# Assignment 3

# Part 1

Selecting a random pivot for every subarray helps the Randomized Quicksort algorithm to be a fast sorting approach that reduces the possibility of worst-case performance. When the array is already sorted or reverse-sorted, situations in which conventional quicksort would have difficulty, this method is very helpful. The main purpose, `randomized\_quicksort`, provides the sorting process starting point. It calls for three parameters: the `array` itself; `low` and `high` indices, which define the limits of the current subarray. The function runs recursively; the base case occurs when the subarray is already sorted—that is, when it contains only one or zero entries. The method `randomized\_partition` inside `randomized\_quicksort` selects a pivot at random. Avoiding biassed splits that might result in poor performance depends on this pivot choosing procedure. Randomized\_partition chooses a random index between low and high, swaps the element at that index with the final element of the subarray, then places the pivot at the end of the array segment. This layout makes the following step—partitioning the array around the pivot—simpler. Organizing the array around the pivot falls to the partition function. It lays components bigger than the pivot on its right and less than or equal on its left. A pointer `i` follows the last smaller element found. Any element less than or equal to the pivot is swapped with the element at the `i` pointer as the function iteratively over each element moves `i` ahead. The pivot is immediately after the final smaller element in proper orientation when the loop ends. Following partitioning, `randomized\_quicksort` recursively sorts the left and right halves of the array using the pivot's index. Randomized Quicksort is a dependable and efficient sorting algorithm as this random pivot selection and recursive division provide an average-case time complexity \( O(n \log n) \).



On the input array [3, 6, 8, 10, 1, 2, 1] the Randomized Quicksort method produces the sorted output [1, 1, 2, 3, 6, 8, 10]. This properly arranged array shows how well the method handles and arranges duplicate values—like the two 1, in a sorted sequence cases. Especially in cases where the array is already sorted or reverse-sorted, the method efficiently organizes the items in ascending order by using random pivot selection, therefore avoiding the possible hazards of worst-case performance that may develop in conventional quicksort. The Randomized Quicksort method effectively divides the array around a randomly selected pivot for this particular input, therefore producing balanced averages. Maintaining an average-case temporal complexity of 𝑂 (𝑛 log ) O(nlogn) depends on this equilibrium. As demonstrated with this input, the method performs well even with unsorted and repeated items, therefore verifying that randomized pivot selection adds to the resilience and efficiency of the approach. Regardless of the starting order of the array, the last sorted result, [1, 1, 2, 3, 6, 8, 10], emphasizes how well Randomized Quicksort achieves correct sorting. This demonstrates how well the method avoids degenerate situations and guarantees consistent sorting across many kinds of input.

# Part 2

Chaining a hash table helps one to effectively manage collisions. Under this method, every hash table entry—or "bucket—is shown as a list with many key-value pairs. The hash table starts with a default size of 10 but may dynamically expand as the number of entries rises, therefore preserving efficiency as it develops. By modulating the hash value of each key with the table size, a hash function maps each key to a particular index in the database thereby guaranteeing that the index fits inside the boundaries of the table. This method lets important value pairs be efficiently stored and retrieved within the suitable bucket.  
  
The insert function computes the index where the key-value pair will be kept using the hash function therefore enabling either insertion or updating of the pair. Should the bucket at that index already contain the key, the current value is changed. Should the key absent, it finds its way into the bucket. Following each insertion, the load factor— computed as the element to bucket ratio—is examined. Should the load factor surpass a threshold, say 0.75, the hash table automatically doubles its starting size. Rehashing all current key-value pairs to suit the bigger database guarantees effective data distribution and helps to lower collisions in this scaling operation. The search capability gets the value connected to a given key. Computation of the key's index using the hash function allows one to immediately search for the key from the suitable bucket. Should the key be located, the function generates the corresponding value; should not be the case, it generates None. In a similar vein, the delete operation finds the bucket based on the index of a key, then locates and deletes the key-value pair inside that bucket. Should the deletion be successful, the element count decreases and True is returned. Should the key elude us, it returns False. Maintaining the performance of the hash table depends much on the load factor. A low load factor indicates that the key-value pairs are equally distributed over buckets, therefore lowering the probability of collisions and maintaining the time complexity for insert, search, and delete operations near to 𝑂 (1). Higher load factors, however, increase the likelihood of collisions, which results in longer lists inside buckets and slowed down performance. The hash table automatically extends as the load factor above 0.75, therefore preserving effective operations even as the number of entries increases. This chaining-based hash table offers a strong way to handle collisions while maintaining best performance by combining dynamic resizing with efficient hash.

A screenshot of a computer program

Description automatically generated

Three key-value pairs—("apple," 100), ("banana," 150), and ("grape," 200)—added show the hash table's underlying structure. Every bucket stands for a list where many objects may be kept; the hash function decides which bucket each key is allocated to. For this instance, the hash function puts `"apple"` in bucket 6, `"banana"` in bucket 9, and `"grape"` in bucket 8. Printing the table exposes the particular bucket where every key-value pair has been kept, therefore highlighting the entry distribution. The software then looks for the key `"apple"` and finds it in bucket 6 with success, therefore producing the corresponding value, 100. The programme then does a delete operation on the key `"banana". The table is displayed once again upon deletion, verifying that bucket 9 is empty now and therefore showing that "banana" has been effectively deleted. The other important value pairs—`"apple"` in bucket 6 and `"grape"` in bucket 8—remain unaltered. This example shows how well the hash table can manage simple operations such insertion, search, and deletion. Although there were no collisions in this particular case, the usage of chaining guarantees that, should numerous objects hash to the same bucket, every item may still be stored and retrieved using distinct lists inside each bucket. By letting many entries cohabit in a single bucket while preserving orderly and accessible storage, this chining technique is efficient for controlling possible collisions.