# CS150 Project 2 Report

Oanh Doan

November 12, 2017

## 1 Introduction

The goal of this project is to create a program that can efficiently manage events on a college campus. An event contains information on name of the event, date and location at which it will take place, name of organization that is coordinating the event, and the type of event. The program provides a structure to organize a list of events that allows for efficient performance of the following actions [2]:

- Add an event
- Search for an event
- Remove an event
- Update an event
- Print out all events

More specifically, the user has the option to search for an event by one property or by a combination of properties. For example, the user can search for an event named "CS Social", but can also specify that "CS Social" has to take place in the week of October 30, 2017 and to be organized by Professor Liew. The program not only needs to deal with multiple search inputs, but also needs to do it in the fastest way.

The key here is to store all data in different containers. For each action, there will be one container that will minimize running time. Thus, based on the type of actions the user wants to perform, the program pulls out data from the appropriate containers.

Each container is built using a different data structure. In this project, I propose a data structure that I think is efficient for each container. The, I will run experiments to analyze complexity of the programs and make suggestions for improvements (if any).

## 2 Approach

### 2.1 Design

There are 10 main classes and 1 interface in this program:

- Event: stores all information about an event (name, dates, location, type, organizers).
- Node<E> (generic): stores an element of generic type and an arraylist of events
- DateNode (extends Node): the element is of type Date. An object of this class stores all events that occur on a particular date.
- EventByProperty (interface): states required methods (add, remove, find) for each container.

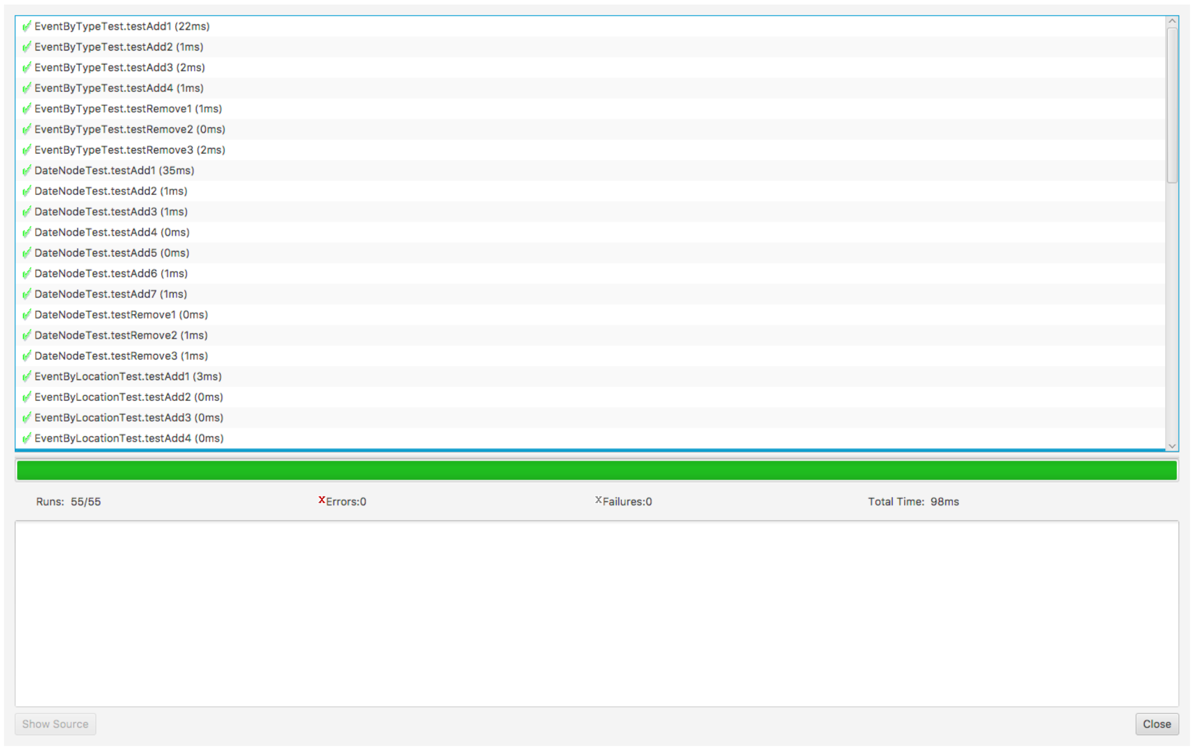
The following classes are event containers. This means that they store all events, but organize them in different ways.

