *DiskMultiMap*s your *createNew()* method creates, or what data the hash tables hold. Your *IntelWeb* class may use as many *DiskMultiMap*s as you like and use them to store data in any way you like, so long as you meet the requirements specified in this document.

The second parameter to *createNew()* specifies the expected number of telemetry data items that your data structures will have to hold (e.g., 2,000,000). This parameter can help you to decide the proper size for your hash table(s). Generally, well-performing hash tables have a load factor of at most .75, so your hash tables MUST also have a planned load factor less than or equal to .75 given the value of the second parameter. (You might choose a lower load factor if it noticaeably improves your efficiency.) Once you decide on the number of buckets, you do NOT have to change that number if it later turns out the number of telemetry data items is different than the expected number that was passed to *createNew()*.

If the *createNew()* method is able to successfully create its required empty disk-based data structures, then it MUST return true, leaving the created files open for access. Otherwise, if there is any problem (e.g., one or more of your data files cannot be created), it MUST close any successfully-opened data files (so that all data files end up closed) and return false.

Once your empty disk-based data structures have been created, the user will presumably use the *ingest()* method to ingest one or more telemetry data files and store their information within your data structures, the *crawl()* method to search for attacks, and the *purge()* method to purge specific telemetry data items from the data structures.

The *createNew()* method MUST close any *DiskMultiMap*s this *IntelWeb* object might have created or opened before creating any new ones. If successful, *createNew()* MUST have overwritten any existing *DiskMultiMap* disk files with new, empty versions.

This method MUST run in O(maxDataItems) time.

bool openExisting(const std::string& filePrefix)

The *openExisting()* method is used to open an existing set of *DiskMultiMap*(s) that were previously created and populated using calls to *IntelWeb*'s *createNew()* with the given file prefix and *ingest()*. Any data previously ingested into these *DiskMultiMap*s will be available for access after the *openExisting()* method has been called.

If this method is able to successfully open its data structures, then it MUST return true. On the other hand, if there is any problem (e.g., one or more of your data files cannot be opened), this method MUST close any successfully-opened data files (so that all data files end up closed) and return false.

Once this method has been called successfully, the user may use the *ingest()* method to ingest one or more additional telemetry log data files and store their information within your data structures, the *crawl()* method to search for attacks, and the *purge()* method to purge specific telemetry data items from the data structures.

This method MUST close any *DiskMultiMap*s this *IntelWeb* object might have created or opened before opening any new ones.

This method MUST run in O(1) time.

void close()

The *close()* method is used to close any *DiskMultiMap*s created or opened by this *IntelWeb* object.

This method MUST run in O(1) time.

bool ingest(const std::string& telemetryFile)

The *ingest()* method is used to insert all the data from a telemetry log file of the specified name into your disk-based data structures. While you will ingest data from each telemetry file into your data structures once, you'll be searching through these structures (using *crawl()*) over and over and over, so the speed of ingestion is much less important than the speed at which you can search your data structures. So you MUST design your data structures to facilitate fast searching rather than trying to make insertion fast if it will slow down searching.

A telemetry log file is a text file that might contain millions of lines of log data. The lines look something like this:

```
m26500 http://delta.com/eay/ hq.exe
m491 http://nicovideo.jp/cvsc/ fd.exe
```

```
m491 fd.exe bwk.exe
m491 fd.exe dux.exe
m491 dux.exe fozv.exe
m7711 http://taboola.com/ggxr/nrvy/t/ kj.exe
m9040 http://softonic.com/q/ y.exe
m15006 http://feedly.com/z/ ik.exe
m15006 ik.exe kka.exe
m15006 kka.exe s.exe
m15006 kka.exe http://jrj.com.cn/ycx/xt/
m15006 s.exe gyp.exe
m19072 http://free.fr/bc/eju/pvab/y/ dpo.exe
m3430 http://office365.com/bnpl/ e.exe
m3430 e.exe vi.exe
…
```

The components of a line can be machine names, program names, or URLs, each of which is a string of non-whitespace characters. The lines are in one of three different formats:

File Creation Events

A *File Creation Event* indicates that a program (e.g., foo.exe) created another program (e.g., bar.exe) on a particular machine (e.g., m491). Its format is:

```
machineID <space> creatorFile <space> createdFile <newline>
```

And here's an example of what an actual example might look like:

```
m491 foo.exe bar.exe
```

File Download Events

A *File Download Event* indicates that a given URL (e.g., http://www.viruses.com/downloads) sent a program (e.g., foo.exe) down to a particular machine (e.g., m7711). Its format is:

```
machineID <space> URL <space> downloadedFile <newline>
```

And here's an example of what an actual example might look like:

```
m7711 http://www.viruses.com/downloads foo.exe
```

File Contact Events

A *File Contact Event* indicates that a given program (e.g., foo.exe) contacted a specific

URL (e.g., http://www.viruses.com/downloads) on the Internet on a particular machine (e.g., m15006). Its format is:

```
machineID <space> programFile <space> URL <newline>
```

And here's an example of what an actual example might look like:

```
m15006 foo.exe http://www.viruses.com/downloads
```

