Assignment 3: Understanding Algorithm Efficiency and Scalability

Milan Bista

University of Cumberlands

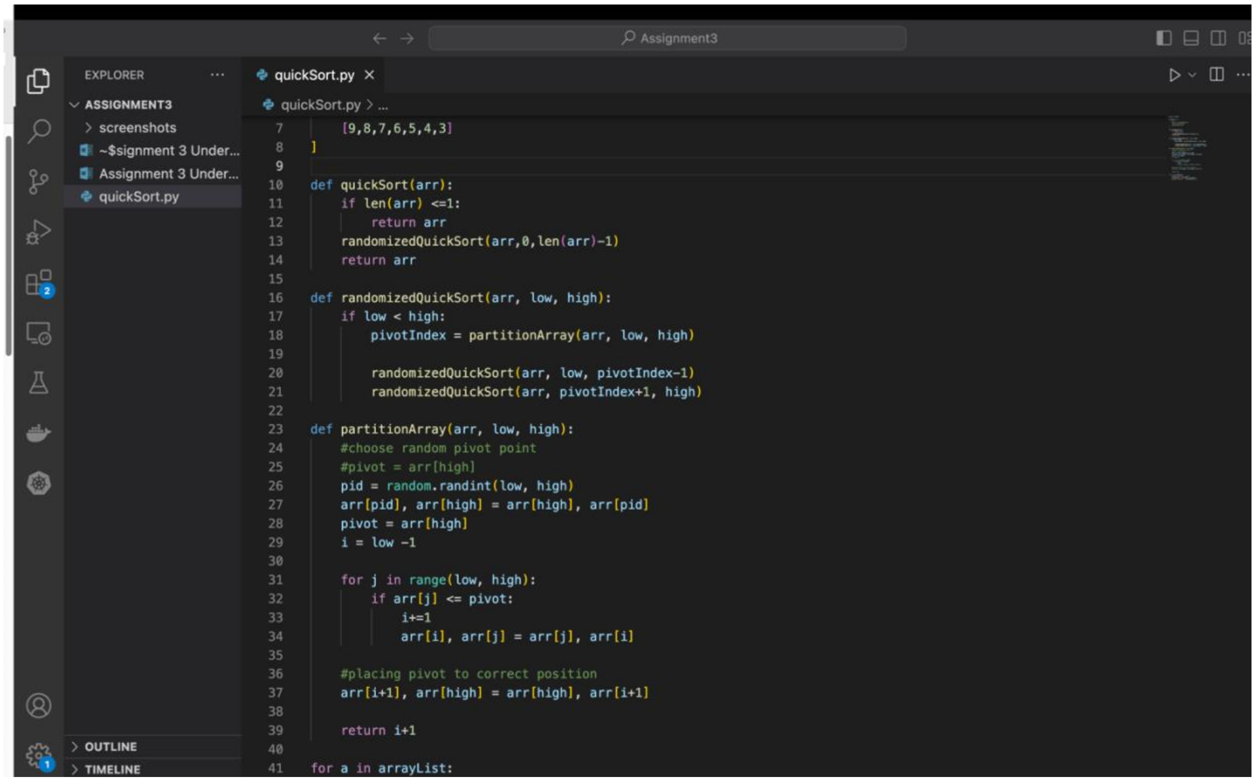
2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

