The airport connection check is a high-level function exposed by the API of a travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary. For example, a traveler may intend to fly from Gothenburg to Los Angeles, but there is a connection through Frankfurt. Therefore, their itinerary is Gothenburg -> Frankfurt (Flight A) and Frankfurt -> Los Angeles (Flight B).

This service will ensure that the connection through Frankfurt is a valid one. For example, if the arrival airport of Flight A differs from the departure airport of Flight B, the connection is invalid. That is, if we pass in two flights, and Flight A arrives in Frankfurt, but Flight B departs from Munich, it is not a valid connection.

Likewise, if the departure time of Flight B is too close to the arrival time of Flight A, the connection is invalid. If Flight A arrives in Frankfurt at 8:00, and Flight B departs at 8:05, there is not sufficient time to complete the customs process and board the flight.

```
validConnection(Flight flightA, Flight flightB)
                returns ValidityCode
```

A **`Flight`** is a data structure consisting of:

- A unique identifying flight code (string, three characters followed by four numbers).
- The originating airport code (three character string).
- The scheduled departure time from the originating airport (in universal time).
- The destination airport code (three character string).
- The scheduled arrival time at the destination airport (in universal time).

There is also a **flight database**, where each record contains:

- Three-letter airport code (three character string).
- Airport country (two character string).
- Minimum connection time (integer, minimum number of minutes that must be allowed for flight connections to be valid).

`ValidityCode` is an integer with value 0 for OK, 1 for invalid airport code, 2 for a connection that is too short, 3 for flights that do not connect (Flight A does not land in the same location that Flight B departs from), or 4 for any other errors (malformed input or any other unexpected errors).

Design system test cases using the category-partition method for the `validConnection` function.

1. Identify choices (aspects that you control and that can vary the outcome) for the two input flights and the database.
2. For each choice, identify a set of representative values.
3. Apply ERROR, SINGLE, and IF constraints.
    a. ERROR = This representative value will trigger an error no matter that it is paired with.
    b. SINGLE = This representative value should give an OK response, but we want to make sure we try it once.
    c. IF = This representative value can only be used if a certain value is set for another choice.

Sample Solution

Recall the lectures on system testing.

The approximate process of writing system tests is the following:

1. **For each high-level independently testable feature surfaced by an interface, you need to identify the parameters. These can be explicit (passed into the function) or implicit (configuration options or other environmental factors - such as databases - that influence the outcome of the function).**
2. **Each parameter can be manipulated in many ways through testing. For each parameter, you must identify choices - aspects of that input that you can vary, and that will have some impact on the outcome of testing this function. For example, if an input is a data structure, the choices might include fields of that data structure that impact the outcome of the function. If that data structure is serialized from a file, then choices may include the status of the file (whether that file exists, is corrupted, and so on).**
3. **You cannot exhaustively test a function, there are too many possible values that can be fed in. So, instead, you partition the input domain for each choice into representative values (types of input). If you try at least one concrete input from each of these value types, you should trigger different outcomes and be more likely to notice faults. We discussed some methods of performing this partitioning in class.**
4. **Once representative values are chosen, you can form test specifications - abstract recipes for tests - by choosing one value for each choice. You can transform these recipes into actual test cases by coming up with concrete input values that fit into the category of value for each choice.**

In this exercise, you have been asked to perform Steps 1-3 above - identify the parameters, split the parameters into choices, then partition the input values for each choice into representative values. You can constrain the number of test specifications by adding further constraints (ERROR, SINGLE, IF) that note when certain representative values should be used in combination with other values. Note that you do not have to use all constraints (i.e., you do not need to use SINGLE unless it makes sense to do so).

This function has two explicit inputs - the two flights - and an implicit input - an airport database. A flight is a complex data structure containing several fields, each of those fields represents a controllable input category. Your choices should revolve around those fields.

Remember that the function's parameters may influence each other (testing this function requires considering both Flight A and B's field values as well as what is in the database), so the representative values must reflect how different choices or variables can interact. IF-constraints are also a good way to indicate when representative values for two choices should be paired.

Parameter: FlightA

Choice: Flight code:

- ***Malformed (string that does not follow stated formatting convention) [error]***
- ***not in database [error] (Note: I assumed here that there was a flight database too. State any assumptions you make in your answer.)***
- ***valid***

Choice: Originating airport code:

- ***malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***valid city***

Choice: Scheduled departure time:

- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***legal***

Choice: Destination airport (transfer airport - where connection takes place):

- ***malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***valid city***

Choice: Scheduled arrival time (tA):

- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***legal***

Parameter: FlightB

Choice: Flight code:

- ***Malformed (string that does not follow stated formatting convention) [error]***
- ***not in database [error]***
- ***valid***

Choice: Originating airport code:

- ***Malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***differs from transfer airport [error]***
- ***same as transfer airport***

Choice: Scheduled departure time:

- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***before arriving flight time (tA) [error]***
- ***between tA and tA + minimum connection time (CT) [error]***
- ***equal to tA + CT [single]***
- ***greater than tA + CT***

Choice: Destination airport code:

- ***malformed (not a three-letter string) [error]***
- ***not in database [error]***
- ***valid city***

Choice: Scheduled arrival time:

- ***malformed (not following formatting convention) [error]***
- ***out of legal range (not a valid time) [error]***
- ***legal***

<u>Parameter: Database record</u>

This parameter refers to the database record corresponding to the transfer airport.

Choice: Airport code:

- ***Malformed (not a three-letter string) [error]***
- ***blank [error]***
- ***valid***

Choice: Airport country:

- ***Malformed (non-string) [error]***
- ***blank [error]***
- ***Invalid (not a country) [error]***
- ***valid***

Choice: Minimum connection time:

- ***malformed (not following formatting convention) [error]***
- ***blank [error]***
- ***> 0 [error]***
- ***0 [single]***
- ***valid***