Here's how the *ingest()* method might be used:

```
void someFunc()
{
	IntelWeb x;
	x.createNew("mydata",10000); // create a new set of data files
	…
	if (x.ingest("January-telemetry.dat"))
		cout << "Loaded data from January\n";
	if (x.ingest("February-telemetry.dat"))
		cout << "Loaded data from February\n";
	…
}
```

Your *ingest()* method MUST not store any of the telemetry it loads in RAM-based data structures that last beyond the lifetime of the *ingest()* method; it MUST store all of its data structures in *DiskMultiMap* data files on disk. Your *ingest()* method may use local variables to hold a little telemetry data during ingestion, but once the *ingest()* method finishes, your *IntelWeb* object MUST not hold any of the loaded telemetry data in any form in RAM (not in a RAM-based linked list, binary search tree, hash table, etc.).

Your *ingest()* method MUST ensure that it does not improperly damage or overwrite data ingested during earlier calls to *ingest()*. In other words, the user may call *ingest()* as many times as they like, and your disk-based data structures MUST maintain every single piece of data ever ingested. This is true even if the user instantiates multiple IntelWeb objects:

```
void someFunc()
{
	IntelWeb x;

	  // create a new set of data files with a prefix of mydata
	x.createNew("mydata", 10000);
	x.ingest("January-telemetry.dat");
	x.ingest("February-telemetry.dat");
	x.close();
	…
	IntelWeb y;
	  // re-open our mydata data files!
	y.openExisting("mydata"); // opens the existing data file!
	  // add March and April's data
	y.ingest("March-telemetry.dat");
	y.ingest("April-telemetry.dat");
	y.close();
}
```

After completion of the code snippet above, your *IntelWeb* class should have created a set of one or more data files that cumulatively contains January's telemetry, February's telemetry, March's telemetry and April's telemetry.

This method MUST run in O(N) time, where N is the total number of lines being ingested from each telemetry data file.

```
unsigned int crawl(const std::vector<std::string>& indicators,
                   unsigned int minPrevalenceToBeGood,
                   std::vector<std::string>& badEntitiesFound,
                   std::vector<InteractionTuple>& badInteractions)
```

The *crawl()* method is responsible for (a) **discovering and outputting an ordered vector of all malicious entities** found in the previously-ingested telemetry, and (b) **outputting an ordered vector of every interaction**[5] **discovered that includes at least one malicious entity**, by "crawling" through the ingested telemetry!

Given the following inputs:

1. An aggregated set of millions of lines of telemetry - you've already digested this telemetry and stored it in the disk-based data structures with the *ingest()* method!
2. A vector of one or more initially-known malicious entities (e.g., a vector of one or more strings with the names of bad files and/or websites) – these are passed to *crawl()* in the *indicators* parameter.
3. A prevalence threshold value P_{good} (e.g., $P_{good} = 12$) that can be used to determine whether an entity is too popular to be considered malicious: If the number of telemetry log lines that refer to entity E is greater than or equal to P_{good}, then E is considered legitimate (aka not malicious) because it's too popular to be an advanced attack – this prevalence threshold is the *minPrevalenceToBeGood* parameter.

Your *crawl()* method MUST:

1. Analyze the ingested telemetry to discover *ALL* malicious entities, based on application of the Rules below. It MUST store a list of all discovered unique malicious entities into the *badEntitiesFound* output parameter.
2. Analyze the ingested telemetry to discover all unique interactions involving at least one bad entity, and place these interactions into the *badInteractions* output parameter.

[5] Every line of telemetry represents a single "interaction." A given interaction details a single event on a single machine, e.g., file a.exe created file b.exe on machine m1234, or website www.foo.com downloaded file c.exe onto machine m5678, or file q.exe contacted website www.virus.com on machine m2352.

Your *crawl()* method MUST discover ALL malicious entities within the provided telemetry, and it MUST place these discovered malicious entities in **lexicographical order** (i.e., sorted as strings are sorted in increasing order) within the *badEntitiesFound* parameter. Your *crawl()* method MUST place entities within the *badEntitiesFound* only if they were actually observed within the ingested telemetry lines. The *badEntitiesFound* parameter MUST NOT contain ANY values other than those malware entities discovered within the telemetry, even if the *badEntitiesFound* vector was not empty upon the call to *crawl()*. The *badEntitiesFound* MUST contain only one entry for each malicious entity that was discovered (i.e., your vector MUST not contain any duplicate values). For details on the Rules to use to discover malicious entities, see the section below. If no malicious entities are discovered within the telemetry, then *badEntitiesFound* must be empty upon completion of the *crawl()* method.