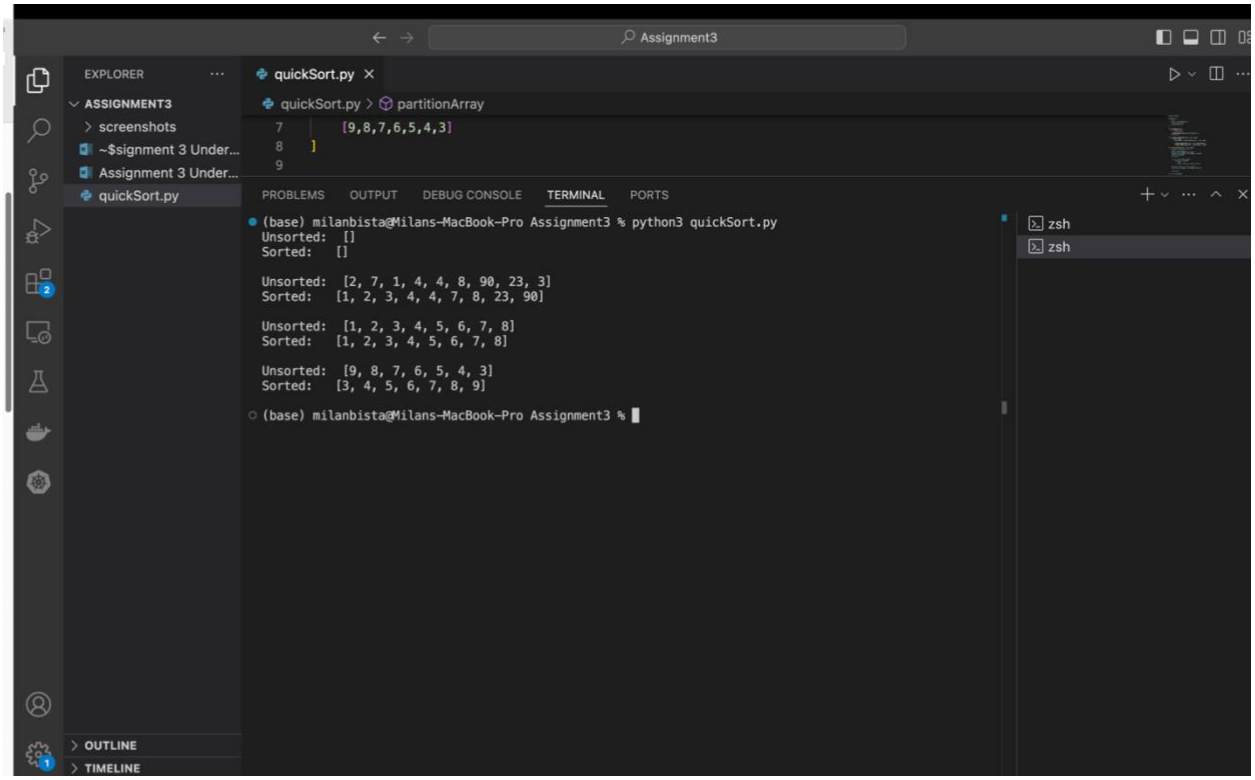
Instructors: Machica Mcclain / Vanessa Cooper

Part 1: Randomized Quicksort Analysis

1. Implementation

- I have implemented a randomized quicksort algorithm with different edge cases such as empty list, reversed, random etc.





2. Analysis

In Randomized Quicksort, the algorithm starts by picking a pivot element from the array at random. This pivot is used to partition the array into two parts: elements less than the pivot go to one side, and elements greater than the pivot go to the other. The algorithm then recursively applies this process to each of the two subarrays until the entire array is sorted.

The efficiency of Quicksort depends heavily on how well the pivot divides the array. If the pivot divides the array into roughly equal parts, each partitioning step reduces the problem size significantly, leading to fewer steps overall. Choosing the pivot randomly helps ensure that, on average, the divisions are well-balanced.

When we say the average-case complexity is O(nlogn), it means that on average, Randomized Quicksort needs about O(n)comparisons to process each level of recursion and about O(logn) levels of recursion.

