**Security Log Manager**

**Proposal**

**David Lovato and Arpit Patel**

**CSC316: Data Structures for Computer Scientists**

**dalovato@ncsu.edu and ampatel7@ncsu.edu**

**North Carolina State University**

**Department of Computer Science**

**January 24, 2018**

# **Black Box Test Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| **Failure to Start Program** | Precondition: The user can start the software and knows how to locate the required input files.  Available files: input1.txt, input2.txt  When specified to enter an input file, enter “filedoesntexist.txt”  Check results | The system prompts the user to enter an input file indicating that the specified file does not exist. | Output:  “File does not exist. Please enter valid file name.  To get started, please enter an input filename:” |
| **Successfully start Program** | Precondition: The user can start the software and knows how to locate the required input files.  Available files: input1.txt, input2.txt  When specified to enter an input file, enter “input1.txt”  Check results | The system prints the message that the developer chose to display upon successful start up of the program. | Output:  “Please use the following commands:  'generate profile' - Generates operational profile for a given time period.  'produce user report' - Produces activity report for a given user  Type 'quit' at any time to quit.  What do you want to do?” |
| **Produce User Report** | Precondition: The program is running with “input1.txt” as the input file.  The program prompts the user for a username for which the program will produce a user report.  When prompted, the user inputs “jtking.”  Check results | The software produces a report of activity for the provided username. The user activity information is listed in chronological order (ascending), and for each user activity, the timestamp, action, and resource are displayed. If two user activities have the same timestamp, the user activities should be displayed in alphabetical order by action + resource.  The output it should have is posted below under “Output1.txt” | Output:  “Activity Report for jtking[  01/18/2018 12:58:14PM - delete demographics information  01/18/2018 01:22:21PM - view prescription information  ]” |
| **Unable to Produce User Report** | Precondition: The program is running with “input1.txt” as the input file.  The program prompts the user for a username for which the program will produce a user report.  When prompted, the user inputs “dalovato.” | The user has no recorded user activity, the system indicates that “No activity was recorded.” | Output:  “Activity Report for dalovato[  No activity was recorded.  ]” |
| **Generate Operational Profile** | Precondition: The program is running with “input1.txt” as the input file.  The software prompts the user to enter a begin time and end time to use when generating an operational profile for all user activity between the begin and end times.  The user enters:  “01/17/2018 12:00:00AM”  “01/20/2018 12:00:00PM” | The user is presented with the operational profile, where the information is displayed in descending order of the frequency of the action + resource affected. Each action + resource should include the frequency of occurrence and percentage (rounded to 1 decimal) occurrence out of all user activity. If two action + resources have the same frequency, the action + resources should be sorted alphabetically.  The program output is listed under “Output2.txt” | Output:  “OperationalProfile[  view prescription information: frequency: 2, percentage: 40.0%  create immunization order: frequency: 1, percentage: 20.0%  delete demographics information: frequency: 1, percentage: 20.0%  delete prescription information: frequency: 1, percentage: 20.0%  ]” |
| **Generating Operational Profile for times with no activity** | Precondition: The program is running with “input1.txt” as the input file.  The software prompts the user to enter a begin time and end time to use when generating an operational profile for all user activity between the begin and end times.  The user enters:  “01/17/2018 12:00:00AM”  “01/17/2018 01:00:00AM” | No user activity occurred between the begin and end times, the system indicates “No activity was recorded.” | Output:  No activity was recorded. |
| **(Diabolical Test) Enter String for Time in Generating Operational Profile** | Precondition: The program is running with “input1.txt” as the input file.  The software prompts the user to enter a begin time and end time to use when generating an operational profile for all user activity between the begin and end times.  The user enters:  “1/17/2018 12:00:00 AM”  “January 21, 2018, 5 o’clock” | The program should say “Invalid input” (or the error message the developer chooses to output) and prompt the user for new input. | Output:  Prints stack trace of DateParseTimeException |
| **Quit Software Successfully** | Precondition: The program is running with “input1.txt” as the input file.  The user then selects the option to quit or enters “quit” into the command line. | The software quits and the input files remain unedited. | Software quits. |

**Supplemental Information:**

**Equivalence Class Partitioning:** For the “Generate Operational Profile” tests and “Produce User Report” tests, the inputs were divided into two classes, valid and invalid input. The valid input was any input that would lead to a valid output statement, such as returning the frequency of actions or actions by a certain user. Any input that led to no results being returned, such as a username or time slot with no activity, tested the invalid input class.

