Programming Assignment 4
Cyber Spider

The Project 4 specification document is complete. We're providing you with

- The source files DiskMultiMap.h, IntelWeb.h, MultiMapTuple.h, and InteractionTuple.h.
- The test data generator p4gen.cpp that you can compile, along with the files sources.txt and malicious.txt that it can use. (See pp. 15-16 of the spec to see how to use it to generate huge synthetic telemetry data files for testing your program.)
- The p4tester.cpp test harness, and its associated file graphtemplate.html.
- An implementation of DiskMultiMap.h and DiskMultiMap.cpp (using in-memory data structures) that you can use to test IntelWeb even if you are not confident that your implementation of DiskMultiMap is correct. Of course, since it uses in-memory data structures to hold all the data, the spec forbids you from using this as the DiskMultiMap implementation you turn in.

The File input writeup may be helpful when implementing *IntelWeb::ingest()*.

News since March 6:

- On p. 43, the performance requirement for *IntelWeb::crawl()* has been changed to "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 having done O(T) accesses to disk files and O(T log T) operations involving in-memory data structures." Before it had said it had to run in O(T) time, which would be rather difficult if you have to produce a sorted vector of T items!
- See above for the p4tester and an implementation of DiskMultiMap.

- The spec was inconsistent about the order of the parameters for IntelWeb::crawl(); now it's consistent, and is the order that IntelWeb.h uses and p4tester.cpp expects: minPrevalenceToBeGood is the second parameter.
- BinaryFile.h was updated to make less weird the symptom of a problem some people who were reading past the end of a BinaryFile were having. Correct DiskMultiMap implementations won't see a difference in behavior.

In the Project 4 warmup, you were asked to keep track of where removed nodes are in the disk file so that push_front can efficiently find one to reuse, if one is available. We ask you to do something similar for DiskMultiMap, but with an additional requirement: The bookkeeping information for doing this has to be on the disk, not in memory. If you did not do this for DiskList, you can either do it as part of the warmup, or get DiskMultiMap working without that requirement, then implement it for DiskList to get it working in a simpler context, and then implement it for DiskMultiMap.

The requirement expressed in terms of DiskList is this: 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 DiskList object goes away but the disk file remains, then another program can create a DiskList, open it using the existing disk file, and have that DiskList be able to continue using the on-disk list, including being able to efficiently locate the available nodes that became available during the execution of the previous program.)

If we did not require that the nodes for DiskList be stored on disk, but still insisted that no new storage for a node be allocated as long as there is at least one node that has been removed from the list and is available for reuse, then this would have been a solution.