Milan Bista

Quick Sort time Complexity = $O(n \log n)$

Each partitioning step requires $O(n)$ work to examine each element to place it either the left or right subarray. After partitioning, the resulting subarray are half the original size

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

where,

$T(n)$ is the total time needed to sort an array of size n

$2T\left(\frac{n}{2}\right)$ represents the time needed to sort the two subarrays, each approx. size $\frac{n}{2}$

$O(n)$ is the partiiong step that divides the array around pivot

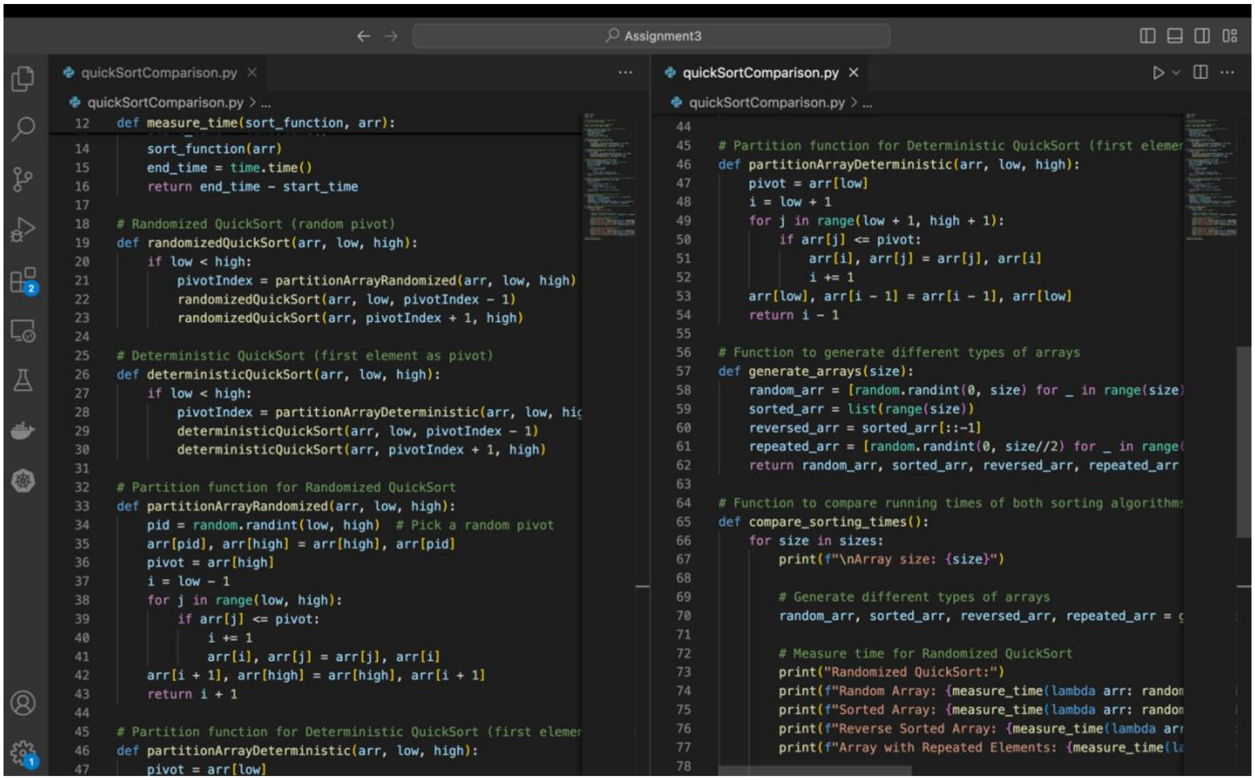
Total time = number of levels $\times$ time per level