**Diabolical Testing:** For diabolical testing, an invalid time format was tested for the “Generate Operational Profile” function of the software. It was assumed that the user interface was a command line, and that the program was looking for a number, such as a date, long, or integer. A string was used for input to test error catching for the program.

**Boundary Value Analysis:** For boundary value analysis, the boundary values of “12:00:00 AM” and “12:00:00 PM” were tested for input times in the “Generate Operational Profile” black box test. This should test the boundaries of the AM and PM times for that algorithm.

**Supplemental files:**

**input1.txt:**

USERNAME, TIMESTAMP, ACTION, RESOURCE  
jtking, 1/18/2018 1:22:21PM, view, prescription information  
mbbrown, 1/18/2018 1:23:47PM, create, immunization order  
ssoulcrusher, 1/18/2018 1:22:01PM, delete, prescription information  
jdschmidt, 1/18/2018 1:24:21PM, view, prescription information  
jtking, 1/18/2018 12:58:14PM, delete, demographics information

**Output1.txt:**

Activity Report for jtking[  
 1/18/2018 12:58:14PM - delete demographics information  
 1/18/2018 1:22:21PM - view prescription information  
]

**Output2.txt:**

OperationalProfile[  
 view prescription information: frequency: 2, percentage: 40.0%  
 create immunization order: frequency: 1, percentage: 20.0%  
 delete demographics information: frequency: 1, percentage: 20.0%  
 delete prescription information: frequency: 1, percentage: 20.0%  
]

**Algorithm Design**

**Algorithm generateOperationalProfile(E, start, end)**  
 **Input** E, an unsorted list of log entries  
 the start time to filter the log entries  
 the end time to filter the log entries  
 **Output** S, a string representing the operational profile

Dictionary D <- empty dictionary

// Empty dictionary to hold logs

// looping through entries until end of input file, E

For each log in E do

// store the timestamp

timestamp <- log.timestamp()

//if the timestamp is correct, add it to the dictionary

If (timestamp >= start AND timestamp <= end) then

//this is the key used for the dictionary

String s <- (log.action() + “ “ + log.resource())

// it was assumed that the remove method of this dictionary could give us the object

// associated with that key, and would return null if key was not in the dictionary

Frequency <- D.remove(s)

// check to see if entry is already in dictionary; if it is, then increment the frequency and

// insert it back into the dictionary. Otherwise, add it to the dictionary.

If (frequency DOES NOT EQUAL null)

D.insert(s, freq+1)

Else

D.insert(s,1)

D.mergesort()

//if the list needs to be sorted, sort it here by frequency, then alphabetic if same freq

S <- D.toString() //return in the format need

Return S

**Algorithm getLogsForUser(E, user)**  
 **Input** E, an unsorted list of log entries  
 the user for which to get logs  
 **Output** L, a sorted list of logs for the given user

//create empty dictionary for logs entered by that user

Dictionary D <- empty dictionary

//create empty list to return

L <- Empty List

//loop through each of the logs in the log entries input, E

For each log in E do

//store the username of that log

username <- log.user()

//add it to the dictionary if it matches the user

If (user equals username)

timeStamp <- log.timestamp()

Description <- log.description()

D.insert(user, timestamp, description)

// if needed, sort the list by timestamp, then by alphabetical order by action + resource

L = mergesort(D)

return L

**Data Structures**

The abstract data types that we used for our algorithms were dictionaries. The data structure that we will be implementing would be a skip list. We chose the skip list because the algorithms were executing lots of insertions and removals, making the skip list ideal because its O(log n) runtime for insertion and removal. Additionally, the skip list is always sorted, so that saves some time as well by not having to sort at the end. The set of operations we need for our algorithms were insert(o) and remove(o), which both have O(log n) runtime. Additionally, we added an operation that returns the dictionary as a string, for the generateOperationalProfile() algorithm. Overall, by choosing a skip list, we believe that this will give us the most efficient algorithm with a O(n log(n)) runtime.

