Report

**DiskMultiMap Class:**

**Description: Return false if cannot create a new file. If can, initialize the number of number of bucket to given bucket and name to given name, write information of the head node to the file so that we can find them, write all empty nodes in the binary file to fill all buckets so that we won’t have problem reading them. The big-O of this function is O(1)**

bool DiskMultiMap::createNew(const string& filename, unsigned int num){

     close the file if it’s currently open

    return false if cannot  create a new file

    update the file name and bucket

    initialize the head with total number of buckets with offset of empty node list

    initialize an empty node

    for(all buckets)

    {

        write numBuckets number of empty nodes in the binary file

   }

    return true;

}

**Description: Return false if cannot open this file. If can, initialize the number of offset to the next available spot to add a node, taking consideration of existing node and deleted node, and name to given name. The big-O of this function is O(1)**

bool DiskMultiMap::openExisting(const std::string& filename){

    close the file if it’s currently open

    return false if cannot open the file

    taking considering of existing node and deleted node

    advance offset to the right place

    set the name

    return true;

}

**Description: Return false if the item to be inserted is too big or cannot be read. Constantly advance to the next node until we find an open spot or a spot of deleted node. If the numOfEmptyNodes is zero then I just add the Nodes at the end of the file. If there is empty spot I add that node to empty spot and set the off set of the bucket to point to that node. The big-O of this function is O(1) because I just append item to the end of file when the emptylist pointer points to zero, otherwise I just insert into removed node spot.**

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

{

    if(size too big)

        return false;

    calculate the bucket using hash function and store the info in a temp node

    if(numOfEmptyNodes == 0)

    {

        add a node of given value to the end of the file as a head node(next = 0)

        set the next node to the current head;

        write the info into the file

    }

    else

    {

            read from the list of deleted nodes the first spot available in to temp

            remove the spot from the list of deleted nodes

            link the list

            add a node of given value to the end of the file as a head node(next = 0)

            set the next node to the current head;

            write the info into the file

    }

    return true;

}

**Description: Find the bucket then linearly search the list and remove all the matched node then use the emptylist pointer points t that spot. The big-O of this function is O(N)**

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

{

    set count to keep track of number of nodes removed

use prev to keep track of the offset of the previous node

while (not the end of the list)

{

read the curr node and compare the content in curr node to parameters

if (matched)

if(deleted node is empty)

set the off set of deleted node to zero

else

set the off set of the deleted node to the nodes that empty list points to

set the off set of empty list to deleted node.

If (first node to delete in bucket list)

Set the bucket off set to the next of the deleted node

Else

Set the off set of previous node to the next of deleted node.

    return count;

}

**Description: Find the bucket number using hash function, then linearly search the whole file to find out whether that’s in the file using iterator. The Big-O of this function is O(N)**

DiskMultiMap::Iterator DiskMultiMap::search(const std::string& key)

{

    find the right offset using hash function

    get an iterator from the key

    increase iterator if the spot is empty or not target

    return it;

}

**Description: use hash function to find the right bucket if there is item in the bucket return an iterator contains the offset of the bucket pointing to. Otherwise, return a invalid iterator**

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

{

read the head of the bucket and find the position

if (the bucket doesn’t have item)

return invalid iterator;

else

return iterator containing the offset of bucket pointing to

}

**Description: Return false if cannot open this file. If can, initialize the number of offset to the next available spot to add a node, taking consideration of existing node and deleted node, and name to given name.**

DiskMultiMap::Iterator& DiskMultiMap::Iterator::operator++()

{

    advance to the next node

    read the next node

    if(it’s the last node)

    {

        setInvalid() so that cannot increase iterator

        return \*this;

    }

  return \*this

 }

**IntelWeb class:**

**Description: Create two new DiskMultiMap for this class, if anyone is unsuccessful to open then close all the Map return false. Otherwise return true.**

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

**Description: The same idea as the function of createNew.**

bool openExisting(const std::string& filePrefix);

void close();

**Description: insert all the contents into my forward map and backward map with the same Node but the key and value is reversed.**

bool ingest(const std::string& telemetryFile);

Use a temporary file to hold the file

Insert all the contents into my forward map and backward map.

**Description: I use a queue to hold all the malicious indicator then loop through my two maps with each indicator, then insert bad entities into the badentities vector. This function I fail to fulfill the big-O requirement which is O(NlogN), My big-O is O(N) because I use nested loop to go through my malicious queue and my two maps.**

**Also, this function fails to find all the malicious entities from my two maps. It can only find part of them.**

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

unsigned int minPrevalenceToBeGood,

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

std::vector<InteractionTuple>& badInteractions

);

initialize an empty queue, and empty set discovered

for all the items in indicators to this queue.

While(queue is not empty)

{

initialized two iterators for my two maps

while(the item in my forward map and prevalence of that entities is greater than minPrevalence)

{

insert that key into my discovered set;

advance my iterator

}

while (the item in my backward map and prevalence of that entities is greater than minPrevalence)

{

insert that key into my discovered set;

advance my iterator

}

pop entities from malicious queue;

}

copy all the bad entities in my discovered set into badEntitiesfound vector

copy all the bad nodes containing bad entities into my bad interaction tuple.

**Description: I use the entity to search through my two maps and remove the nodes containing that entities return true, otherwise return false. The big-O is O(N)**

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

create two iterators for my two maps

iterate through each map to find the nodes and remove it.

If there exists nodes associated with the entity then return true otherwise return false;