- EventByName: Events are organized by names, in the form of a tree map, where the key is event name and the value is a list of events of that name. It's possible that 2 events have the same name but happen at different times or are organized by different organizations.
- EventByType: Events are organized by types. It stores a linked list, where each node is an object of the class Node<String>. The key in each node is event type. Since there are only a few event types, this list is easy to manage (i.e add/remoe/find by type).
- EventByOrganization: Similar to EventByName. This container organizes events in a tree map where the key is organization's name. Since an event can be co-organized by multiple organizers, it can belong to different "nodes" in the tree.
- EventByLocation: Similar to EventByOrganizer. The difference is that the key is location and each event can belong to only one "node".
- EventByDate: This class contains a sorted linked list of DateNode. Recall that each node represents a date. When an event is added, it is added to all nodes on which the events occur. Similar logic applies to deletion and search.

The last 3 classes read input file, simulate the program behavior, and conduct experiments.

- FileScanner: read a text file and create a database of events for search engine.
- SearchEngine: simulates program's behavior, prompts users to give inputs and perform some actions
- Experiment: measure run time of different methods using different containers to analyze complexity and address our original question.

Each class has a unit test class, except for interface, generic classes, experiment, search engine, and file scanner. Since search engine deals with user's input, it cannot be tested. But all smaller methods that are used within search engine are tested.



```
✓ EventByTypeTest.testAdd1 (22ms)
✓ EventByTypeTest.testAdd2 (1ms)
✓ EventByTypeTest.testAdd3 (2ms)
✓ EventByTypeTest.testAdd4 (1ms)
✓ EventByTypeTest.testRemove1 (1ms)
✓ EventByTypeTest.testRemove2 (0ms)
✓ EventByTypeTest.testRemove3 (2ms)
✓ DateNodeTest.testAdd1 (35ms)
✓ DateNodeTest.testAdd2 (1ms)
✓ DateNodeTest.testAdd3 (1ms)
✓ DateNodeTest.testAdd4 (0ms)
✓ DateNodeTest.testAdd5 (0ms)
✓ DateNodeTest.testAdd6 (1ms)
✓ DateNodeTest.testAdd7 (1ms)
✓ DateNodeTest.testRemove1 (0ms)
✓ DateNodeTest.testRemove2 (1ms)
✓ DateNodeTest.testRemove3 (1ms)
✓ EventByLocationTest.testAdd1 (3ms)
✓ EventByLocationTest.testAdd2 (0ms)
✓ EventByLocationTest.testAdd3 (0ms)
✓ EventByLocationTest.testAdd4 (0ms)

Runs: 55/55        Errors:0        Failures:0        Total Time: 98ms
```

Figure 1. Results of unit tests

## 2.2 Choice of algorithm and data structures

- TreeMap [10]: We use tree map organize events in EventByName, EventByOrganization, EventByLocation where key words are some strings as names, locations, or organizers. Since String is comparable, TreeMap makes it convenient to search for an event using key words ($O(\log n)$). The complexity of insertion is $O(\log n)$ for all these containers. The complexity of deletion is $O(\log n)$ for EventByName, but is more complicated for the other two. Since we can only delete an event given event name, the program has to search through all nodes in the tree to find events of that name and delete it. So the complexity is at least $O(n)$.

- ArrayList [11]: We choose to use ArrayList to store events in each node of the containers because ArrayList is easy to manage. Adding an event to list end takes $O(1)$ while searching and deleting take at most $O(n)$ where $n$ is the number of events in a node, not the total number of events in the program.

- LinkedList [6]: EventByType and EventByDate use linked lists. This makes sense for EventByType because we have only a few types, so the number of nodes in the list is small, which makes it fast to add/delete/search for a node. I first chose this data structure for EventByDate because it well represents a calendar or a chain of days, which makes it easier to think through implementation. However, as I conduct experiments, I realize that this may not be a good choice, especially if we have a big data set of events in several years. This would require the list to have hundreds of nodes. At this point, even though complexity is still $O(n)$, search, deletion, and insertion are not time efficient since $n$ is big.

- Interface [5]: Creates a contract for all containers
- Comparable interface [4]: Event class implements Comparable interface to make events comparable. In addtion, we also override the method equals() [7] to make it possible to compare two different objects with same data. This method will be implicitly used by contains() method in all data structures.
- Scanner [9]: Reads input file as well as users' input
- PrintWriter [8]: writes data to an output file when specified by the user
- Random Class [3] [1]: is used to generate random numbers in creating random names, organizations, locations, ... of events.