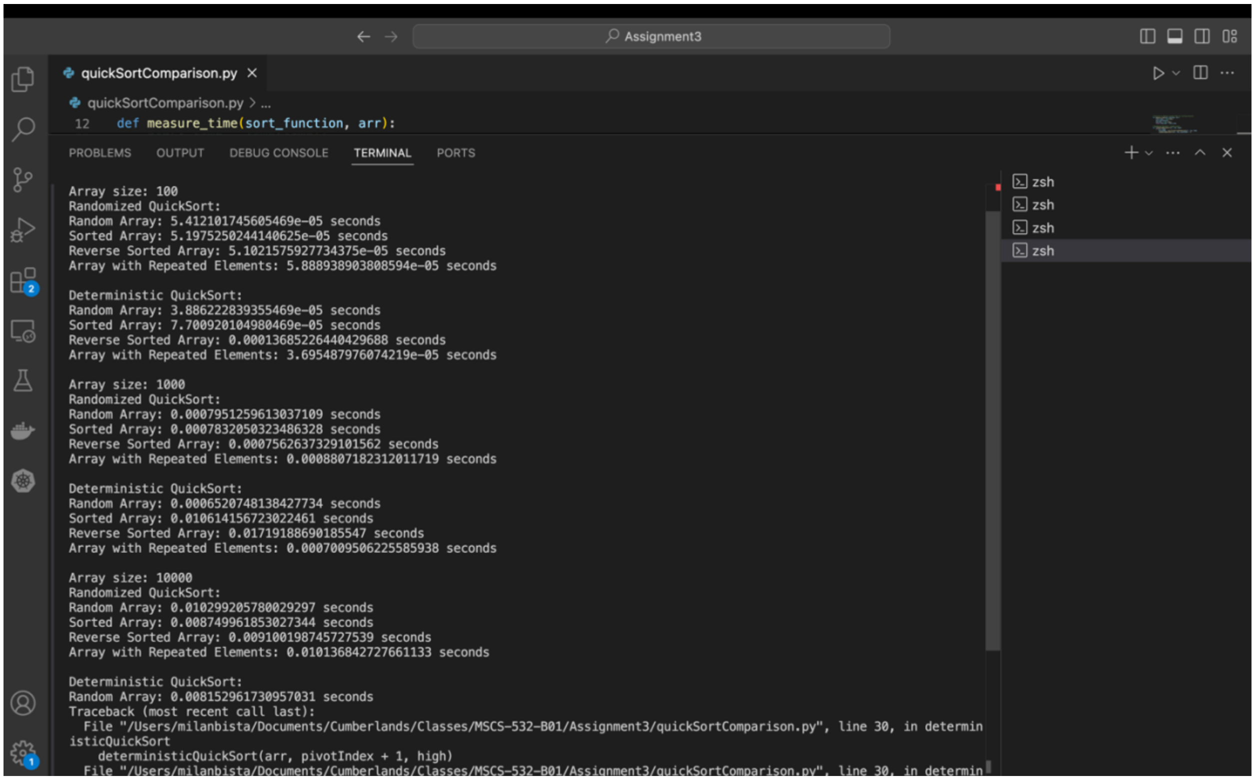
$$= O(\log n) \times O(n)$$
$$= O(n \log n)$$

where $\log n$ is the time per each subarrys

3. Comparison

- Empirically comparing the running time of Randomized Quicksort with Deterministic Quicksort (using the first element as the pivot) on different input sizes and distributions:





This study empirically compares **Randomized Quicksort** and **Deterministic Quicksort** (using the first element as the pivot) across various input sizes and data distributions: randomly generated arrays, already sorted arrays, reverse-sorted arrays, and arrays with repeated elements.

We tested both algorithms on input sizes of 100, 1000, 10000, and 100000 elements, generating arrays for each distribution. Running times were measured to compare the performance of both algorithms under different conditions.

Randomized Quicksort consistently performed near its expected O(nlogn) time complexity for all distributions. Its random pivot selection ensures balanced partitions, leading to efficient performance in all cases.

On the other hand, **Deterministic Quicksort** showed O(nlogn) performance for random and repeated element arrays but degraded to O(n^2) on sorted and reverse-sorted arrays. The use of the first element as the pivot caused unbalanced partitions, which led to inefficient sorting in these cases.

The main difference between the algorithms is how they handle unbalanced partitions. Randomized Quicksort avoids this issue by selecting a random pivot, which ensures more balanced partitions. In contrast, deterministic quicksort's use of the first element as the pivot causes poor performance on sorted and reverse-sorted arrays due to unbalanced partitions.

