Homework-2

Part (a): Storing Bounded Sensor Readings:

Data Structure:

Since the sensor IDs and readings are within fixed, small ranges, A Hash Map (Dictionary) for quick access to the latest reading of each sensor and a list to maintain a history of readings. This allows for efficient handling and constant time operations.

- **Dictionary (**latest_readings**)**: Maps each sensor ID to its most recent reading. This allows O(1) access for updating and retrieving the latest reading.
- **List (**readings_history**)**: Stores all readings in insertion order to help track the history and provide easy deletion of the most recent reading. The list is a chronological log of readings that lets us access or remove the latest reading efficiently but also maintains a complete history of past readings.

Functions:

1. Inserting a new reading:

   ```
   def insert_reading(sensor_id, reading):

       latest_readings[sensor_id] = reading

       reading_history.append((sensor_id, reading))
   ```

   Time Complexity:

   - Updating the dictionary takes O(1) time.
   - Appending reading to a list of readings takes O(1) time.
   - Overall Time complexity: O(1)

2. Search for an arbitrary reading:

   ```
   def search_reading(sensor_id):

       if sensor_id exists in latest_readings:
   ```

```
        return latest_readings[sensor_id]
    else:
        return “sensor_id not found in latest readings”
```

Time complexity:

- Sensor_id lookup in the dictionary and retrieval both take O(1) time.
- Overall Time complexity: O(1)

3. Deleting the most recent reading:

```
def delete_most_recent():
    if reading_history:
        sensor_id, reading = reading_history.pop()
        # Only delete if this is the current most recent reading
        if latest_readings.get(sensor_id)==reading:
            del latest_readings[sensor_id]
    else:
        return “There are no recent readings to delete”
```

Time complexity:

- Removing the last item from the history list takes O(1) time and also sensor_id lookup and matching with the latest reading also takes O(1) time.
- Overall Time complexity: O(1)

4. Searching for the most recent reading:

```
def search_most_recent():
    if reading_history:
```

```
        return reading_history[-1]
    else:
        return "readings are not available"
```

Time complexity:

- Fetching the last reading in the list which is represented as the most recent reading takes O(1) time.
- Overall Time complexity: O(1)

Part (b): Storing Unbounded Sensor Readings:

Data Structure:

For unbounded readings, we need a data structure that handles an expanding dataset efficiently. A balanced binary search tree (BST) is ideal here as it provides logarithmic time complexity for insertions, deletions, and searches like AVL Tree or Red Black Tree.

- **Dictionary (**sensor_bsts**)**: Each sensor ID maps to its own BST, where each reading of that sensor is stored.
- **Balanced BST (**BST**)**: Each sensor's BST organises readings in a sorted order, allowing efficient insertion, search, and deletion.
- **Using timestamps in BST:** Each node in BST will store not only reading but also timestamp representing order of insertion. To retrieve the most recent reading , timestamp can be used to find the largest timestamp(most recent) node.

Functions:

1. Inserting a new reading:

```
current_timestamp = 0

def insert_reading(sensor_id, reading):
    global current_timestamp
    current_timestamp += 1
```

```
    new_node = BSTNode(reading, current_timestamp)

    if sensor_id not in sensor_bsts:

        sensor_bsts[sensor_id] = BST()

    sensor_bsts[sensor_id].insert(new_node)
```

Time Complexity:

- BST insertion takes O(log n) time whereas node creation takes O(1) time.
- sensor_id lookup takes O(1) time.
- Overall Time complexity: O(log n)

2. Search for an arbitrary reading:

```
def search_reading(sensor_id, reading):

    if sensor_id exists in sensor_bsts:

        return sensor_bsts[sensor_id].search(reading)

    else:

        return "sensor_id not found in sensor bsts"
```

Time complexity:

- Sensor_id lookup in the sensor_bsts and retrieval both takes O(1) time.
- Search for a particular reading within that BST takes O(log n) time
- Overall Time complexity: O(log n) assuming BST is balanced.

3. Deleting the most recent reading:

```
def delete_most_recent(sensor_id):

    if sensor_id exists in sensor_bsts:

        bst = sensor_bsts[sensor_id]
```

bst.delete_max() #assuming delete_max() removes node with highest timestamp

else:

return “There are no recent readings to delete”

Time complexity:

- Sensor_id lookup in the sensor_bsts and retrieval both takes O(1) time.
- Deletion in a balanced BST takes O(log n) time.
- Overall Time complexity: O(log n)

4. Searching for the most recent reading:

def search_most_recent(sensor_id):

if sensor_id exists in sensor_bsts:

bst = sensor_bsts[sensor_id]

return bst.get_max() #assuming get_max() fetches max node with timestamp for recent reading

else:

return “sensor not found”

Time complexity:

- Fetching the max node with the highest timestamp takes O(log n) time.
- sensor_id lookup in dictionary and retrieval takes O(1) time.
- Overall Time complexity: O(log n)