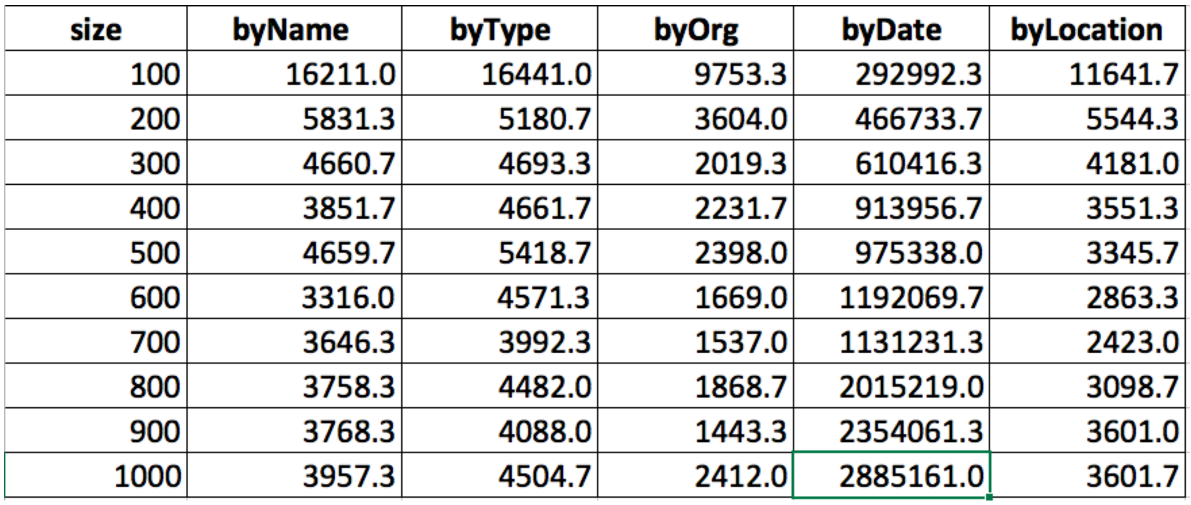
## 3 Methods

We first analyze the complexity of finding items by each search category and writing records to file using EventByName.

- We use 10 files of various input sizes, from 100 and 1000. All of these files contain randomly generated events. This is done by generating random numbers and concatenating the number with appropriate strings, such as *name_1234*, *org_321893*. The dates are also randomly generated over a period of time. Furthermore, the order of these inputs are shuffled when writing to files.
- For each input size and each search category, we try 3 random seeds and record the average run time.
- To measure time of writing records, we use the write() method in EventByName. We also try various input sizes, each with 3 different seeds.

## 4 Data and analysis

The following table presents search time by each category.



| size | byName | byType | byOrg | byDate | byLocation |
|---|---|---|---|---|---|
| 100 | 16211.0 | 16441.0 | 9753.3 | 292992.3 | 11641.7 |
| 200 | 5831.3 | 5180.7 | 3604.0 | 466733.7 | 5544.3 |
| 300 | 4660.7 | 4693.3 | 2019.3 | 610416.3 | 4181.0 |
| 400 | 3851.7 | 4661.7 | 2231.7 | 913956.7 | 3551.3 |
| 500 | 4659.7 | 5418.7 | 2398.0 | 975338.0 | 3345.7 |
| 600 | 3316.0 | 4571.3 | 1669.0 | 1192069.7 | 2863.3 |
| 700 | 3646.3 | 3992.3 | 1537.0 | 1131231.3 | 2423.0 |
| 800 | 3758.3 | 4482.0 | 1868.7 | 2015219.0 | 3098.7 |
| 900 | 3768.3 | 4088.0 | 1443.3 | 2354061.3 | 3601.0 |
| 1000 | 3957.3 | 4504.7 | 2412.0 | 2885161.0 | 3601.7 |

Figure 1. Time search by each category (nanoseconds)

From this table, we can see that the search time when input size is 100 is significantly greater than the rest. This might have happened because as the system needs to set up something before it actually starts the search. Thus, we will exclude the search time of input size 100 from our analysis. The following graphs show search time of each category as input size increases.

