Homework-2

Part (a) - Storing Bounded sensor readings :

Data Structure:

- For this scenario of bounded sensor readings , a Hash Map(Dictionary) can be used for constant time access to operate on the sensor readings. Here, the Dictionary(Hashmap in python) will map each sensor ID *s* to its most recent reading and thus this provides us O(1) constant access time for our operations.
- Since this is bounded, a list data structure can be used to manage the history of readings with recent reading at the last index. This helps us ensure efficient deletions and updates in O(1).

Functions and Pseudocode:

```
dictionary = {} #Initialise dictionary to store most recent reading of each sensor
list = [] #to store history of all readings in an order
```

1. Insert a new reading:

   ```
   function insertReading(sensor_ID, reading):
       dictionary[sensor_ID] = reading
       list.append((sensor_ID, reading))
   ```

   This function updates the dict with the new reading and appends it to the list to maintain history.

2. Search for an arbitrary reading:

   ```
   function searchReading(sensor_ID):
       if sensor_ID exists in dictionary:
           return dictionary[sensor_ID]
     else:
           return “Reading has not been found”
   ```

This checks for sensor_ID in dict before retrieving it.

3. Delete the most recent reading:

```
function deleteMostRecentReading():
  if list is not empty:
      latest_reading = list.pop()
      sensor_ID, reading = latest_reading

      if sensor_ID in dictionary and dictionary[sensor_ID] == reading:
          del dictionary[sensor_ID]
  else:
    return “No recent readings to delete”
```

   This function pops last item in list for recent reading after checking its existence and checks if that sensor_ID exists in dictionary and stored reading in dictionary matches with the one that gets deleted so that if reading in dictionary has already been updated, this deletion could be avoided.

4. Search for the most recent reading:

```
function searchMostRecentReading():
   if list is not empty:
     return list[-1];
   else:
     return “No available readings in the list”
```

   This function fetches the last item in the list which is our most recent.

Time complexity:

1. Insert a new reading:
   - Adding or lookup in a dictionary takes O(1) time since dictionaries offer constant time access for insertion.
   - Appending to a list also takes O(1) time.
   - Overall Time complexity - O(1)
2. Search for an arbitrary reading:
   - Lookup for sensor_ID and retrieving both takes O(1) time.
   - Overall Time complexity - O(1)

3. Delete the most recent reading:
    - Pop operation takes O(1) time.
    - Lookup for sensor_ID and match verification is also O(1).
    - Deleting a key in a dictionary also takes O(1).
    - Overall Time complexity - O(1).
4. Search for the most recent reading:
    - Access of the last item in a list using list[-1] takes O(1) time.
    - Overall Time complexity - O(1).

Part (b) - Storing unbounded sensor readings :

Data Structure:

- In unbounded sensor readings, a Balanced Binary Search Tree can be used like an AVL Tree. Because the data structure in this scenario should handle dynamic data efficiently,
- A balanced BST here will store readings for each sensor in a sorted order and this makes our operations efficient.
- A dictionary of BSTs needs to be maintained where each sensor ID maps to its own BST of readings.
- The balanced BST allows efficient O(logn) operations due to its **self-balancing nature**, making it suitable for dynamic data insertion, deletion, and retrieval in **logarithmic time**.

Functions and Pseudocode:

dictionary = {} #Initialise dictionary where each key is a sensor_ID and each value is a BST instance.
BST class initialization to manage sensor's readings and our operations are performed within each sensor's BST.

1. Insert a new reading:

```
function insertReading(sensor_ID, reading):
    If sensor_ID not in dictionary:
        dictionary[sensor_ID] = BST()
    dictionary[sensor_ID].insert(reading)
```

This function if sensor_ID does not exist in dict, then creates a new BST for this sensor, then inserts a new reading value in BST for that particular sensor.

2. Search for an arbitrary reading:

```
function searchReading(sensor_ID, reading):
    if sensor_ID exists in dictionary:
        return dictionary[sensor_ID].search(reading)
   else:
        return “Sensor_ID has not been found”
```

This checks for sensor_ID in dict before retrieving it and searches BST.

3. Delete the most recent reading:

```
function deleteMostRecentReading(sensor_ID):
   if sensor_ID exists in dictionary:
        latest_reading = dictionary[sensor_ID].get_max()

        if latest_reading is not None:
            dictionary[sensor_ID].delete(latest_reading)
   else:
      return “No recent readings to delete”
```

This function calls get_max and gets the highest value in BST assuming it is the most recent one and deletes it from BST.

4. Search for the most recent reading:

```
function searchMostRecentReading(sensor_ID):
    if sensor_ID exists in dictionary:
       return dictionary[sensor_ID].get_max();
    else:
       return “sensor_ID not found”
```

This function fetches the max value for a sensor as its the most recent.

Time complexity:

1. Insert a new reading:
    - Checking for sensor_Id and adding a BST instance takes O(1) time.
    - Inserting a reading into any balanced BST has a time complexity of O(log n) where n is the number of readings of that particular sensor's BST.
    - Overall Time complexity - O(log n)
2. Search for an arbitrary reading:
    - Lookup for sensor_ID takes O(1) time.
    - Searching in a BST takes O(log n) time where n is the number of readings of that particular sensor's BST.
    - Overall Time complexity - O(log n)
3. Delete the most recent reading:
    - Lookup for sensor_ID takes O(1) time.
    - Retrieval of max value in a BST (assuming most recent) takes O(log n).
    - Deleting a node in any balanced BST takes O(log n).
    - Overall Time complexity - O(log n).
4. Search for the most recent reading:
    - Lookup for sensor_ID takes O(1) time.
    - Retrieval of max value in a BST (assuming most recent) takes O(log n).
    - Overall Time complexity - O(log n).