**Introduction to Custom Data Structures**

Let’s go over the Custom Data Structures pattern, its real-world applications, and some problems we can solve with it.

**About the pattern**

Although many coding problems can be solved using existing data structures like arrays, linked lists, stacks, queues, trees, and hash tables, sometimes these structures may not perfectly fit the requirements of a given problem or may not provide the desired efficiency. This is where we need custom data structures. These structures can be implemented using basic data structures as building blocks and incorporate unique features or behaviors specific to the problem domain. In easier words, a custom data structure is the modified version of an existing data structure.

For example, we have to build a web crawler. It starts with a set of seed URLs, visits each page, finds links on each page, and then follows those links to find more pages. Crawling the web means dealing with lots of pages and handling many URLs. Additionally, storing and managing these URLs efficiently while ensuring uniqueness and prioritizing certain pages (e.g., based on relevance or importance) are critical challenges. Basic data structures like arrays or hash tables might not be sufficient to handle the scale and complexity of the web. To address the challenges of this task, a custom data structure, such as a URL queue, can be designed to manage the URLs to be crawled. It is responsible for maintaining a queue of URLs to visit, ensuring uniqueness, and potentially prioritizing URLs based on various criteria.

Using custom data structures makes it easier and more efficient to solve problems that would otherwise be difficult with the existing data structures.

Each custom data structure can be effectively implemented as a class in programming languages like Python, Java, or C++. Classes facilitate abstraction, enabling users to interact with the data structure through well-defined methods and properties without needing to understand the underlying implementation details. Moreover, custom data structures represented as classes allow code reuse.

A yellow sign with black text

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

The following examples illustrates a problem that can be solved with this approach:

1. **Custom stack with getMin() in O(1) complexity:** getMin() **in O(1) complexity:** Design a stack data structure to retrieve the minimum value in O(1) time.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

1. **Two sum III:** Design a data structure that accepts a stream of integers and checks if it has a pair of integers that sum up to a particular value. This data structure will have two main methods:

* add(number): Adds number to the data structure.
* find(value): Returns TRUE if there exists any pair of numbers whose sum is equal to value, otherwise, it returns FALSE.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**Does your problem match this pattern?**

Yes, if either of these conditions is fulfilled:

* **Modification of an existing data structure:** The problem requires customizing an existing data structure, that is, adding a feature to it or modifying an existing feature. Examples include min stack and maximum frequency stack.
* **Multiple data structures involved:** The problem requires combining one or more data structures to solve the problem efficiently. An example would be implementing a Least Recently Used (LRU) cache.

**Real-world problems**

Many problems in the real world share the custom data structure pattern. Let’s look at some examples.

* **Video games:** By modifying/combining the standard data structures, we can maintain the state of the players, levels, and other relevant game details efficiently.
* **Customizing search engines:** Search engines use custom data structures such as customized trees and arrays to quickly search and display data.
* **Managing car parking:** A custom data structure can be utilized to allow efficient tracking of available parking spots, dynamically allocating spaces, and handling reservations and payments seamlessly in multi-level parking garages.

**Solution: Time Based Key-Value Store**

**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)O(1), whereas to get a value is O(n)O(n), where nn represents the total number of values in a list. However, the space complexity of the naive approach is O(n)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. For this, we check if the key already exists in the valuesDict dictionary.