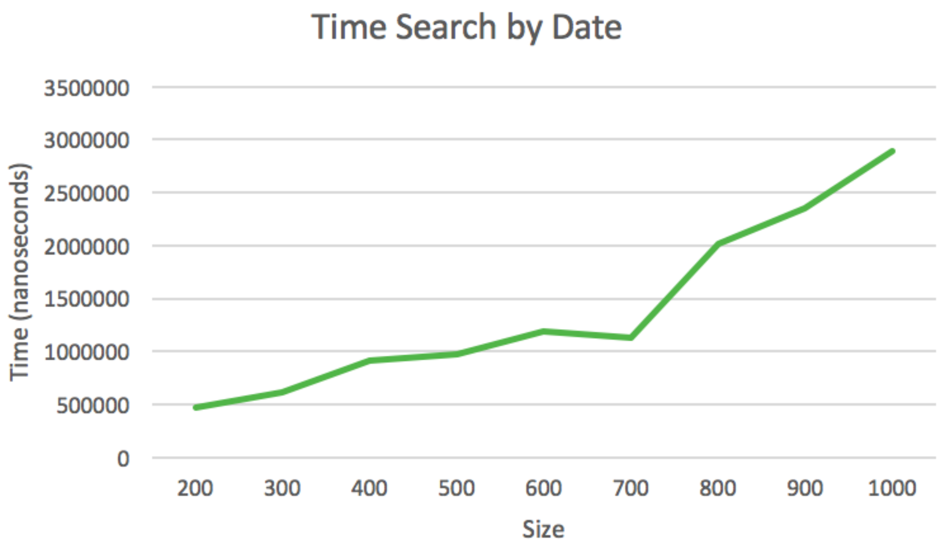


Figure 2. Time search by Date (nanoseconds)

First of all, search by date uses the container EventByDate. Since EventByDate is a sorted linked list of dates, searching for a date in this container takes $O(n)$. The fact that the linked list is sorted does not provide much help. A sorted array list of dates can probably be more efficient in this case because the complexity can be reduced to $O(\log n)$ by using binary search.

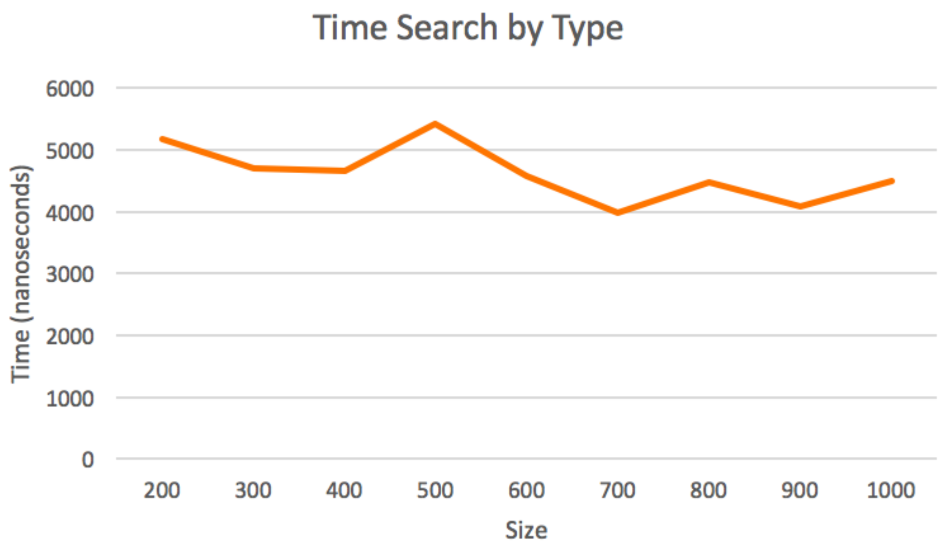


Figure 3. Time search by Type (nanoseconds)

Second, search by type uses the EventByType container. This is also a linked list, but we do not see a linear trend as in search by date. The reason is that the size of this list is very small; the list is restricted to only 5 types of events, including music, art, athletics, talk, and academic. As a result, search time using this linked list is almost constant.

