Solution: Time-Based Key-Value Store

Let's solve the Time-Based Key-Value Store problem using Custom Data Structures.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized approach using binary search
    - Solution summary
    - Time complexity
    - Space complexity

Statement

Implement a data structure that can store multiple values of the same key at different timestamps and retrieve the key's value at a certain timestamp.

You'll need to implement the **TimeStamp** class. This class has the following functions:

- **Init():** This function initializes the values dictionary and timestamp dictionary.
- **Set Value(key, value, timestamp):** This function stores the key and value at any given timestamp.
- **Get Value(key, timestamp):** This function returns the value set for this key at the specified timestamp.

> **Note**: When a query requests the value of a key at a timestamp that is more recent than the most recent entry for that key, our data structure should return the value corresponding to the most recent timestamp.

Constraints:

- $1 \leq$ `key.length`, `value.length` ≤ 100
- `key` and `value` consist of lowercase English letters and digits.
- $1 \leq$ `timestamp` $\leq 10^3$
- At most 2×10^3 calls will be made to **Set Value** and **Get Value**.
- All the **timestamps** `timestamp` of **Set Value** are strictly increasing.

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which to follow based on considerations such as time complexity and implementation constraints.

Naive approach

The naive approach uses three individual lists to store the key, value, and timestamp, each in a separate list. To set a value, we'll simply append `key`, `value`, and `timestamp` in their respective lists. To get a value, we'll perform a linear search and search for a specific `value` for the given `timestamp` and `key` throughout the list.

The time complexity to set a value is $O(1)$, whereas to get a value is $O(n)$, where n represents the total number of values in a list. However, the space complexity of the naive approach is $O(n)$.

Optimized approach using binary search

The key idea is to minimize the time complexity by using the binary search instead of linear search. We can use the binary search to implement our logic and can decrease the time complexity to a great extent.

We will use the two dictionaries. The first dictionary, `valuesDict`, stores the values against a specific key. The second dictionary, `timestampDict`, stores the timestamps corresponding to the same key to keep track of the values stored at a specific timestamp.

Set Value(key, value, timestamp): This function adds the `key` with the `value` for the given `timestamp`. The following steps solve the problem:

- We check if the `key` already exists in the `valuesDict` dictionary. If it exists, we perform the following steps:
  - Compare the `timestamp` with the last timestamp stored in the `timestampDict` dictionary for the given key. If the provided timestamp is less than the last stored timestamp, it updates the `value` to be the last stored value for that key.
  - If the provided `value` is not equal to the last value stored in the `valuesDict` dictionary for the given key, we append the `value` and `timestamp` to the lists associated with that key in the `valuesDict` and `timestampDict` dictionaries, respectively.
- If the `key` does not exist in the `valuesDict` dictionary, it creates a new entry in both the `valuesDict` and `timestampDict` dictionaries.

Get Value(key, timestamp): This function returns the `value` set previously, with the highest timestamp for the respective `key`. This function uses the `searchIndex` function, which uses the binary search in its implementation. To implement this function, we initialize the `left` and `right` variables as starting and ending positions of the `timestampDict` dictionary. We then find the middle position and move these pointers to get the required value. If the required value is less than the middle value, we increment the `right` pointer. Otherwise, we increment the `left` pointer.

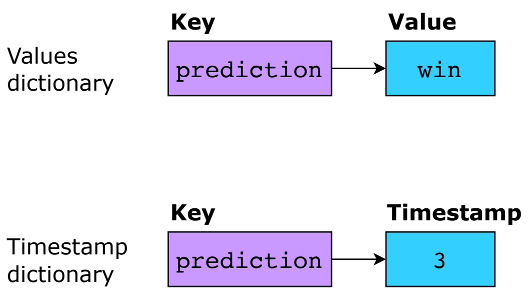
We check the following conditions to get the required value:

- We first verify whether or not the required `key` exists. If it does not exist, then return the empty string.
- If the `key` exists, we check the following conditions:
  - If the given timestamp does not exist but is greater than the timestamp that was set previously, it returns the value associated with the nearest smaller timestamp.
  - If the given timestamp exists, it returns the value associated with the given key and timestamp.

