Bradley Zhu

304627529

**CS32 CyberWeb Report**

1)

My program contains no known bugs. The only weird thing I’ve noticed is it takes longer to crawl through my files after having purged something, and I don’t know why that would happen. Both crawl and purge still work successfully though.

2)

**DiskMultiMap::Iterator**

For my Iterator for DiskMultiMap I decided to store a pointer to the binary file it’s on, an integer representing the address of the node it’s “keeping track” of, and a string representing the key that the iterator is supposed to find.

*Iterator(string key, BinaryFile \*b, int address);*

We were allowed to create additional constructors. I have 3 parameters. The string key is what we’re trying to find. The pointer to the BinaryFile helps me read in the nodes from it. The integer, address, represents the node from which we’re supposed to find other nodes with the key. My constructor initializes the Iterator in a valid state, and as such it runs in O(N/B) time, as required, because at worst it runs through one bucket, where each bucket should have N/B items.

This constructor initializes my Iterator’s private members, then iterates through all the node’s in the starting from the address passed into it. Once it reaches the end, the address will be equal to zero, so I set my m\_address to zero to show that it’s not valid. If in the loop it finds what it’s looking for, then it’ll return keeping m\_address at the value where it found the node it was looking for.

*bool isValid() const;*

All I have to do for this method is to return whether the m\_address private variable is zero or not. If it isn’t, then it’s pointing to a valid node.

*Iterator& operator++();*

I implemented a prefix increment operator for the Iterator class. This operator does nothing if the Iterator it’s called on is invalid by immediately returning a pointer to itself. Otherwise, the ++ operator advances 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. I do this by iterating through the nodes starting at the next node of the one that it’s m\_address is pointing to. If I find what I’m looking for I set m\_address to the right value (the location of the correct node), the return a pointer to itself. If there is no next association with the same key, then the ++ operator changes the Iterator’s state to invalid by setting m\_address equal to zero, then returns a pointer to itself. The method returns a reference to the Iterator it's called on.

The required time complexity is O(N/B), which I meet because at most it iterates through one complete bucket, where each bucket should have N/B items.

*MultiMapTuple operator\*();*

I implemented the unary \* operator for your Iterator class, which is a dereference operator allowing me to examine an association pointed to by a valid Iterator. The \* operator returns an object of type MultiMapTuple. If it’s not valid I return a tuple with empty values (“”). Otherwise I access the node at my current m\_address, create a MultiMapTuple with the same values, then return it.

This method runs in O(1) time, as required.

**DiskMultiMap**

For my DiskMultiMap class I have 3 private structures: a head, buckets, and nodes. Heads store information about the current end of the file (in tail), which heads are deleted (in deletedHead), and the number of buckets in my file (nbuckets). Buckets store information about the position of the first node belonging (bhead). My nodes store information about the key, value, and context, as well as an integer pointing to the next node. In addition to the structures, I also have a binary file.

*~DiskMultiMap();*

For my destructor I simply close my binary file if it isn’t already closed.

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

I implemented the createNew() method. This method creates 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 it overwrites the original file with a brand-new file.

I start by creating the file, or overwriting an existing one. Next, I needed to figure out where I could start putting nodes:

int currposition = sizeof(Head);

for every bucket:

set it’s head to point to zero (null)

write the bucket to my binary file

iterate my current position integer by the size of a bucket

}

I then fix up all my pointers of the head including the head’s tail, deletedHead, nbuckets.

This method runs in O(B) time, as required. I loop through every bucket to make them.

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

This method adds a new association to the disk-based hash table associating the specified key with the associated value and context strings. Since I’m implementing a multimap, my insert function can properly store duplicate associations.

I first make sure my file is open, then I make sure the strings are less than 120 characters long.

if I have deleted nodes:

Set location of node at a deleted node's location

update list of deleted heads

otherwise:

Set location of node at a the head's tail

update the head’s tail;

Use a has to find what bucket I need to put it in

Create the node with values specified

Updates pointers of private nodes accordingly

Set head of hashed bucket to newly created node,

Set newly created node's next pointer to old head

I return true afterwards.