Your *crawl()* method MUST also place into the *badInteractions* parameter all unique interactions that include *at least one* malicious entity. Each interaction MUST be stored in an *InteractionTuple* variable, which details what machine the interaction took place on (*context)*, the initiator of the interaction (*from*), and the target of the interaction (*to)*:

```
 struct InteractionTuple
{
        std::string from;
        std::string to;
        std::string context;
};
```

For example, if we know or discovered that file a.exe is malicious and one of our telemetry lines shows file a.exe creating file b.exe on machine m1234, then your *badInteractions* vector MUST have an *InteractionTuple* containing these values:

{from="a.exe", to="b.exe", context="m1234"}

Similarly, if we know that file q.exe is malicious and one of our telemetry lines shows file q.exe contacting good website www.google.com[6] on machine m5678, then your *badInteractions* vector MUST have an *InteractionTuple* containing these values:

{from="q.exe", to="www.google.com", context="m5678"}

Similarly, if we know that file r.exe is malicious and one of our telemetry lines shows file r.exe was downloaded from good website www.yahoo.com[7] on machine m10101, then your *badInteractions* vector MUST have an *InteractionTuple* containing these values:

{from="www.yahoo.com", to="r.exe", context="m10101"}

[6] We would consider that www.google.com is good because it has a high enough prevalence in the telemetry logs.

[7] Believe it or not, legitimate websites get compromised all the time by hackers and can be used to distribute malware attacks. So this example of a user getting a malware file from yahoo.com, while contrived, is hypothetically possible.

Upon completion of the *crawl()* method, your *badInteractions* parameter MUST contain a **unique instance** of every interaction that contains either a *from* OR a *to* item that is known or was discovered to be malicious. Your *badInteractions* MUST NOT contain any interactions that do not contain a known or discovered malicious entity that was actually observed within the telemetry logs.

If the same interaction occurs on two different machines, e.g.:

> {from="www.yahoo.com", to="r.exe", context="m10101"}
> {from="www.yahoo.com", to="r.exe", context="m09090"}

Then this is considered two unique interactions, and both MUST be added to the *badInteractions*. After all, we want to know about all interactions with malicious entities, across all machines that the government monitors. However, if you have two or more identical interactions from the same machine, you MUST only add one of them to the *badInteractions*. So if we observed these two interactions amongst our millions of log lines:

> {from="www.yahoo.com", to="r.exe", context="m10101"}
> {from="www.yahoo.com", to="r.exe", context="m10101"}

you MUST only add one of the above interactions to the *badInteractions.*

Items stored in your *badInteractions* MUST be ordered first by their *context* (i.e., the machine they occurred on, m1234), second by their *from* field (i.e., the initiator of the activity, www.yahoo.com), and third by their *to* field (i.e., the target of the activity, r.exe). The *badInteractions* MUST only contain one entry for each malicious interaction (i.e., your vector may not contain any duplicates).

Your *crawl()* method MUST return the number of malicious *entities* it discovered within its ingested telemetry. This includes both known malicious entities (i.e., those passed within the *indicators* parameter) that were found within the telemetry lines, as well as newly discovered malicious entities that were discovered using the Rules in the next section. **Note: You should not return the number of interactions discovered, but the number of malicious entities!**

You may assume that all filenames, websites, and machine names are case sensitive. For example, "m1234" and "M1234" would represent two different machines, and "a.exe", and "A.EXE" would represent two different files.

While the number of ingested telemetry interactions can be so huge that we are requiring that you store them in disk files, **you may assume that the number of *malicious* entities and bad interactions, while possibly large, is still small enough for them to fit in RAM**. (We'd better let you assume this, otherwise you might think you couldn't fit them into the vector parameters that *crawl()* must fill!) **You may also assume that the**

number of entities whose prevalence is at least *minPrevalenceToBeGood* is small enough for them to fit in RAM. This means that you may use in-memory data structures (e.g., STL containers) to help you crawl through the data efficiently.

Assuming there are T telemetry lines that refer to known or discovered malicious entities within your ingested telemetry data, your *crawl()* method MUST complete its operation in O(T) time. It should complete its search VERY quickly. If it takes more than a second or two to discover all malicious entities in millions of telemetry data items, you'll know you have a poor design (so your *crawl()* implementation will receive little credit).

So how do you go about discovering new malicious entities, and how exactly should your *crawl()* method store the relationships it discovers in the *badInteractions* parameter! To find out, read on!

Rules To Discover new Malicious Entities

An entity MUST be deemed malicious by your *crawl()* method if (a) the user explicitly specifies that the entity is malicious by passing it in the *indicators* parameter AND the entity was ALSO present in one or more actual ingested telemetry log lines, or (b) the entity was determined to be malicious based on application of one or more of the following rules:

1. If there exists a "download" relationship within a telemetry line where a file F is downloaded from a website W, and website W has been identified as a malicious entity, and the downloaded file F has a prevalence lower than P_{good}, then the downloaded file MUST also be designated a malicious entity.

   For example, if www.bad.com is currently known to be malicious[8], file *foo.exe* is referenced fewer than P_{good} times within the logs, and a telemetry item like the following exists:

   {m1234, www.bad.com, foo.exe}

   then *foo.exe* will be determined to be malicious as well.

2. If there exists a "download" relationship within a telemetry line where a file F is downloaded from a website W, and file F has been identified as a malicious entity, and website W (where F was downloaded) has a prevalence lower than P, then website W MUST also be designated a malicious entity.

[8] The website might be known to be malicious either (a) because it was provided in the set of known malicious entities, or (b) it was subsequently discovered to be malicious by these rules

For example, if *bar.exe* is currently known to be malicious, and www.barbar.com occurs fewer than P_{good} times within the logs, and a log line like the following exists (which indicates that file *bar.exe* was downloaded from www.barbar.com):

{m640, www.barbar.com, bar.exe}

then www.barbar.com will be determined to be malicious as well.

