**People Tech – Week 1 assignment**

**Swetha Kare**

**Learnings for Array Data Structure**

Arrays allow sequential storage of data into the memory and it helps to add elements in contiguous memory locations;

Intertion into arrays is easy if it is added at the end of the current element; if not then we need to shift the elements from their original position till the end which is challenging

Deletion based on index creates a challenge because if we delete an element then we will have to have the positions shift forward as we cannot have any empty places in the array.

Arrays can be both static and dynamic; dynamic arrays are lists in java they keep growing, shrinking as the elements get added or removed. Static arrays have a limited size and adding elements beyond that predefined size is not possible.

**Time complexity:**

Inserting an element at the end : O(1)

Inserting an element at the beginning or in the middle : O(n).

Deleting an element from the end: O(1)

Deleting an element from the beginning or in the middle: O(n).

Traversing an array takes O(n)

Searching an element in an array takes O(n)

Searching an element in binary search takes O(log n)

Space complexity of the array is O(n) it is because the array is allotted with memory even if we use it or not.

If index is known then finding an element in an array is very simple and take O(1) time complexity/

**Examples of Array Data Structure used in the real world.**

In gaming arrays could be used as maps or grids.

In storing elements one after the other and performing sort operations it is easy to use arrays

**Learning for LinkedList Data Structure:**A linked list is a linear data structure where elements (nodes) are connected by pointers. Each node contains data and a reference to the next node. There are different types, including singly linked lists (single direction traversal), doubly linked lists (bi-directional traversal), and circular linked lists (the last node points back to the first).

Basic linked list operations include:

* Creation: a new node is created and other nodes are LinkedIn to it, all the nodes store reference to the next nodes.
* Insertion: Adding a node at the start is O(1) time complexity, to add nodes in the middle or in the end we need to traverse the list and the worst case time complexity to add a node in the middle or end is O(n).
* Deletion: Removing a node from the front is O(1) time complexity while removing the node from the end and at a specified position could be O(n) time complexity.
* Traversal: to obtain a node from the list we need to perform traversal; the time complete of a traversal operation could be O(n)

Comparing arrays and linked lists:

* Arrays allow quick random access (O(1)), but insertions and deletions require shifting elements (O(n)), and resizing an array requires copying data to a new memory block.
* Linked lists dynamically grow or shrink in memory and are more efficient for frequent insertions and deletions (O(1) at the head), though accessing elements requires traversing the list (O(n)).

1. Arrays do not occupy a lot of memory whereas linked lists occupy a lot of memory compared to the arrays because they will be storing the next nodes reference.

***Hands-On Practice*:**

**A screenshot of a computer

Description automatically generated**O(n) time complexity; O(n) space complexity. This is a very simple prefix sum approach   
  
**A screenshot of a computer

Description automatically generated**  
used a hashmap to find two numbers in an array that add up to a target. It iterates over the array, checking if the complement (target - current number) exists in the hashmap. If it does, it returns the indices of the two numbers; otherwise, it stores the current number and its index in the hashmap.  
  
O(n) time and space complexity.  
  
  
  
**A screenshot of a computer

Description automatically generated**  
removing duplicates from a sorted array by using an ArrayList to store unique elements. It iterates through the array, checks if each element is already in the ArrayList, and adds it if it’s not. After processing, it copies the unique elements back into the original array.  
  
time complexity is O(n^2) because it has ar.contains() which will check n times for each elemts of the n sized array

Space complexity is O(n).

1. **A screenshot of a computer

   Description automatically generated**incrementing a number represented by an array of digits. It starts from the last digit, adding 1 to it. If the digit is 9, it sets it to 0 and moves to the next digit. If all digits are 9 (e.g., 999), a new array is created with an additional digit to handle the carry (e.g., 1000).  
     
   Time Complexity and space complexity is O(n).

**Learning for Stacks:**

Stacks are for LIFO operations; can be implemented with any data structure   
A stack is a data structure that follows the Last In, First Out (LIFO) principle, meaning the most recently added item is the first to be removed. It operates with key operations such as push, pop, peek, isEmpty, and Size.

Stacks can be implemented using arrays, linked lists, deques, or queues. With arrays, the stack's size may be fixed or dynamic, with simple and fast access but potential resizing overhead. Linked lists offer dynamic sizing with efficient insertions and deletions, though they require extra memory for node references. Deques provide optimized operations for adding and removing from one end, while two queues can simulate stack behavior with more complex operations.

The time complexity for stack operations is generally O(1), meaning they are performed in constant time. The space complexity is O(n), where n is the number of elements, reflecting the memory needed to store the stack's items.

Stacks are used in various applications, including managing function calls, evaluating expressions, handling undo/redo operations, backtracking algorithms, and checking balanced parentheses. They are simple to implement and efficient for many problems but have limited direct access to elements and can involve memory overhead depending on the implementation.  
  
