Custom Data Structures: Introduction

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

We'll cover the following

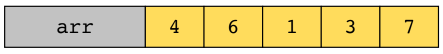
Overview

By now, you’re probably aware of a lot of data structures. Some popular data structures include array, linked list, stack, and queue. Many problems can be solved using these existing data structures. However, there might be instances where you need a more specialized data structure to solve the problem. This is where the custom data structure pattern comes into play. A custom data structure does not have to be something completely – it can be a modified version of an existing data structure. For example, you can modify the tree structure to also include pointers to parents or even add some other data structure like an array for every node.

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

The illustration below shows some commonly used data structures that can be used to make a custom data structure:

Array



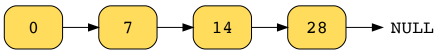
Properties

The most significant properties of arrays are:

1. Arrays store a fixed-size collection of elements of the same data type.
2. The elements in an array are stored in a contiguous block of memory.
3. Due to these two properties, the elements in the array can be accessed by index in O(1) and are iterable.

1 of 4

Linked list



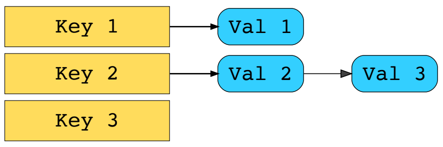
Properties

The most significant properties of a linked list are:

1. Linked lists are used to implement stacks, queues, graphs, etc.
2. Linked lists are dynamic in nature – memory is only allocated when needed.
3. As the addresses of linked list nodes cannot be calculated directly, random access by index is not supported. Instead, accessing an element takes O(n) time.

2 of 4

Hash map



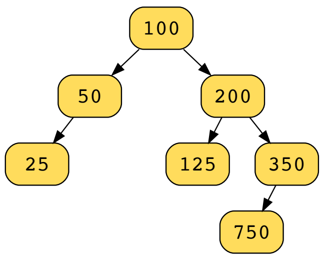
Properties

The most significant properties of hash maps are:

1. Access by key, insertion and deletion take on average O(1) time.
2. Hash maps store data in the form of key-value pairs.
3. Unlike arrays, hash maps can store values of more than one data type.

3 of

Trees



Properties

The most significant properties of trees are:

1. Trees can be seen as undirected graphs with no cycles.
2. In a tree, every node has a parent, except the root node.
3. A node without children is called a leaf node.
4. If the parent node has at most two children, it is a binary tree as shown in the diagram. A tree can have more than two children, such as an n-ary tree.
5. Due to their hierarchical structure, trees naturally support recursion.
6. The most commonly used traversals through trees are breadth-first search and depth-first search.
7. Powerful and widely used subtypes include: binary search trees, heaps, tries and n-ary trees.

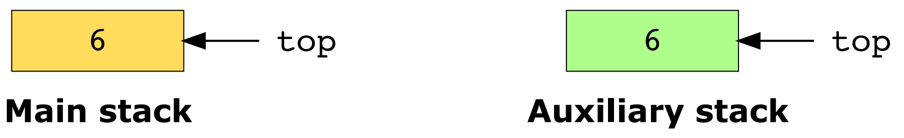
4 of 4

Example

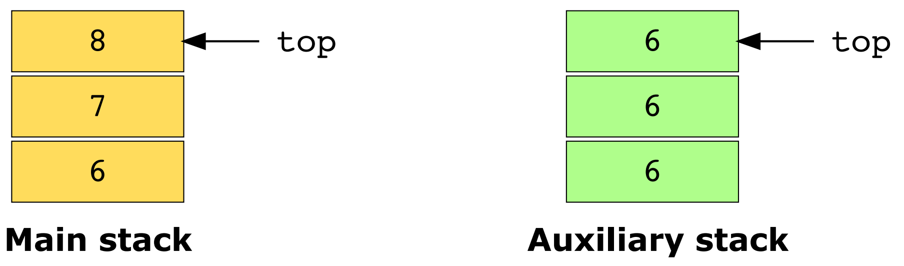
The following example illustrates a problem that can be solved with this approach:

Custom stack with getMin() in O(1) complexity

1. We will store the new elements on to the main stack, and will also push the element on to the auxiliary stack if it is less than the top of the auxiliary stack. Else, the current top of the auxiliary stack is pushed again.

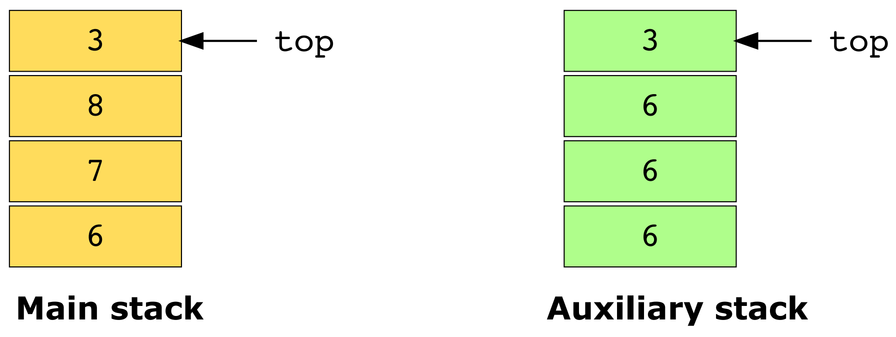


2. Both the elements (7 and 8) pushed are greater than the current top of the auxillary stack, hence 6 is pushed twice.



1 of 2

3. Element 3 is less than 6 hence 3 is pushed to the auxiliary stack too.



By maintaining an auxiliary stack, we can get the minimum in O(1) time complexity.
When pop() is called, the tops of both the stacks are removed.

2 of 2

Does my problem match this pattern?

- Yes, if either of these conditions is fulfilled:
    - 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.
    - The problem requires combining one or more data structures to solve the problem efficiently. An example would be implementing an LRU cache.
- No, if existing data structures may be used to efficiently solve the problem.

Real-world problems

Many problems in the real world share the custom data structure pattern.

- **Keeping various states of games:** By modifying/combining the standard data structures, maintain the state of the players, levels, and other relevant game details efficiently.
- **Used in search engines:** Search engines use custom data structures such as customized trees and arrays to quickly search and display data.
- **Data in tabular format:** Data can be stored in tabular format and accessing data can be made much more efficient by tweaking existing data structures. For example, say you have data in JSON format, which contains nested arrays. In order to access the nested arrays, the entire JSON string needs to be processed to find it. We can design a custom data structure that enables efficient access and updating of these nested arrays.

Strategy time!

Match the problems that can be solved using the custom data structures pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Match The Answer

Select an option from the left-hand side

Find the k^{th} largest element in an array

Custom Data Structures