3. If there exists a "creation" relationship within a telemetry line where a file F creates a file G on a particular machine, and file F has been identified as a malicious entity, and file G has a prevalence lower than P_{good}, then the created file G MUST be designated as a malicious entity.

   For example, if *bar.exe* is currently known to be malicious, and *bletch.exe* occurs fewer than P_{good} times within the logs, and a log line like the following exists (which details the creation of file *bletch.exe* by file *bar.exe*):

   {m989, bar.exe, bletch.exe}

   then *bletch.exe* will be determined to be malicious as well.

4. If there exists a "creation" relationship where a file F creates a file G on a particular machine, and file G has been identified as a malicious entity, and file F has a prevalence lower than P_{good}, then the creating file F MUST be designated as a malicious entity.

   For example, if *bar.exe* is currently known to be malicious, and *my_installer.exe* occurs fewer than P_{good} times within the logs, and a log line like the following exists (indicating that file *bar.exe* was created by file *my_installer.exe*):

   {m721, my_installer.exe, bar.exe}

   then *my_installer.exe* will be determined to be malicious as well.

5. If there exists a "connect" relationship within a telemetry line where a file F connects to a website W, and website W has been identified as a malicious entity, and the connecting file F has a prevalence lower than P_{good}, then the connecting file MUST also be designated a malicious entity.

   For example, if www.bad.com is currently known to be malicious, file *goo.exe* is referenced fewer than P_{good} times within the logs, and a log line like the following exists (indicating that *goo.exe* connected to www.bad.com):

   {m6542, goo.exe, www.bad.com}

   then *goo.exe* will be determined to be malicious as well.

6. If there exists a "connect" relationship within a telemetry line where a file F connects to a website W, and file F has been identified as a malicious entity, and website W has a prevalence lower than P_{good}, then website W MUST also be designated a malicious entity.

   For example, if *bar.exe* is currently known to be malicious, and www.unknown.com occurs fewer than P_{good} times within the logs, and a log line like the following exists (indicating that file *bar.exe* connected to www.unknown.com):

   {m897340, bar.exe, www.unknown.com}

   then www.unknown.com will be determined to be malicious as well.

Furthermore, these six rules apply transitively. This means that every time a new entity E has been determined to be malicious, the six rules above MUST be re-applied to all relevant entities that are connected by other telemetry to E, and so on.

Computing Entity Prevalence

The prevalence of an entity can be determined by counting up all of the telemetry lines that include that entity in any way, including duplicate telemetry items from the same machine. So, given the following telemetry:

```
m1 a.exe b.exe
m1 b.exe a.exe
m2 a.exe g.exe
m2 a.exe g.exe
m3 a.exe c.exe
```

The prevalence values would be:

P(a.exe) = 5
P(b.exe) = 2
P(c.exe) = 1
P(g.exe) = 2

Applying The 6 Rules to Discover Malicious Entities and Interactions: Example

For example, let's assume that your *IntelWeb* class were previously asked to *ingest()* millions of telemetry lines, including the following relevant telemetry lines:

m0562 a.exe b.exe
m0109 c.exe b.exe
m0109 explorer.exe d.exe
m0562 c.exe www.attacker.com
m0562 c.exe www.attacker.com
m1174 q.exe www.attacker.com
m3455 c.exe www.google.com
m3455 www.google.com a.exe
m0007 c.exe www.attacker.com

Let's further assume that after ingesting these millions of telemetry items, your class somehow computed the following entity prevalence values:

P(a.exe) = 2 P(b.exe) = 2 P(c.exe) = 5 P(d.exe) = 4 P(q.exe) = 1
P(www.attacker.com) = 4 P(www.google.com) = 1,352,980

Now, let's assume that the user calls your *crawl()* method, passing in the following:

indicators = {a.exe, www.rare-malicious-website.com}
minPrevalenceToBeGood = 1000

That is, the user somehow learned earlier, perhaps from reading a security alert, that these two indicators:

a.exe
www.rare-malicious-website.com

are known to be associated with malicious activity. Further, the user decides that any files or websites that were observed in more than 1,000 telemetry lines cannot possibly be malicious.

Given these inputs, our *crawl()* algorithm MUST subsequently infer:

1. That *b.exe* is bad (through Rule 3, above, *a.exe* taints *b.exe* on m0562)
2. That *c.exe* is bad (through Rule 4, above, *b.exe* taints *c.exe* on m0109)
3. That www.attacker.com is bad (through Rule 6, above, *c.exe* taints www.attacker.com on m0562 or m0007)
4. That *q.exe* is bad (through Rule 5, above, www.attacker.com taints *q.exe* on m1174)

The transitivity of the rules ensures that we discover new malicious entities that aren't necessarily directly adjacent to known-malicious entities. For example, in step 4 above we have determined that q.exe is malicious, even though there is no direct link between *q.exe* and our only officially-known bad file *a.exe*.