**Learning for Queues:**A queue is a data structure that follows the First In, First Out principle, meaning the first item added is the first to be removed. It operates with key operations such as enqueue to add an item to the back, dequeue to remove an item from the front, peek to view the front item without removing it, isEmpty to check if the queue is empty, and Size to return the number of items.

Queues can be implemented using arrays, linked lists, or deques. With arrays, the queue’s size may be fixed or dynamic, offering simple and fast access but potentially requiring resizing. Linked lists provide dynamic sizing with efficient insertions and deletions at both ends, though they require extra memory for node references. Deques, which allow additions and removals from both ends, can be used to implement queues with optimized operations.

The time complexity for queue operations is generally O(1), meaning they are performed in constant time. The space complexity is O(n), where n is the number of elements, reflecting the memory needed to store the queue's items.

Queues are used in various applications, including task scheduling, managing requests in web servers, handling print jobs, and implementing breadth-first search in algorithms. They are simple to implement and efficient for managing ordered data but can involve memory overhead depending on the implementation.

**Learning for Hash Tables:**

A hash table is a data structure that provides fast access to data using a mechanism known as hashing. It works by mapping keys to values via a hash function, which computes an index into an array where the value is stored. The key operations are insertion, where a key-value pair is added; deletion, where a key-value pair is removed; and retrieval, where a value is accessed using a key.

Hash tables can be implemented using arrays and various collision resolution techniques, such as chaining (where each array index points to a linked list of entries) or open addressing (where colliding elements are placed in the next available slot according to a probing sequence). Chaining allows hash tables to handle collisions more flexibly, while open addressing often provides better cache performance but requires careful handling of collisions and deletions.

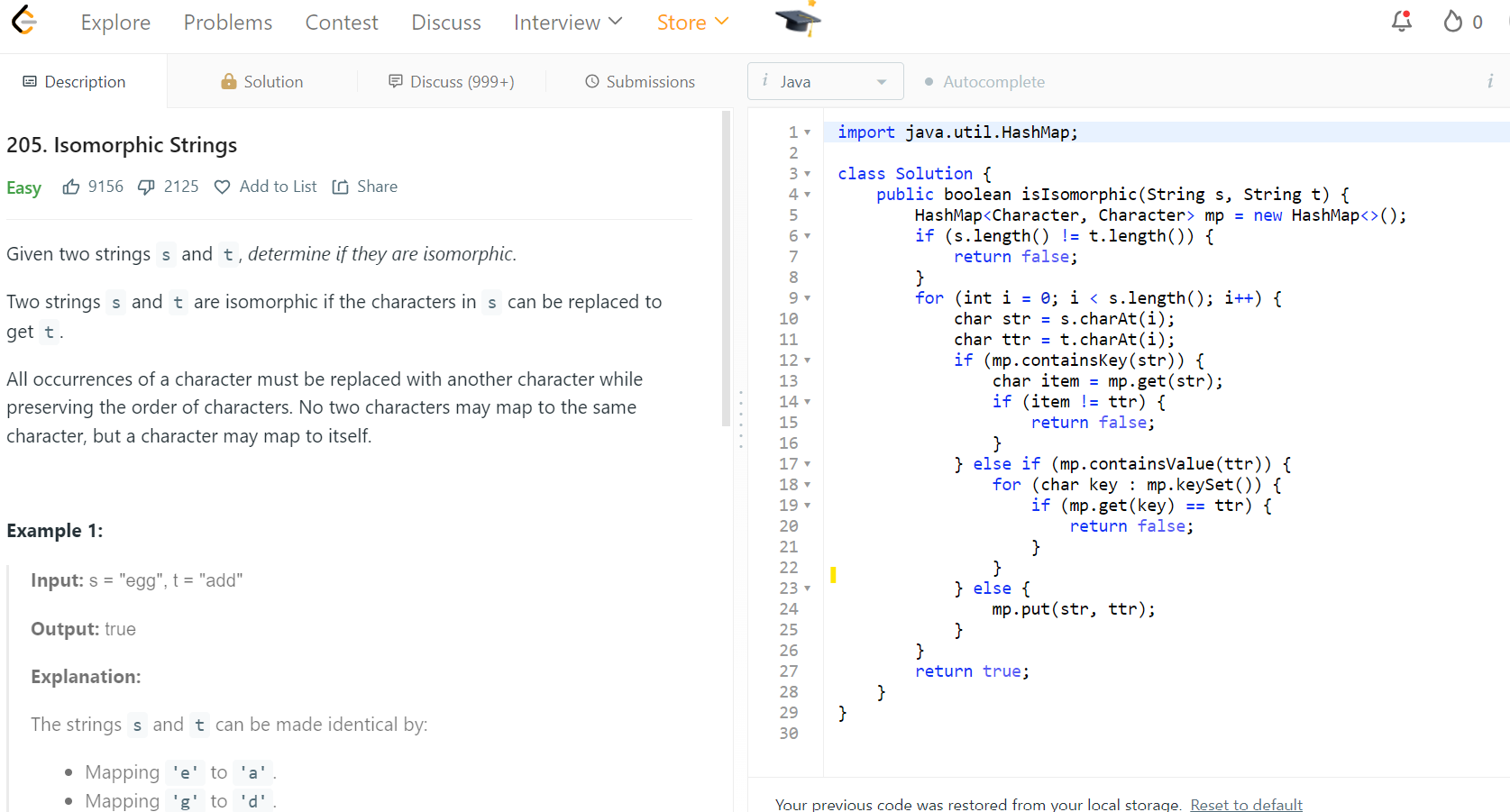
The time complexity for average-case operations such as insertion, deletion, and retrieval is generally O(1), meaning they are performed in constant time, assuming a good hash function and minimal collisions. However, in the worst case, due to collisions, operations can degrade to O(n), where n is the number of elements.