Here’s a demonstration of the algorithm above:

```
Set Value("prediction", "win", 3)
```

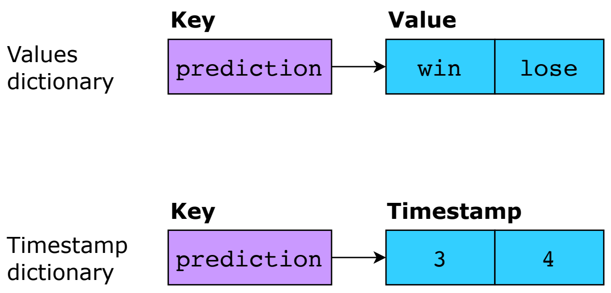
"prediction" is added as a key in Values dictionary and Timestamp dictionary, and "win" and 3 are added as corresponding values in both dictionaries.



1 of 7

```
Set Value("prediction", "lose", 4)
```

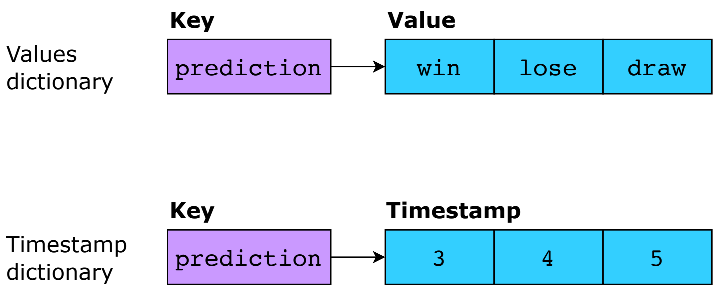
"prediction" is already present as a key in Values dictionary and Timestamp dictionary So, "lose" and 4 are added as corresponding values in both dictionaries.



2 of 7

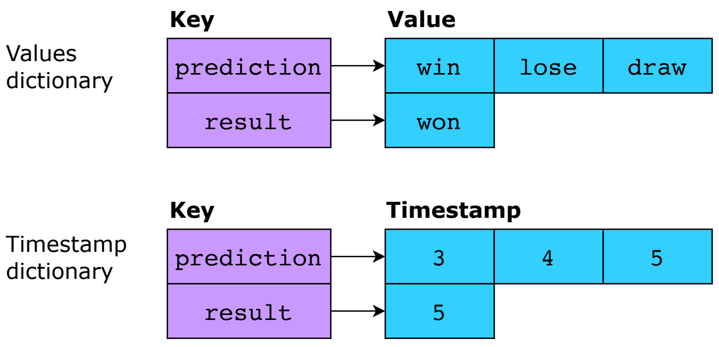
```
Set Value("prediction", "draw", 5)
```

"prediction" is already present as a key in Values dictionary and Timestamp dictionary So, "draw" and 5 are added as corresponding values in both dictionaries.



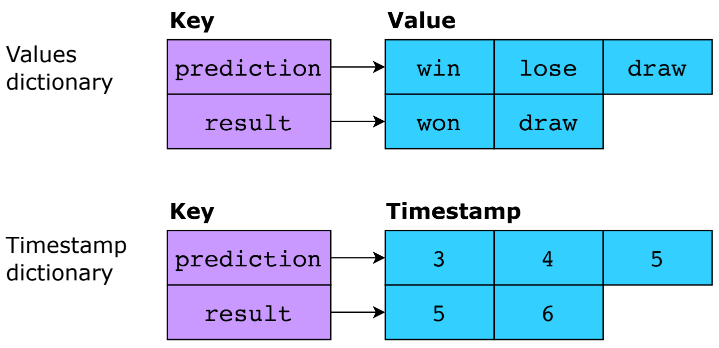
```
Set Value("result", "won", 5)
```

"result" is added as a key in Values dictionary and Timestamp dictionary, and "won" and 5 are added as corresponding values in both dictionaries.