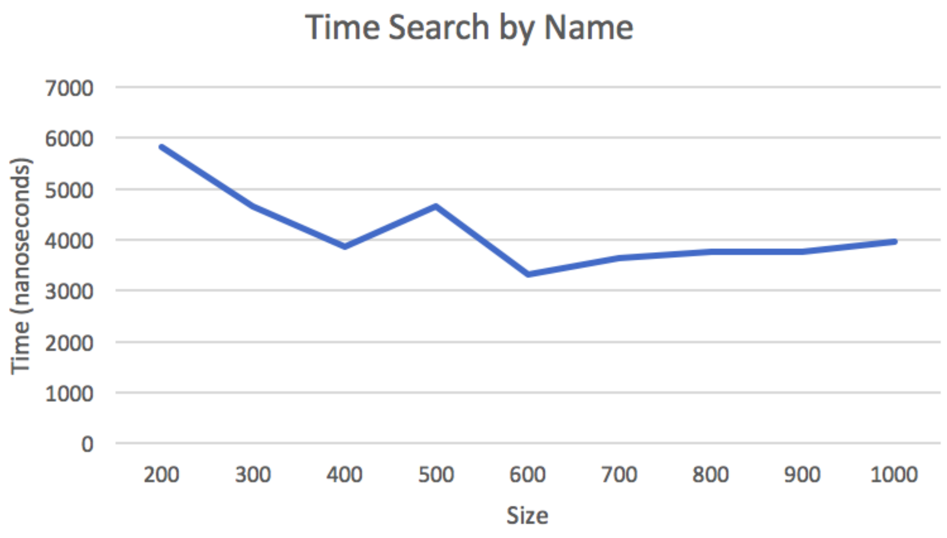


Figure 4. Time search by Name (nanoseconds)

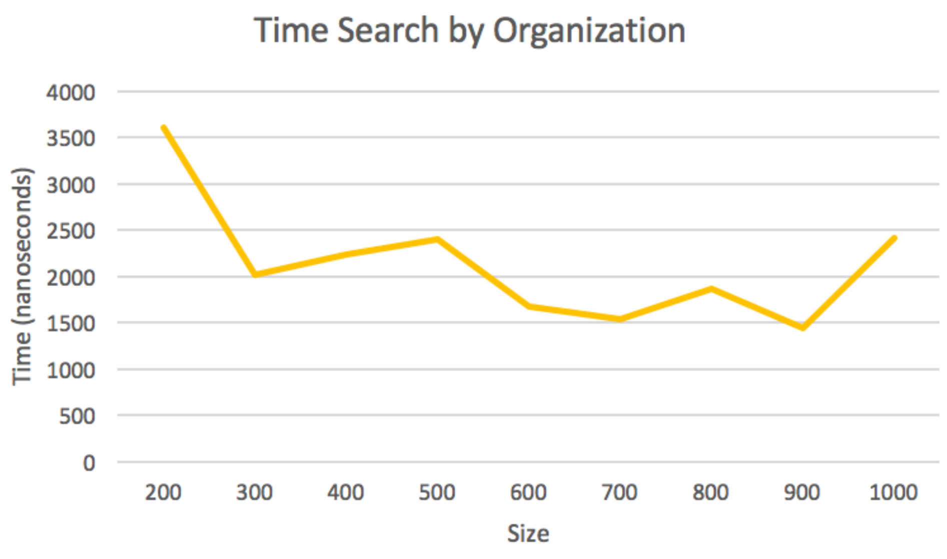


Figure 5. Time search by Org (nanoseconds)

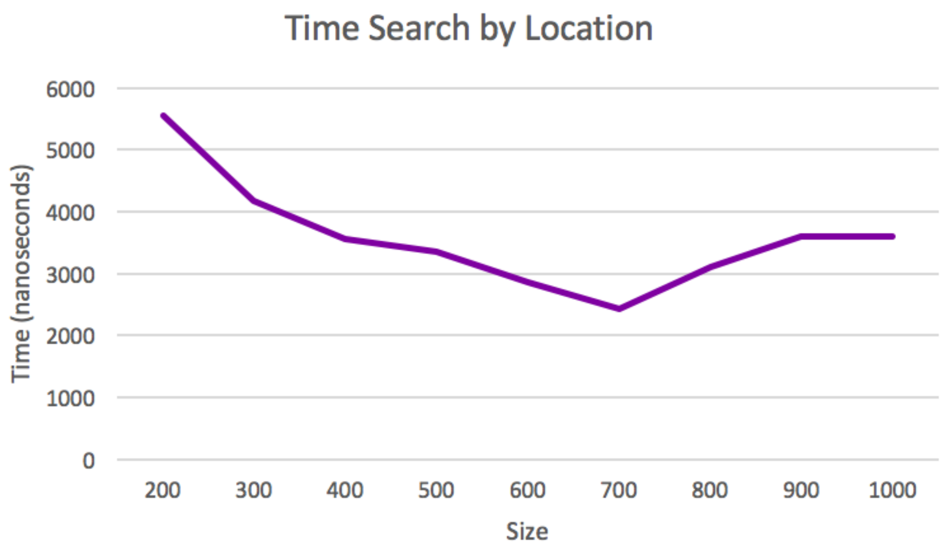
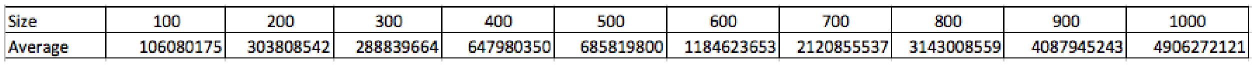