One alternative data structure we could have used to solve this problem is a search table. However, the drawback with using a search table is that the insert and removal methods have O(n) runtime, making it less ideal for our algorithms since they involve lots of insertions and removals.

# 

# 

# 

# 

# 

# 

# **Algorithm Analysis**

**Algorithm generateOperationalProfile(E, start, end)**  
 **Input** E, an unsorted list of log entries  
 the start time to filter the log entries  
 the end time to filter the log entries  
 **Output** S, a string representing the operational profile

Dictionary D <- empty dictionary (1)

For each log in E do (n)

timestamp <- log.timestamp() (1)

If (timestamp >= start AND timestamp <= end) (2)

String s <- (log.action() + “ “ + log.resource()) (2)

Frequency <- D.remove(s) (log n)

if(frequency does not equal null) (1)

D.insert(s, freq+1) (log n)

Else (1)

D.insert(s,1) (log n)

D.mergesort() (nlogn)

S <- D.toString() (n)

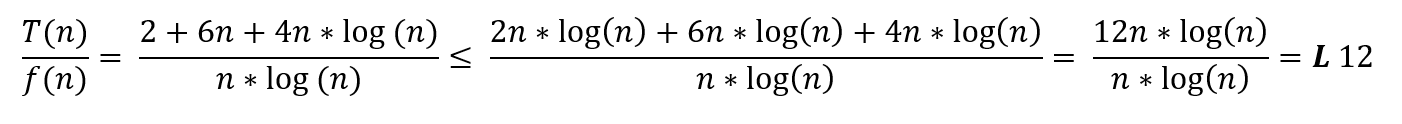
Return S (1)

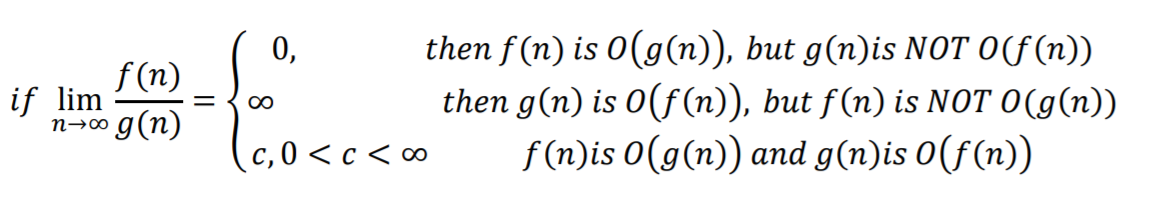
T(n) = 1 + n\*[1+2+2+log(n)+1+log(n)+1+log(n)]+n\*log(n)+1

T(n) = 2 + 6n + 4n\*log(n)

Big oh = O(n log(n))

Proof T(n) is O(f(n))





Since 12 is a constant, T(n) is O(f(n)).

**Algorithm getLogsForUser(E, user)**  
 **Input** E, an unsorted list of log entries  
 the user for which to get logs  
 **Output** L, a sorted list of logs for the given user

Dictionary D <- empty dictionary (1)

L <- Empty List (1)

For each log in E do (n)

username <- log.user() (1)

If (user equals username) (1)

timeStamp <- log.timestamp() (1)

Description <- log.description() (1)

D.insert(user, timestamp, description) (log n)

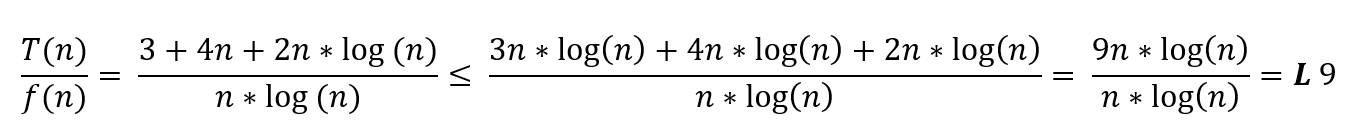
L = mergesort(D) (n log n)

return L (1)

