Criterion C: Development

Introduction

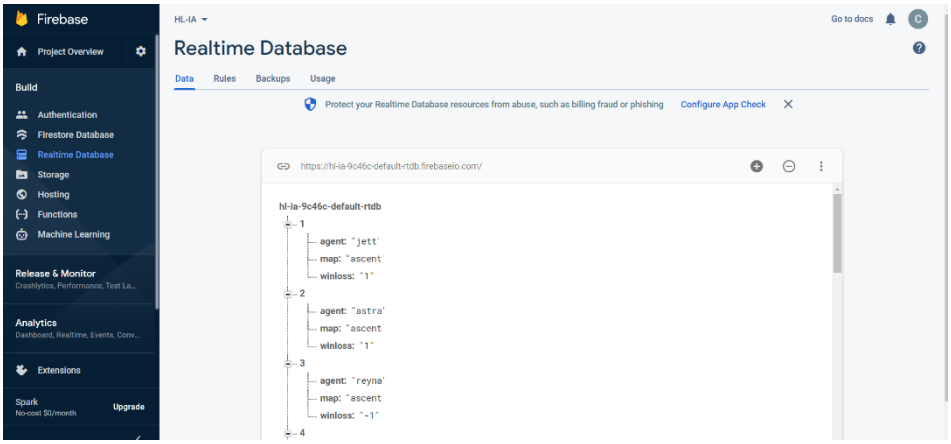
I used Android Studio and its built-in IntelliJ interface to develop my Java program and used Android Studio's tools for a Graphical User Interface. The program allows my client to **access and edit his Valorant match history, sort his match history, and receive advice for specific maps.**

Techniques

1. Google Firebase
2. Storing Valorant Matches
3. Sorting Match History
4. Calculating Advice

Google Firebase:

Implementing a hierarchical composite data structure:



I used Google Firebase to store all my data. It stored all the Valorant matches of my client's history. This is a **hierarchical composite data structure**, as the objects each have variables attached to them containing an ID for the object, the agent used, the map it was played on, and whether it was a win or a loss. This is useful because I could access all of my client's match information together by saving (See "FireBase Documentation" in Appendix A for more info on Google Firebase).

Connecting Google FireBase to Java:

I use a private `vDatabase = FirebaseDatabase.getInstance().getReference();` DatabaseReference object "vDatabase" to **connect to the FireBase**. I establish the connection each time my app needs to be refreshed. This is useful because it helps me connect to FireBase from any method within each class (See "FireBase Documentation" in Appendix A for more info on Google Firebase).

Adding Data to FireBase:

After receiving match data from the user, the app **adds the match information to the database** (See “Read and Write Data” in Appendix). This is useful for saving the information for my client.

```
vDatabase.child(Integer.toString( databaseSize+1)).child("winloss").setValue(match.getWinloss());
vDatabase.child(Integer.toString( databaseSize+1)).child("map").setValue(match.getMap());
vDatabase.child(Integer.toString( databaseSize+1)).child("agent").setValue(match.getAgent());
vDatabase.child("size").setValue(Integer.toString( databaseSize+1));
```

Storing Valorant Matches

Match Class:

The “Match” class is the type of object that is being stored and worked with in the app. This is useful to my client because his gameplay is based on his match history and this class allows the app to organize his data using class variables defining the result of the match, the map it was on, and what agent was used. I use **polymorphism** in this class to allow Match objects to be created with the three class variables set to null or for them to be set as the object is initialized (See “polymorphism in java” in appendix for further explanation). This is useful for my code because some places in my code, I have access to all three variables already and want to initialize it at once, but sometimes I could not.

```
public Match(){}
public Match(String w, String m, String a){
    winloss = w;
    map = m;
    agent = a;
}
```

This leads me to my next technique of using **encapsulation** with getter and setter methods (See “encapsulation” in appendix for further explanation). These methods were useful because they allowed me to change the class variables after the object has been created.

```
public String toString() { return winloss; }

public String getWinloss() { return winloss; }

public String getMap() { return map; }

public String getAgent() { return agent; }

public void setWinloss(String winloss) { this.winloss = winloss; }

public void setMap(String map) { this.map = map; }

public void setAgent(String agent) { this.agent = agent; }
```

Using Linked Lists to Store Matches

I implemented a **Linked List ADT** to store the matches by using a List class and a Node class (See "Linked List" in Appendix for further explanation). This was useful in storing my matches in a fast, accessible way.

```
public class LinkedMatchList {
    private MatchListNode front;
    public LinkedMatchList() { front = null; }
    //adds the given value to the end of the list
    public void add(Match value){...}
    public String toString(){...}


    public Match get(int index){...}
    public void remove(int index) {...}
    public int size(){...}
}
```

```
public class MatchListNode {
    Match data;
    MatchListNode next;
    public MatchListNode(){}
    public MatchListNode(Match data){...}
    public MatchListNode(Match data, MatchListNode next){...}
    public int getLength(){...}
}
```

Populating Linked List by Reading Database Info Using Recursion:

To make the code easier to work with, I populate a "LinkedMatchList" with data from FireBase. To do this, I use a **recursive method** that calls itself, while incrementing its integer "ID" higher until it reaches the size of the database. In this method, I **read data from FireBase** by navigating using ".child()" with match ID's and the variable information (*See "Retrieve Data" in Appendix*).

```
public void fillMatchList(){
    // Update the Match Linked List so it can be used in the app!
    recursionMatchesFromDatabase( ID: 1);
}
public void recursionMatchesFromDatabase(int ID){
    Match match = new Match();
    vDatabase.child(Integer.toString(ID)).child("winloss").get().addOnCompleteListener(new OnCompleteListener<DataSnapshot>() {

        public void onComplete(@NonNull Task<DataSnapshot> task) {
            match.setWinloss(String.valueOf(task.getResult().getValue()));
        }
    });
    vDatabase.child(Integer.toString(ID)).child("map").get().addOnCompleteListener(new OnCompleteListener<DataSnapshot>() {

        public void onComplete(@NonNull Task<DataSnapshot> task) {
            match.setMap(String.valueOf(task.getResult().getValue()));
        }
    });
    vDatabase.child(Integer.toString(ID)).child("agent").get().addOnCompleteListener(new OnCompleteListener<DataSnapshot>() {

        public void onComplete(@NonNull Task<DataSnapshot> task) {
            match.setAgent(String.valueOf(task.getResult().getValue()));
        }
    });

    linkedMatchList.add(match);
    if(ID<databaseSize){
        recursionMatchesFromDatabase( ID: ID+1);
    }
}
```

Sorting Match History

Reversing the List

In the **additional libraries** to Java I use, I utilize android studio's GUI by using a switch here to control the sorting of the match history. I use a **stack** to reverse my linked list of Match objects. First, I put all the linked list objects into a stack, empty the list, then put the stack objects into the list. This is clever because the stack utilizes its "last in, first out" quality, allowing the stack to naturally reverse the list by filling it up and emptying it out immediately. This helps my client because it allows him to sort his match history to his liking.


```
//Sorting the list
Switch switch1 = findViewById(R.id.switch1);
switch1.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton compoundButton, boolean b) {
        //Reverse the order using a stack (last in first out)
        Stack<Match> stack = new Stack<>();
        for(int i =0; i<linkedMatchList.size();i++){
            stack.push(linkedMatchList.get(i));
        }
        while(!(linkedMatchList.size()==0)){
            linkedMatchList.remove( index: 0);
        }
        int stacksize= stack.size();
        for(int i =0; i<stacksize; i++){
            linkedMatchList.add(stack.pop());
        }
```

Calculating Advice:

Searching for Specific Map:

When calculating the advice, it is needed to discern only the matches that are in the certain map that the user is looking for. To do this, it sifts through each match and tests if the match's map is equal to the one the user is looking for. This **search** is useful because my client requested advice for each specific map, so I had to calculate statistics with boundaries set by the user.

```
String map = t.getText().toString();
HashMap<String, ArrayList<Integer>> matchesWithMap = new HashMap<>();
//this hashmap stores the w/l ratio of agents playing on this specific map
for(int i = 0; i<linkedMatchList.size(); i++){
    if(linkedMatchList.get(i).getMap().equals(map)){
        if(matchesWithMap.containsKey(linkedMatchList.get(i).getAgent())){
            if(linkedMatchList.get(i).getWinloss().equals("1")){
                int temp = matchesWithMap.get(linkedMatchList.get(i).getAgent()).get(0);// get amount of wins
                matchesWithMap.get(linkedMatchList.get(i).getAgent()).set(0, temp+1);//change it to one more
            }else{
                int temp = matchesWithMap.get(linkedMatchList.get(i).getAgent()).get(1);//get amount of losses
                matchesWithMap.get(linkedMatchList.get(i).getAgent()).set(1, temp+1);//change it to one more
            }
        }else{
            ArrayList<Integer> temp = new ArrayList<>();
            if(linkedMatchList.get(i).getWinloss().equals("-1")){
                temp.add(0); //wins
                temp.add(1); //losses
            }else{
                temp.add(1);
                temp.add(0);
            }
            matchesWithMap.put(linkedMatchList.get(i).getAgent(), temp);
        }
    }
}
```

Display Map Description

In addition to agent selection advice, this page displays a description of the map being played on. This is done by using a **2D Linked List** (See "Multidimensional Collections" in Appendix for further explanation of usage). I created 4 classes for this. A "Linked List Linked List," "Linked List List Node," "Linked String List," and "String List Node" for this to work. For each List object in the parent List, there are two strings: one for the map name and one for the map

```
//Set map description
TextView mapdescription = findViewById(R.id.textViewMapDescription);
for(int i =0; i<linkedListList.size(); i++){
    for(int y=0; y<linkedListList.get(i).size(); y++){
        if(linkedListList.get(i).get(0).equals(map)){
            mapdescription.setText(linkedListList.get(i).get(1));
        }
    }
}
```

description. This is used by searching the list for the object with the map name key and then displaying that corresponding description. It was my clever implementation of what can be similarly done with a HashMap.

Word Count: 804 words