# **Homework-2 Solution**

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 most recent reading of each sensor and a list to maintain a history of readings. This allows for efficient handling and constant time operations.

* **Dictionary (**recent\_readings**)**: Maps each sensor ID to its most recent reading. This allows O(1) access for updating and retrieving the most recent 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 most recent reading efficiently but also maintains a complete history of past readings.

Functions:

1. Inserting a new reading:

def insert\_reading(sensor\_id, reading):

recent\_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)

1. Search for an arbitrary reading:

def search\_reading(sensor\_id):

if sensor\_id exists in recent\_readings:

return recent\_readings[sensor\_id]

else:

return “sensor\_id not found in recent readings”

Time complexity:

* Sensor\_id lookup in the dictionary and retrieval both take O(1) time.
* Overall Time complexity: O(1)

1. 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 recent\_readings.get(sensor\_id)==reading:

del recent\_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 most recent reading also takes O(1) time.
* Overall Time complexity: O(1)

1. 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 most recent 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)

1. 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.

1. 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)

1. 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)