Also note that we MUST not infer that www.google.com is bad, even though it is connected to *a.exe* and *c.exe*, and these files are known or were discovered to be malicious. This is because the prevalence of www.google.com, P(www.google.com), within the telemetry logs is greater than or equal to our threshold value of 1,000. Similarly, we don't infer that *d.exe* is malicious, since it has no direct connection to a known or inferred malicious entity.

So, based on the principle of transitivity, once we discover a new bad entity through a connection with an existing, known bad entity, we can use this newly-classified bad entity to discover other, more distant bad entities. This process continues until we've run out of new connections to/from the expanding group of all known bad entities. In practice, since advanced attacks tend to be of limited size – targeting dozens or fewer victims, leveraging at most hundreds of malicious files and websites – this transitive discovery process can be performed very quickly, even when hunting across literally trillions of lines of log data… that is, if you use the right data structures and algorithms!

After applying our rules over and over, when our *crawl()* method returns, *badEntitiesFound* will contain these entities in this order*:*

a.exe
b.exe
c.exe
q.exe
www.attacker.com

Note that the items are unique (there are not multiple copies of a.exe even though malicious file a.exe was observed in multiple lines of telemetry), and that they are ordered lexicographically. Also notice that this output doesn't include www.rare-malicious-website.com. Since this entity was not actually found within our telemetry log lines, it MUST NOT be included in the output vector of malicious entities (even though the user indicates that it was malicious by placing it in the *indicators* parameter).

Further, when *crawl()* returns, *badInteractions* will contain these discovered interactions, in this order:

m0007 c.exe www.attacker.com
m0109 c.exe b.exe
m0562 a.exe b.exe
m0562 c.exe www.attacker.com
m1174 q.exe www.attacker.com
m3455 c.exe www.google.com
m3455 www.google.com a.exe

Notice that there are no duplicate items in the list above. Further notice that all of the items are first ordered by their machine number, then by the from field, and then by their to field.

Gotchas with the crawl() Method

Be careful when writing the *crawl()* method; it's very easy to crawl in circles if you don't take precautions not to do so! For example, consider the following three telemetry items:

m1 www.virus.com b.exe
m2 b.exe c.exe
m3 c.exe www.virus.com

That is, on machine 1, www.virus.com downloads *b.exe*, and on machine 2, *b.exe* creates *c.exe*, and finally on machine 3, file *c.exe* contacts www.virus.com.

Now, let's assume that we naively apply the Rules above, assuming the user specified the website www.virus.com in the malicious indicators parameter. What would happen?

A not-so-carefully-designed algorithm would probably start by looking up its one known indicator, www.virus.com, in its data structures. Instantly, it would determine that there was one piece of telemetry where www.virus.com initiated an action of downloading a file *b.exe* (on machine m1). It would then, using Rule #1, determine that file *b.exe* is malicious.

The algorithm, having discovered a new malicious entity (*b.exe*), would look up *b.exe* in its data structures, and discover that *b.exe* initiated an action of creating a new file *c.exe* (on machine m2). It would then, using Rule #3, determine that file *c.exe* is malicious.

The algorithm, having again discovered a new malicious entity (*c.exe*), would look up *c.exe* in its data structures, and discover that *c.exe* initiated an action of connecting to a website (on machine m3). It would then, using Rule #6, determine that website www.virus.com is malicious.

A not-so-carefully-designed algorithm, having again discovered a new malicious entity (www.virus.com), might look up www.virus.com in its data structures, and discover that www.virus.com initiated an action of downloading a file *b.exe* (on machine m1). It would then, using Rule #1, determine that file *b.exe* is malicious.

And so on, and so on! Notice that we've entered into an endless loop, where we continually re-discover the same malicious entities over and over again:

www.virus.com → b.exe → c.exe → www.virus.com → b.exe → c.exe →
www.virus.com → b.exe → c.exe → www.virus.com → b.exe → c.exe →
www.virus.com → b.exe → c.exe → www.virus.com → b.exe → c.exe → …

Your *crawl()* algorithm MUST take this kind of thing into account and prevent such infinite loops.

Hints For Implementing the crawl() Method

Here's a few suggestions that might help you implement your *crawl()* method:

1. You might want to have a data structure (or two) that lets you quickly look up a given entity (e.g., *a.exe* or www.virus.com) and find all of the telemetry items associated with this entity. Your *ingest()* method could construct this data structure from your input telemetry items.

   For example, given the following telemetry:

Machine	Initiator Entity	Target Entity
m1	a.exe	b.exe
m2	b.exe	c.exe
m3	a.exe	d.exe
m4	www.bad.com	c.exe

   You could build an on-disk map data structure that associates each Initiator Entity to all related telemetry items that hold that Initiator Entity:

   a.exe → {m1,a.exe,b.exe}, {m3, a.exe, d.exe}
   b.exe → {m2, b.exe, c.exe}
   www.bad.com → {m4, www.bad.com, c.exe}

   In this way, if you know that *a.exe* is bad, you can quickly look up *a.exe* in the data structure and find the telemetry items that let you determine that *b.exe* and *d.exe* are likely bad too. And once you discover that *b.exe* is bad, you can quickly look it up and discover the telemetry item that indicates that *c.exe* is bad as well. And then we can look up *c.exe* and discover that www.bad.com is bad… oh wait – no we can't! There's no entry in our simple data structure that allows us to look up *c.exe* and discover its relationship to www.bad.com. Why? Because our map only maps from Initiator Entity to related telemetry items, not Target Entity to related telemetry items. Maybe you can figure out a way to fix this?!!?

