# Q1

1. Explain the concept of (Re)Sorting (On Demand). Make certain to include why (re)sorting on demand may still be useful or necessary despite us already having access to the Array-based Sorted List (ASList) and Binary Search Tree (BST) implementations.

(Re)Sorting on demand is a strategy where data is kept unsorted during insertions and only sorted when needed like before iteration or searching. This approach offers faster insertions (O(1)) compared to continuously sorted structures like Array-based Sorted Lists (ASList) or Binary Search Trees (BST), which have higher insertion costs. It’s especially useful when sorted access is infrequent or when sorting criteria may change at runtime. Despite the availability of ASList and BST, on-demand sorting remains valuable for its simplicity, flexibility, and efficiency in the right contexts.

1. Explain the principal differences between the Insertion Sort and Heap Sort algorithms. Be certain you Provide a general idea of how the algorithms operate and explain which algorithm is generally the most efficient in the worst-case scenario and explain which of the two algorithms is likely preferable if a collection is already sorted or almost entirely sorted. Your response should be 1-2 paragraphs in length.

Insertion Sort is a simple, intuitive algorithm that builds the final sorted list one element at a time by comparing each new element to those already sorted and inserting it in the correct position. It works well for small datasets and is particularly efficient when the input is already sorted or nearly sorted, with a best-case time complexity of O(n) and a worst-case of O(n²).

Heap Sort, on the other hand, uses a binary heap data structure to sort elements. It first builds a max heap from the input data, then repeatedly removes the largest element and rebuilds the heap until the list is sorted. Heap Sort has a consistent O(n log n) time complexity in all cases, making it more efficient in the worst-case scenario than Insertion Sort. However, if the data is already sorted or nearly sorted, Insertion Sort is typically preferable due to its lower overhead and faster performance in those specific cases.

# Q2

1. Explain what the Set ADT is, and what distinguishes it in practice from the two List ADTs (Unsorted List and Sorted List).

The Set Abstract Data Type (ADT) is a collection that stores unique elements, meaning no duplicates are allowed, and it typically does not maintain any specific order among elements. The primary operations supported by a Set include adding elements, removing elements, and checking for membership. What distinguishes the Set ADT in practice is its focus on uniqueness and fast membership testing, often implemented with hash tables or binary search trees to allow efficient lookups, rather than maintaining order or duplicates like List ADTs.

1. Provide an example when using the bit-vector implementation of a set can be advantageous, and another example when it is a poor choice.

A bit-vector implementation of a set can be advantageous when dealing with a set of non-negative integers within a small, known range. However, it is a poor choice when the universe of possible values is large or unbounded, or when storing non-integer or sparse data.

# Q3

1. Explain what a hash function is and provide an example of how one can be applied to numeric data and non-numeric data.

A hash function is a function that takes an input (called a key) and maps it to an integer, usually used as an index in a hash table. For numeric data, a simple hash function might distribute numbers based on their remainder. For non-numeric data like strings, a hash function could convert each character to its ASCII value, combine them using arithmetic, and then take the result modulo the table size

1. Explain what a hash table is, and what performance advantage it carries relative to our prior data structure implementations.

A hash table is a data structure that uses hash functions to store and retrieve values efficiently. It offers average-case constant time (O(1)) for insertions, deletions, and lookups, which is significantly faster than lists (O(n)) or even binary search trees (O(log n)) for large datasets.

1. Explain what issues can hamper this performance advantage, and what steps are taken to mitigate the slow-down.

This performance can be hampered by collisions, where different keys hash to the same index. Collisions are mitigated using techniques like chaining or open addressing. Resizing the table and using good hash functions help maintain performance as the number of elements grows.

1. Suppose employees at a company are assigned an 8-digit employee ID according to the following criteria:
   1. The first four digits correspond to a department code (ex: the accounting department may have code 0051, the IT department code 3505, etc.)
   2. The second four digits correspond to the order in which the employee was hired within the company on the whole (ex: the second employee would have the last four digits “0002,” the one-hundredth “0100,” etc.)
   3. Now, suppose you are given the task to store employee information in a small hash table by index and are given the following options for the hashing function based on the employee ID:
      1. Option 1: Compute the hash index using the first four digits of the employee ID, mod 100.
      2. Option 2: Compute the hash index using the last four digits of the employee ID, mod 100.

Option 2 is better because the first four digits represent the department code, and there are only five departments, meaning only a few distinct values will ever appear. Modding these by 100 would result in a very small and skewed range of hash indices, leading to heavy clustering and collisions in the hash table. Option 2 deals with the last four digits, they can vary more widely and increase sequentially with each hire. If there are several hundred employees, these digits will range from 0001 to, say, 0500, meaning mod 100 will spread the values more evenly across the table, resulting in better distribution and fewer collisions.

* 1. Assuming the company has several hundred employees and five departments, i. Explain which Option is better for this dataset, and why, and ii. Explain how the chosen hash function can be improved with a simple modification (there may be several valid answers to part ii, but there is one that is very straightforward).

The easiest improvement is to combine both parts of the ID for hashing. So, combining Option 1 and Option 2.