GITHUB Link: https://github.com/vtadigotla35101/MSCS532_Assignment3

Part-1: Randomized QuickSort Analysis:

I have implemented the randomized Quicksort algorithm and chosen pivot element uniformly from the subarray being partitioned. Also I have ensured that my implementation handles various edge cases.

a.Theoretical Analysis: On why this Randomized Quicksort is (O(n \log n))?
At each recursive step the Quicksort partitions the array into two subarrays: [T(n) = T(k) + T(n-k-1) + O(n)] where the term k is the number of elements before the pivot. If the pivot is uniformly random each partition is balanced on average: [T(n) = 2T(n/2) + O(n)] Using the Recurrence Tree Method, this expands to: [O(n) + O(n/2) + O(n/4) + ... = O(n \log n)]

b.Observations:

1. Randomized Quicksort always beats the Deterministic Quicksort with random datasets, please verify the attached test results accordingly for each of the different sorts of tests.
2. Deterministic Quicksort struggles with sorted and reverse sorted data ($O(n^2)$).
3. Memory usage differences show recursion depth impact.
4. Repeated elements cause occasional imbalance in both approaches.

I have kept all the test results in the test file test_quicksort_results.txt accordingly, please review it for the same.

c.Theoretical vs. Practical Differences:

1.Expected behavior: Randomized Quicksort should maintain (O(n \log n)) performance on average.
2.Observed behavior: Deterministic Quicksort performs worse in worst-case scenarios like sorted and reverse sorted data.And this is because deterministic pivot selection leads to skewed partitions possibly which is making deterministic quicksort to perform worst.

****

2. HashTable chain Analysis

We analyze the expected time complexity of operations under the assumption of Simple Uniform Hashing. This assumption states that any given key is equally likely to hash into any of the m slots, independently of where any other key hashes.

- n be the number of elements (key-value pairs) currently stored in the hash table.
- m be the number of slots (capacity) in the hash table.
- $\alpha = n/m$ be the **load factor**. This is the average number of elements per chain.

Expected Search, Insert, and Delete Times

The performance of these operations largely depends on the length of the chain in the slot where the key hashes.

1. **Search:**
    - Average Case: O(1+α)
        - O(1) for computing the hash function and accessing the correct slot.
        - O(α) for traversing the linked list (chain) at that slot, as on average each chain has α elements. If the key is not found, we traverse the entire chain.
    - Worst Case: O(n)
        - This occurs if all n keys hash to the same slot, forming a single chain of length n. Then, searching involves traversing this entire list.
2. **Insert:**
    - Average Case: O(1+α)
        - O(1) for computing the hash and finding the correct slot.
        - O(α) for checking if the key already exists in the chain (necessary if we're updating values). If not found, appending to the end of a Python list (which is used for chains) is amortized O(1), so the dominant factor is the search.
    - Worst Case: O(n)
        - If all keys hash to the same slot, checking for existence takes O(n), and then appending is fast.
3. **Delete:**
    - Average Case: O(1+α)
      a. O(1) for hashing and accessing the slot.
      b. O(α) for finding the element in the chain. Once found, deleting from a Python list is O(length of list - index), which in the average case for a chain is O(α).
    - Worst Case: O(n)
      a.All keys hash to the same slot, requiring O(n) to find and delete.

How Load Factor Affects Performance: The load factor is always critical:

1. **Low (if LoadFactor less than 1):**
   a.Chains are short, often containing 0 or 1 element.
   b.Operations approach O(1) time. This is the ideal scenario for hash tables.

2. **High (If LoadFactor greater than 1):**
   a.Chains become too long as expected.
   b.Operations degrade towards O(n), effectively it turns the hash table into a linked list.
   c.Collisions become more frequent, increasing the work needed to traverse chains.

Strategies for Maintaining a Low Load Factor and Minimizing Collisions

1. **Good Hash Function:**

- **Universal Hashing:** using a hash function from a universal family ensures that for any set of n keys, the probability of a not great distribution is very low, even if an adversary chooses the keys. This is crucial for guaranteeing expected O(1+α) performance. Python's built-in hash() function is designed to be robust and perform well for various object types though it's not strictly a universal hash function in the theoretical sense.
- **Distribute Keys Evenly:** The hash function should produce a wide range of hash values that are evenly distributed across the table's capacity. Avoid functions that might map many common keys to the same index.

2. **Dynamic Resizing (Rehashing):**
   - This is the most common and effective strategy to maintain a low load factor.
   - **Thresholds:**
     1. **Expansion:** When the load factor exceeds a certain threshold (if α>0.7 or 0.75), the hash table capacity is increased like it is almost doubled.
     2. **Shrinking (Optional):** When the load factor drops below another threshold (if α<0.2 or 0.25) and the table is sufficiently large, the capacity can be reduced like almost halved to save memory.
   - **Process:**
     1. Create a new, larger (or smaller) hash table array.
     2. For each key-value pair in the old hash table:
        - Calculate its new hash index using the new capacity.
        - Insert it into the new hash table.
     3. Replace the old table with the new table.
   - **Cost of Resizing:** Resizing is an O(n) operation because all n elements must be rehashed and reinserted. However, because it occurs infrequently (only when the load factor crosses a threshold), the amortized cost of insert and delete operations remains O(1). This is similar to how appending to a Python list has amortized O(1) cost even though occasional reallocations are O(n).

Conclusion:

Hashing with chaining is a highly effective data structure due to its average-case O(1) performance for fundamental dictionary operations (insert, search, delete). This efficiency relies heavily on a well-chosen hash function to minimize collisions and dynamic resizing to keep the load factor within an optimal range. While the worst-case scenario can be O(n), its low probability (especially with randomized hashing and proper load factor management) makes it a preferred choice for many applications.

GITHUB Link: https://github.com/vtadigotla35101/MSCS532_Assignment3