T(n) = 1 + 1 + n \* [1 + 1+ 1+ 1+ log(n)] + n\*log(n) + 1

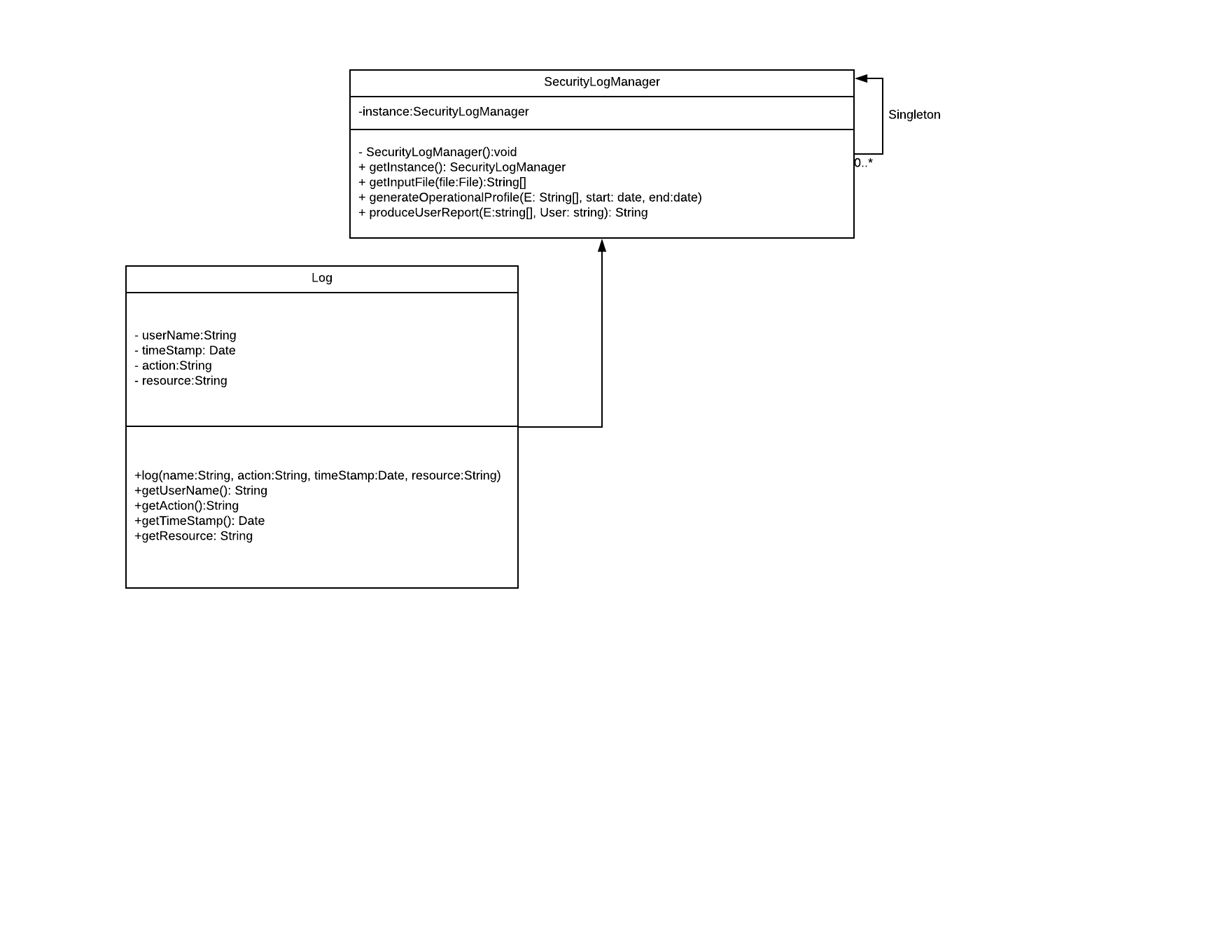
T(n) = 3 + 4n + 2n\*log(n)

Big oh = O(n log(n))



Since 9 is a constant, T(n) is O(f(n)).

**Software Design**

For this project, we chose a Singleton design pattern. We chose this design pattern because we wanted to limit the instantiation of the SecurityLogManager class to one object, since only one object will be needed to coordinate all the actions of this program. Additionally, we decided to create the Log class, which holds all the information a log entry will have, such as the username, time stamp, action, and resource. The Log class is associated and used by the SecurityLogManager, as shown by the association arrow. Furthermore, we decided that the algorithms should be methods under the SecurityLogManager, since those are the main functionalities of the program and will be used by the SecurityLogManager to perform the necessary functions of this software. Lastly, we decided we will not have a GUI for this project, so the software can be used with a command line interface.