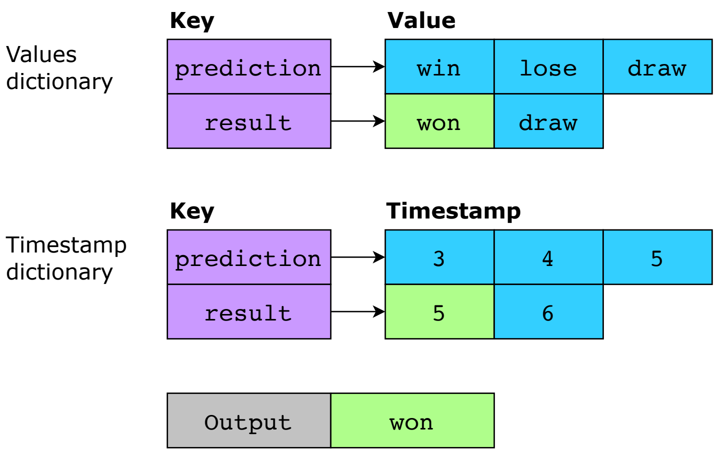
```
Set Value("result", "draw", 6)
```

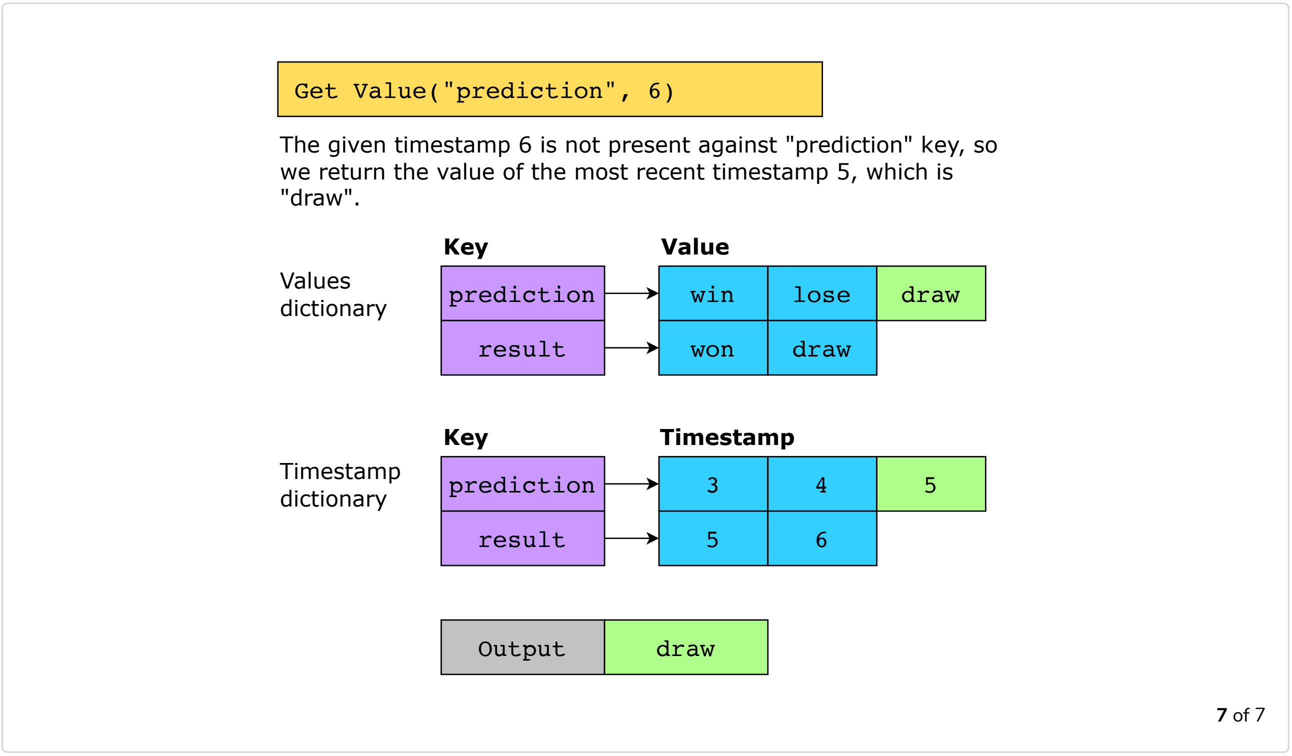
"result" is already present as a key in Values dictionary and Timestamp dictionary So, "draw" and 6 are added as corresponding values in both dictionaries.



```
Get Value("result", 5)
```

The value "won" is present at key "result" and timestamp 5. It is returned as an output.





The given timestamp 6 is not present against "prediction" key, so we return the value of the most recent timestamp 5, which is "draw".

Let’s look at the code for this solution below:

Java

main.java

Triplet.java

```
import java.util.*;

class TimeStamp {
    HashMap<String, List<String>> valuesDict;
    HashMap<String, List<Integer>> timestampDict;
    public TimeStamp() {
        valuesDict = new HashMap<String, List<String>> ();
```