This function has to run in O(N/B) time. I run in constant time.

*Iterator search(const std::string& key);*

This method is used to find all associations in the hash table that match the specified key string. I 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 just use a hash function to figure out which bucket my node (if it exists) would be in. I then just construct an Iterator with the correct parameters, and my iterator constructor does the work. Because my Iterator constructor ran in O(N/B), this also runs in O(N/B), as required.

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

This method removes all associations in the hash table that exactly match the specified key, value, and context strings passed in and returns the number of associations removed.

Hashes based on (string) key value

Searches through given bucket

while my address is valid:

If we found something with same values:

Iterate a counter

Change the old previous node's next pointer to correct pointer.

If curraddress is the head of the bucket, then change the bucket’s head pointer

Move it to the deleted list, and change the deleted list’s head to point to it

Change the node's next pointer to point to the correct next node.

return the counter

This method runs in O(N/B) time because it goes through one bucket, where each bucket contains N/B items.

**IntelWeb**

I contain two DiskMultiMaps. One mapping interactions to, and one mapping them from. For example the set (m0 a.exe b.exe) would be mapped both as (a.exe, b.exe, m0) and as (b.exe, a.exe, m0). I have a custom structure used to compare tuples in the specified way too.

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

The createNew() method creates 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 DiskMultiMaps, and you MUST not use any other disk files.

I call createNew with the correct parameters and a number of buckets equal to maxDataItems/.75. If it fails, then I close both files and return false. If it reaches the end, then it worked so I return true.

This runs in O(maxDataItems \* K) where K is a constant so it is irrelevant.

*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. I have to process the strings where they are of the format “machine from to”. I do this using sstream, and I process the file using ifstream, both of which I used stackoverflow to understand.

For every line in my file:

insert the information into toMap

insert the information (backwards) into fromMp

This must run in O(N) time where N is the total number of items being processed. I meet this because I have a for-loop to process every line which is equivalent to the total number of items.

*unsigned int crawl(const std::vector<std::string>& indicators, unsigned int*

*minPrevalenceToBeGood, std::vector<std::string>& badEntitiesFound,*

*std::vector<InteractionTuple>& interactions);*

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 interaction5 discovered that includes at least one malicious entity, by “crawling” through the ingested telemetry. It returns the number of bad malicious entities found.

I use an ordered set for InteractionTuples because it’s faster than searching through vectors, before copying everything over to the vector (conveniently in alphabetical order). I use an unordered set to store the malicious names because it is much faster to find. I use a unordered map to store the prevalence values because it’s much faster than searching through a vector. I use a queue to make sure I check through everything I need to check through.

Put the first indicators in the queue using for-loop:

Calculate and add prevalence to pMap

If it’s prevalence is above 0, add to bad entities

While the queue isn’t empty:

Get the front value of the queue and pop it off.

Search through the maps for all related entries

Iterate through every node in maps with the key

Construct an interaction tuple based on node

Make sure we didn't already find the bad entity

If prevalence isn’t in the map, calculate it and put it there

If prevalence is below minimum:

Insert into bad entities

Push onto queue

Add to badSets;

Clear out both interactions and badEntities

Shift data from the structures to the maps

Sort badEntities using built in quicksort

Crawl has to be O(T) where T is the number of telemetry lines, which I meet.

*bool purge(const std::string& entity);*

The purge command is used to remove all references to a specified entity from your IntelWeb disk-based data structures. I iterate through both my toMap, removing all references from both maps (because fromMap is just toMap flipped). If I removed anything, I’d return true.

iterator equals toMap.search(entity);

while my iterator is valid:

iterate it

remove it from toMap and fromMap

This method runs in O(M) time assuming there are M items that match the entity within my disk-based data structures.

3)

All my methods satisfy the big-O requirements.