   Notice that because the number of items involved here is proportional to the number of *all* telemetry items, you'll have to use a disk-based data structure, since there are too many items to fit in RAM.

2. If you can't figure out how to go about hunting for new malicious entities, consider using a variation of the Breadth First Search / Level Order Traversal algorithm (Homework #2 to the rescue here!). Just remember, you'll have to prevent infinite cycles/loops (as discussed in the section above). Note also that a maze solver or a tree-based traversal starts searching from one place (a start location or a root node), whereas your crawl algorithm will need to start searches from *each* of the provided malicious indicators; there's not a *single* starting point from!

Notice that because the storage needed for the items involved here is proportional to the number of *malicious* telemetry items and the number of items whose prevalence is at least *minPrevalenceToBeGood*, you'll be able to fit the necessary data structures in RAM, so can use STL containers, for example.

3. To speed up your *crawl()* method, you may wish to use caching (also known as *memoization*). Any time you perform a slow operation (for instance an operation that requires one or more slow disk reads), you can store the result away in a RAM-based *cache* data structure. The next time you have to perform a slow computation during the same crawl, before you do so, you can consult your in-memory cache data structure. If your cache already holds the answer to your computation, simply use the pre-computed result. If not, then compute the answer the slow way, and store the result in the cache for next time.

bool purge(const std::string& entity)

The purge command is used to remove ALL references to a specified entity (e.g., a filename or website) from your *IntelWeb* disk-based data structures. Subsequent crawls of your telemetry using the *crawl()* method MUST NOT find any references to the specified entry after it's been purged.

For example, let's assume that you previously ingested the following telemetry items into your *IntelWeb* log files:

Machine	**Initiator Entity**	**Target Entity**
m1	a.exe	b.exe
m1	a.exe	b.exe
m2	b.exe	c.exe
m3	a.exe	d.exe
m4	www.bad.com	c.exe

Let's further assume, for the purposes of this example, that your *IntelWeb* class uses a disk-based map data structure that associates each Initiator Entity to all related telemetry items that hold that Initiator Entity[9]:

a.exe → {m1,a.exe,b.exe}, {m1,a.exe,b.exe}, {m3, a.exe, d.exe}
b.exe → {m2, b.exe, c.exe}
www.bad.com → {m4, www.bad.com, c.exe}

And let's further assume that you then used the *purge()* method, passing in "b.exe":

[9] By the way, this proposed data organization is on the right track, but is not quite sufficient to meet all of the project's requirements.

```
void someFunc()
{
       IntelWeb x;
       …      //code that loads all of the telemetry above
       x.purge("b.exe");
}
```

Your resulting disk-based data structure should look like this after the *purge()* method finishes (struck-out items have been completely removed from the file):

a.exe → ~~{m1,a.exe,b.exe}~~, ~~{m1,a.exe,b.exe}~~, {m3, a.exe, d.exe}
~~b.exe → {m2, b.exe, c.exe}~~
www.bad.com → {m4, www.bad.com, c.exe}

Be careful when you implement this method – it's tricky! Notice that we not only had to remove those entries where b.exe was a key (e.g., the second line), but also those entries where b.exe was a value as well (e.g., on the first line) but not directly searchable from the hash table!

This method must return true if the specified item was purged from at least one place in your disk-based data file), and false otherwise (e.g., if the specified item was not found and thus could not be purged).

This method MUST run in O(M) time assuming there are M items that match the passed-in string within your disk-based data structures.

Test Harness

We have graciously provided you with a simple test harness (in *p4tester.cpp*) that lets you test your overall Cyber Spider implementation. You can compile this *p4tester.cpp* file and our other provided header files together with your source files to build a complete working tester program. You can then run this program from a Windows command line interpreter (cmd.exe) or from a Mac OS X Terminal window.

Our test harness has five different functions:

You can use it to create a new, empty telemetry database

You can use the –b option to create an empty database of telemetry information with a prescribed maximum capacity:

```
p4tester -b databasePrefix expectedMaxNumberOfItems
```

For example, if you wanted to create an empty database whose file(s) start with the prefix *mydb* and that can hold an expected maximum of 100,000 telemetry data items, you would run the following command:

```
p4tester -b mydb 100000
```

When you use the –b command, our p4*tester.cpp* code will call *IntelWeb::createNew()* to build a empty database with a prefix of *mydb.*

You can use it to add telemetry data to an existing database

After you've created an empty telemetry database with the –b option above, you can use the –i option to ingest data from one or more telemetry log files into your telemetry database:

```
p4tester -i databasePrefix telemetryLogFile
```

For example, if you wanted to ingest telemetry from two telemetry log files called j*anuary-telem.txt* and *february-telem.txt,* and add their data to your existing *mydb* database, you would run the following commands:

```
p4tester -i mydb January-telem.txt
p4tester -i mydb February-telem.txt
```

Each use of the -i command adds the telemetry logs from the specified file into the specified database by calling *IntelWeb::ingest()*. The resulting database should hold the cumulative data submitted across all –i invocations (assuming your *IntelWeb* implementation is correct!).

