Part 1

Designed to quickly sort an array by choosing a random pivot element for every subarray, the Randomized Quicksort method used here is Especially in cases where the array is already sorted or reverse-sorted, this method helps avoid the worst-case performance situations often seen with conventional quicksort. The basic entrance point for array sorting, the randomized\_quicksort function forms the foundation of the method. Three parameters are needed: the low and high indices, which stand for the subarray under current sorting; furthermore, the array itself. The procedure is recursive; the base case for recursion is when the low index is already sorted and shows one or zero items, therefore signaling that the subarray contains one or zero elements and is hence already sorted. The function calls randomized\_partition to choose a random pivot then recursively sorts the subarrays left and right of the pivot. Selection of a random pivot element from the subarray is accomplished by the randomized\_partition function. It achieves this by selecting at random an index between the low and high indices. The element at this random index is then exchanged with the final element of the subarray so that the pivot is always found at the array's end. This layout streamlines the following step's partitioning procedure. The function calls partition to split the subarray around the pivot element after selection and swapping of the pivot. Actually, the partition function divides the array. The function guarantees that all components bigger than the pivot are positioned on its right; all elements less than or equal to the pivot are put on its left. It keeps a pointer i tracking the location of the last smaller element put to do this. Every element in the subarray is compared to the pivot as the function runs over it. Should the element be less than or equal to the pivot, it is swapped with the element at the i pointer, therefore increasing the pointer. The pivot element is switched into its proper place—that of the position exactly after the previous smaller element—when the loop finishes. The recursive calls in the randomized\_quicksort function utilize the index of the pivot that the function then returns to keep sorting the left and right subarrays. This method guarantees that, by balancing the array by random pivot selection, the algorithm has an average-case time complexity of 𝑂 (𝑛 log⁡ O(nlogn), hence rendering it a strong and effective sorting method.



Applying the Randomized Quicksort method on the above input array [3, 6, 8, 10, 1, 2, 1] produces the shown output [1, 1, 2, 3, 6, 8, 10]. Currently sorted as [1, 1, 2, 3, 6, 8, 10]. The program appropriately ordered and managed repeated elements—two instances of 1. By using random pivots throughout the sorting process, the method also sorted the array in ascending order, therefore helping to prevent the worst-case performance that may arise in conventional quicksort should the array be either reverse-sorted or previously sorted. For most circumstances, including this input, the result shows that Randomized Quicksort effectively divides the array around a randomly selected pivot, producing balanced partitions on average, thereby maintaining the time complexity at 𝑂 (𝑛 log). The technique runs well even with repeated pieces and unordered input. This confirms the efficiency of randomized pivot selection as it guarantees that the input array, independent of its beginning order, is effectively and accurately sorted avoiding degenerate situations. The predicted outcome—the sorted output [1, 1, 2, 3, 6, 8, 10] shows the implementation's strength.

Part 2

To manage collisions in the second section of the work, a hash table used using the chaining approach Each bucket in the hash table—which is shown as a list of lists—stores many key-value pairs. Originally set with a default value of 10, the hash table may dynamically expand as the count of entries rises. With a hash function, every key maps to a particular index in the hash table. By use of the modulo of the hash value of the key and the table size, the hash function generates an index. This guarantees that the index always falls within the range of the table size, therefore enabling effective storage of key-value pairs in the suitable bucket. Users of the insert feature may add or change hash table key-value pairs. The hash function computes the index where the key-value pair will be kept after adding a fresh key. Should the bucket already have the key, its value is changed. Should the key not be found, it is added to the bucket or list at the calculated index. The table's load factor is examined after every insertion to decide if resizing is required. The load factor is computed as the element count divided by bucket count. Should the load factor be higher than a threshold—in this example, 0.75—the hash table is enlarged by double its size and rehashing all current key-value pairs into the new, bigger table. The search operation gets the value connected to a certain key. After computing the key's index using the hash function, it searches the bucket—that is, list—at that index. Should the key is located, its related value is returned. Should the key prove elusive, the function yields None. The delete operation takes a key-value pair off of the hash table. First determining the index of the key, it then finds and removes the key-value pair from the list at that index, working much like the search function does. Should the key be effectively eliminated, the element count decreases and the method returns True. Should the key prove elusive, the function generates False. The load factor determines the performance of the hash table really precisely. A low load factor guarantees that important-value pairs are distributed equally throughout the buckets, therefore reducing the possibilities of collisions and maintaining the time complexity for insertion, search, and deletion operations near to 𝑂 (1). Longer lists in the buckets result from increased collision risk brought on by increasing load factor, therefore impairing performance. The table is dynamically enlarged once the load factor surpasses 0.75 to guarantee that operations remain effective even as the table increases. By means of dynamic resizing and effective hash, this use of a hash table with chaining offers a strong method for managing collisions and preserving best performance.

A screenshot of a computer program

Description automatically generated

After adding three key-value pairs—("apple," 100), "banana," 150, and "grape," 200—the state of the hash table shows in the output of the hash table with chaining implementation. Every bucket stands for a list; the hash function's output determines which bucket to store the keys in. Here apple is kept in bucket 6, banana in bucket 9, and grape in bucket 8. The table prints revealing which bucket each three key-value pair has been put into once they have been entered. The software then looks for the key "apple," thereby locating it in bucket 6 and returning the related value 100. The important "banana" is then erased; thus, the hash table is printed once again to verify the deletion. Bucket 9 is empty now, indicating that "banana" has been effectively taken off the table; the other key-value pairs in their respective buckets remain whole. This result shows how well the hash table handles simple operations such insertion, search, and deletion. Though in this case no collisions happened, it also demonstrates how effectively chaining stores many pieces in distinct lists inside buckets resolves collisions.