Figure 6. Time search by Loc (nanoseconds)

Next, since all other containers are trees, we expect complexity to be $O(\log n)$. However, the figures fail to deliver such information. In fact, we observe light fluctuations in all graphs. One possible explanation is that the input sizes are not big enough to observe the trend.

In addition, we look at the writing of all records. Table 2 presents the writing time of different input sizes.



| Size | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| Average | 106080175 | 303808542 | 288839664 | 647980350 | 685819800 | 1184623653 | 2120855537 | 3143008559 | 4087945243 | 4906272121 |

Table 2. Writing time of all records (nanoseconds)

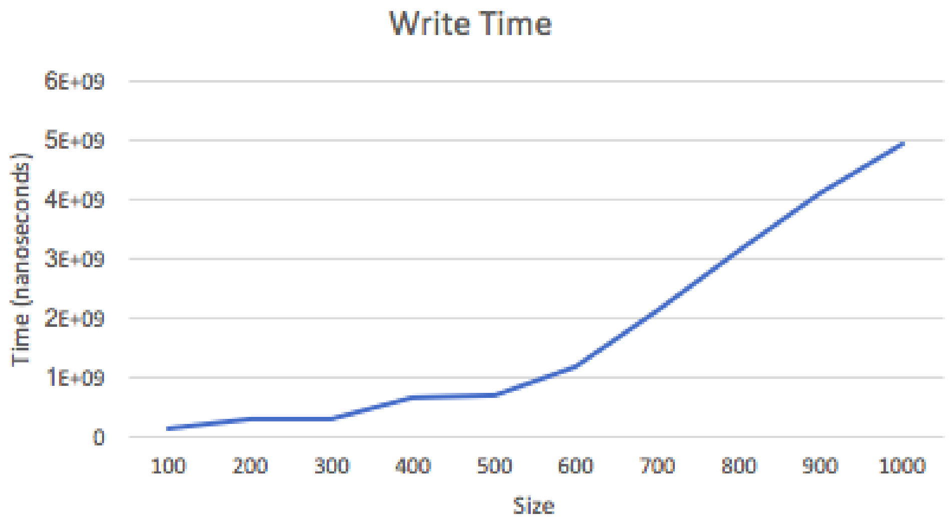


Figure 7. Time search by Loc (nanoseconds)

When writing the records to a file, we go through each event once. Thus, we expect complexity to be $O(n)$. As expected, the graph exhibits a linear, increasing trend of writing time as the number of records goes up.

## 5 Conclusion

- Depending on our purpose, an amount of data can be organized differently to make it most efficient.
- Sometimes the most obvious solution is not the best one. I first came up with the idea of using sorted Linked List as a container to organize events by dates. At the first glance, the idea makes perfect intuitive sense, but as the amount of data grows, it becomes very inefficient and time-consuming to perform any action.

## References

[1] javapractices. Generate random numbers. http://www.javapractices.com/topic/TopicAction.do?Id=62.

[2] Chun Wai Liew. Project description. Lafayette College.

[3] Oracble. Class random. https://docs.oracle.com/javase/7/docs/api/java/util/Random.html.

[4] Oracble. Comparable interface. https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html.

[5] Oracble. Interface. https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html.

[6] Oracble. Linkedlist. https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html.

[7] Oracble. Overriding equals() method in java. https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html.

[8] Oracble. Printwriter. https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html.

[9] Oracble. Scanner. https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html.

[10] Oracble. Treemap. https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html.

[11] Oracle. Arraylist (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.