As described earlier, telemetry log files have lines that look like this:

```
m10672 chrome.exe videogame.exe
m10672 chrome.exe http://www.twitch.com
m10672 disk-eater.exe disk-killer.exe
m22345 firefox.exe http://www.yahoo.com
m22345 http://www.yahoo.com program.exe
m22345 http://www.disksmasher.com disk-killer.exe
m22345 http://www.disksmasher.com datakill.exe
m89241 http://www.disksmasher.com disk-shredder.exe
```

Ideally the cumulative amount of telemetry data ingested will be less than the maximum size specified in the build –b command.

You can use it to search a database for malicious entities

You can use the –s option to search for one or more known-malicious attack indicators within an existing database that you built using the –b and –i commands. This command will find all malicious indicators in your database (including the provided malicious indicators found in the telemetry, and those that are discovered because they are associated with the provided malicious indicators), and output them to a results data file:

```
p4tester -s databasePrefix indicators minGoodPrevalence results
```

For example, let's say you wanted to search through your previously-built *mydb* database for a set of known-malicious indicators (provided in the file *indicators.txt*) to identify new attack indicators that are related to these indicators. Further, let's assume that you want to specify that any file or URL with at prevalence of at least 12 in the input telemetry should be considered good, even if it's connected in the telemetry to your one or more of known-malicious indicators. Finally, let's say you want to save all discovered malicious indicators to a results data file called *stuffFound.txt*. You would do this:

```
p4tester -s mydb indicators.txt 12 stuffFound.txt
```

The *indicators.txt* file that you provide to our program should contain one or more known-malicious indicators (names of files and/or websites). For example, it might contain the following lines:

```
http://www.virus.com
disk-eater.exe
datakill.exe
http://www.stealthyattack.com
```

The resulting output file *stuffFound.txt* that our p4tester.cpp produces will have two sections, separated by an empty line:

1. A list of zero or more discovered indicators (e.g., filenames and/or URLs)
2. A list of zero or more relationships discovered in the database for all of these malicious indicators

Here's an example output data file (e.g., *stuffFound.txt*):

```
datakill.exe
disk-eater.exe
disk-killer.exe
disk-shredder.exe
http://www.disksmasher.com

m10672 disk-eater.exe disk-killer.exe
m22345 http://www.disksmasher.com datakill.exe
m22345 http://www.disksmasher.com disk-killer.exe
m89241 disk-shredder.exe http://www.google.com
m89241 http://www.disksmasher.com disk-shredder.exe
```

As you can see, the above data file lists every malicious indicator that was discovered in the telemetry as well as a list of all relationships between all of those malicious indicators and other indicators (both good and malicious), showing how each indicator is related and what machine the relationship was observed on. Both sets of results will be outputted in the same order that your *IntelWeb::crawl()* method places them in its *badEntitiesFound* and *interactionsFound* output parameters. So if both sets of results are not written in lexicographical order, you'll know your *crawl()* method has a bug.

You can use it to purge entities from a database

You can use the –p option to purge all references to one or more entities within an existing database:

```
p4tester -p databasePrefix purgeFile
```

For example, let’s say you wanted to search through your existing *mydb* database for a set of entities (provided in the file *omit.txt*) that are known to be legitimate but that are being falsely identified as malicious by your program, and delete these items from the database so that they no longer get discovered. You could do this:

```
p4tester –p mydb omit.txt
```

Your *omit.txt* file contains one or more lines each containing an indicator (the name of a file or website) that you wish to purge from your database. For example, it might contain the following lines:

```
http://www.somelegitimatewebsite.com
alegitimateprogram.exe
```

You can use it to create a graphical webpage of your results

You can use the –w option to take a results data file (created with the –s option) that lists all malicious indicators and relationships (e.g., *stuffFound.txt* from above) and generate an html file (viewable with your web browser) that visually shows the relationships between the various malicious entities:

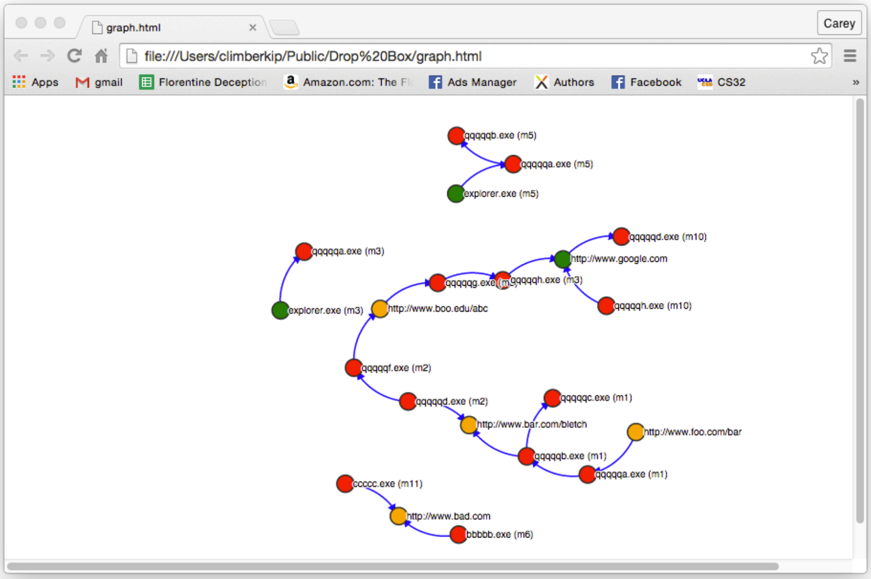
```
p4tester -w results graphtemplate.html resultgraph.html
```