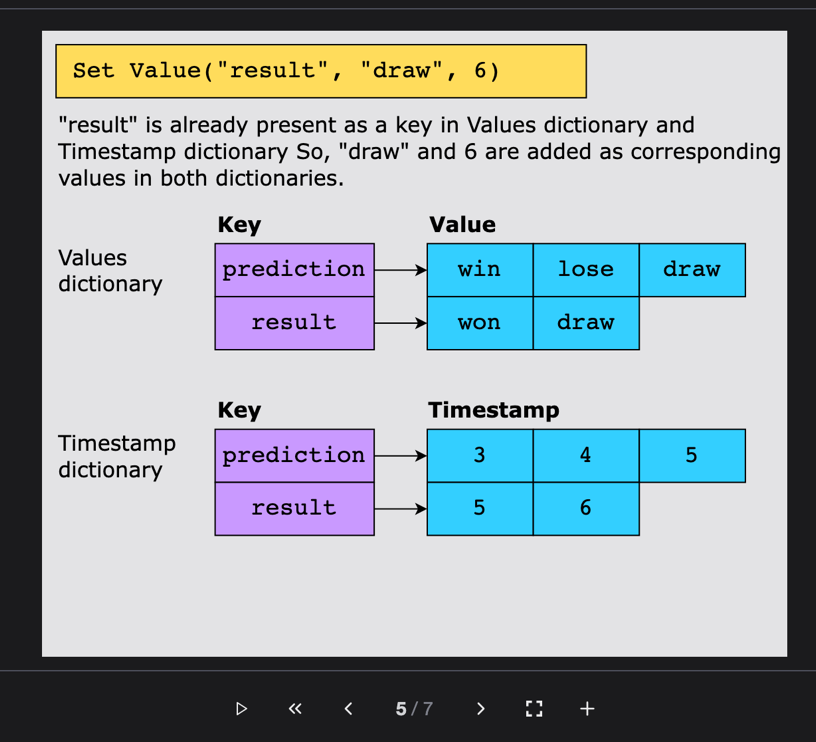
* If the key exists and the provided value is not equal to the last value stored in the valuesDict for this key, we append the value and timestamp to the lists associated with that key in the valuesDict and timestampDict, respectively.
* If the key does not exist in the valuesDict dictionary, it creates a new entry in both the valuesDict and timestampDict dictionaries.

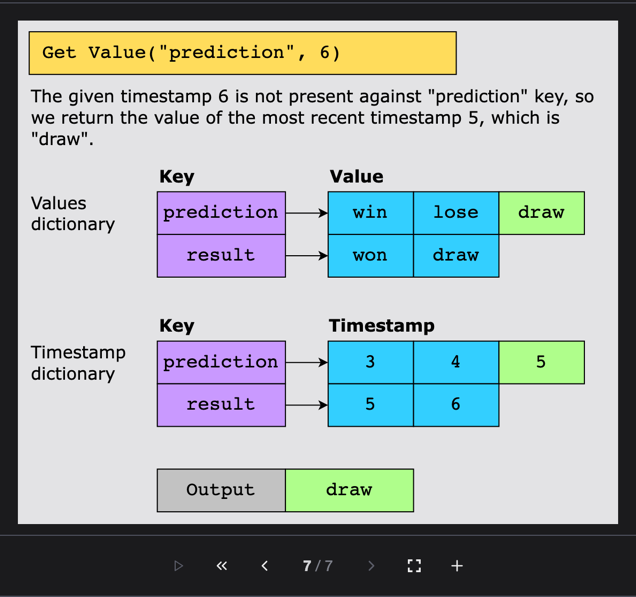
**Get Value(key, timestamp):** This function returns the value set previously, with the highest timestamp for the respective key. This function uses the searchIndex function, which uses the binary search in its implementation. To implement this function, we initialize the left and right variables as 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: A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**Solution summary**

To recap, the solution to this problem can be divided into the following parts:

* **Set Value():** This first checks if the key already exists in the dictionary. If the key exists and the provided value is different from the last stored value for that key, it appends the new value and timestamp to respective lists. If the key is not present, it creates a new entry in both the value and timestamp dictionaries.
* **Get Value():** This checks if the key exists in the dictionary. If it does, it uses binary search to find the index of the rightmost occurrence of the given timestamp in the timestamps list. If an index is found, it returns the corresponding value. Otherwise, it returns an empty string.

**Time complexity**

The time complexity to set the value in the hash map is O(1)O(1). The binary search takes O(log⁡n)O(logn) time to search the element, so the time complexity to get the element will be O(log⁡n)O(logn).

**Space complexity**

The space complexity of the algorithm above is O(n)O(n), where n is the total number of values because we’re calculating the hash map for all the values.