The space complexity is O(n), where n is the number of elements, reflecting the memory needed to store the hash table's entries.

Hash tables are widely used in various applications, including implementing associative arrays, database indexing, caching, and sets. They offer efficient access and management of data but can suffer from issues related to hash function quality and collision handling.

Practiced a few easy level problems on leetcode for stacks, queues, hashtables, Linkedlist.

A screenshot of a computer

Description automatically generated  
  
A screenshot of a computer

Description automatically generated  
  
**Algorithms**:

**Insertion Sort** is a straightforward sorting algorithm that works by gradually building up a sorted portion of the array. It starts with the first element and iteratively inserts each subsequent element into its correct position within the sorted portion. The algorithm involves comparing the current element to those in the sorted section and shifting elements as needed to make space for the new element. Its time complexity is O(n²) in average and worst cases, making it less efficient for large datasets, but it performs well for small or nearly sorted arrays. Its space complexity is O(1), as it sorts in place without requiring extra space beyond the input array. Insertion Sort is often used in applications where the dataset is small or partially sorted, such as in simple database sorting or as part of more complex algorithms that handle small partitions of data.

**Merge Sort** is a more advanced sorting algorithm that follows a divide-and-conquer approach. It divides the array into smaller subarrays, recursively sorts each subarray, and then merges them back together in a sorted manner. This method ensures that the entire array is sorted efficiently. Merge Sort has a time complexity of O(n log n) across average, worst, and best cases, making it much more suitable for large datasets compared to Insertion Sort. However, it requires additional space proportional to the size of the input array, giving it a space complexity of O(n). Merge Sort is particularly useful for sorting large volumes of data, such as in data processing applications, and is also well-suited for linked lists and external sorting where data is too large to fit into memory all at once.

**Applications**

**Insertion Sort**:

* **Small Datasets**: Ideal for small arrays or lists where its simplicity and in-place sorting are advantageous.

**Merge Sort**:

* **Large Datasets**: Efficient for sorting large volumes of data, such as in database management systems or large-scale data processing applications.

In summary, Insertion Sort is efficient for small or nearly sorted data, while Merge Sort is better suited for large datasets and scenarios requiring stable sorting.  
  
A screenshot of a computer

Description automatically generated

**Dijkstra's Algorithm** finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights. It uses a priority queue to explore the closest vertex, updating the shortest known distances as it proceeds. This algorithm is efficient with a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges.  
A screenshot of a computer

Description automatically generated

**Bellman-Ford Algorithm** computes the shortest paths from a source vertex to all other vertices in a graph, handling negative edge weights. It works by repeatedly relaxing all edges and can detect negative-weight cycles. The time complexity is O(VE), making it less efficient than Dijkstra's for graphs with many edges.  
A screenshot of a computer

Description automatically generated

**Breadth-First Search (BFS)** explores a graph level by level, starting from a source node and visiting all its neighbors before moving to the next level. It is used for finding the shortest path in an unweighted graph and has a time complexity of O(V + E).  
A screenshot of a computer

Description automatically generated

**Depth-First Search (DFS)** explores as far as possible along each branch before backtracking. It is used for tasks such as pathfinding, cycle detection, and topological sorting. Its time complexity is O(V + E), and it can be implemented using recursion or a stack.  
A screenshot of a computer

Description automatically generated

**Fractional Knapsack Problem** is a variation of the knapsack problem where items can be divided into smaller parts. It involves maximizing the total value in a knapsack with a weight limit, allowing fractional amounts of items to be included. It is solved using a greedy algorithm, making it efficient for finding optimal solutions when item fractions are allowed.

A screenshot of a computer

Description automatically generated