(We have provided you with a file called *graphtemplate.html* that is required to produce the final *results.html* file – make sure this file is in the same directory/folder as the p4tester executable file.)

For example, let’s say you had previously searched through your database for malicious indicators with the –s command and produced a *stuffFound.txt* file with all discovered indicators and relationships. You could then use the –w command to produce an output html file that visually shows these relationships:

```
p4tester –w stuffFound.txt graphtemplate.html graph.html
```

The *graph.html* file will now contain HTML and javascript code that you can view in your web browser. To view the created html file, simply use Windows Explorer or OS X Finder to find the html file and double click on it. The file should then be loaded and rendered in your web browser and look something like this:



All discovered malicious files will be shown as red bubbles, all discovered malicious websites will be shown as orange bubbles, and all legitimate websites with connections to your malicious entities will be shown as green bubbles. As you can see, each file bubble (e.g., ccccc.exe) is tagged with the machine that the file was observed on, for example “m11” or “m10”. Arrows show the direction of the activity – if file *a.exe* created file *b.exe*, the arrow would point from the bubble labeled *a.exe* to the bubble labeled *b.exe*.

Note: Do not use Internet Explorer to view this HTML file as IE does not appear to properly render the edges (arrows) between the nodes. Instead try using Chrome or Firefox.

Requirements and Other Thoughts

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

1. You should back up your code to a flash drive or an online repository like dropbox, github, or Google Drive frequently (e.g., after creating a new function successfully). If you come to us and complain that your computer crashed and you lost all of your work, we’ll ask you where your backups are.
2. 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
3. No matter what you do and how much you finish, make sure your project builds and at least runs (even if it crashes after a while). WHATEVER YOU DO, don’t turn in code that doesn’t build.

4. Whatever you do, DO NOT add any new public member functions or public data members to the *DiskMultiMap* or *IntelWeb* classes!
5. The entire project can be completed in roughly 800 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.
6. Be sure to make copious use of the C++ STL – it can make things much easier for you!
7. If you need to define your own comparison operators for our *InteractionTuple* struct, feel free to do so! However you MUST place this function within *IntelWeb.h* as an inline function or within *IntelWeb.cpp*.
8. If you don't think you can finish the whole project in time, try to build your hash table without the ability to erase items – if you do so, its on-disk data structure will be much simpler since it won't have to keep a list of available nodes. Then leave the *DiskMultiMap::erase()* and *IntelWeb::purge()* methods for last. MAKE SURE TO BACK UP YOUR WORKING CODE PRIOR TO IMPLEMENTING THESE METHODS, AS THEY'LL RESULT IN LARGE CHANGES THAT MAY CAUSE LARGE BUGS.
9. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. For example, what file layout will you use for your disk-based open hash table? Do you need to have any additional data stored inside it to properly deal with erased nodes? How will you decide where new nodes go? Plan before you program!
10. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
11. You MUST NOT modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
12. 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.
13. Try your best to meet our big-O requirements for each method in this spec. If you can't figure out how, then solve the problem in a simpler, less efficient way, and move on. Then come back and improve the efficiency of your implementation later if you have time.
14. BACK UP FREQUENTLY! We will not accept any excuses if you lose or delete your files.

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., *DiskMultiMap*), we will provide a correct version of that class and test it with the rest of your program. If you implemented the rest of the program properly, it should work perfectly with our version of the class you couldn't get working, 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 Visual Studio and either clang++ or g++!

What to Turn In

You will turn in **five** files:

DiskMultiMap.h	Contains your disk multimap declaration/implementation
DiskMultiMap.cpp	Contains your disk multimap implementation
IntelWeb.h	Contains your intel web declaration/implementation
IntelWeb.cpp	Contains your intel web implementation
report.docx, report.doc, or report.txt	Contains your report

You MUST submit a report that describes:

1. Whether any of your classes have known bugs or other problems that we should know about. For example, if you didn't finish the *IntelWeb::purge()* method or it has bugs, tell us.
2. A high-level description of what data structures and algorithms you chose for each of your classes' non-trivial methods, and for your disk-based data structures. Brief means <1 page of description or pseudocode per class.
3. Whether or not each method satisfies our big-O requirements, and if not, what you did instead and what the big-O is for your version.

Grading

- 95% of your grade will be assigned based on the correctness of your solution.
- 5% of your grade will be based on your report.

Good luck!