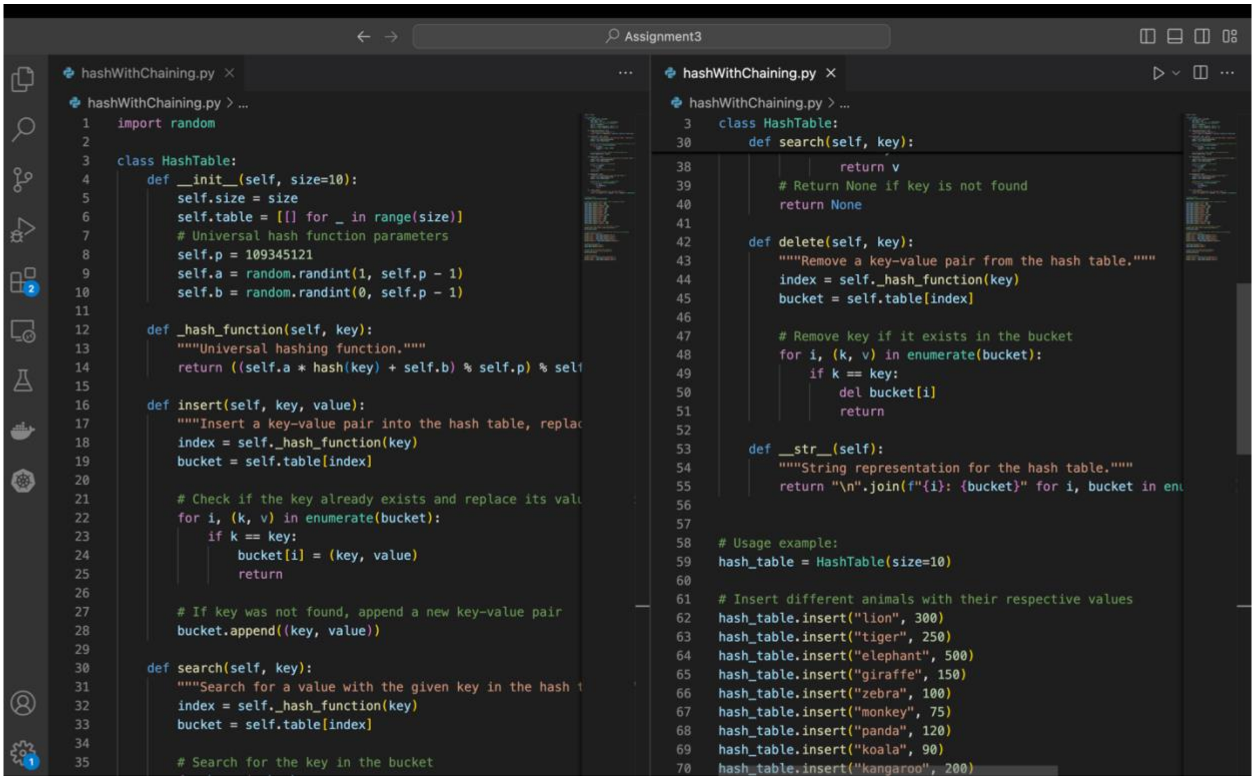
Both algorithms performed similarly on arrays with repeated elements, as the partitions remained balanced. However, Randomized Quicksort still had a slight advantage due to its random pivot selection, which helped avoid worst-case scenarios.

The results confirm the theoretical expectations: **Randomized Quicksort** maintains efficient O(nlogn) performance across all distributions, while **Deterministic Quicksort** performs poorly on sorted and reverse-sorted arrays, resulting in O(n2)O(n^2)O(n2) time complexity. Randomized Quicksort is more resilient to different input distributions, particularly when handling sorted or reverse-sorted data.

Part 2: Hashing with Chaining

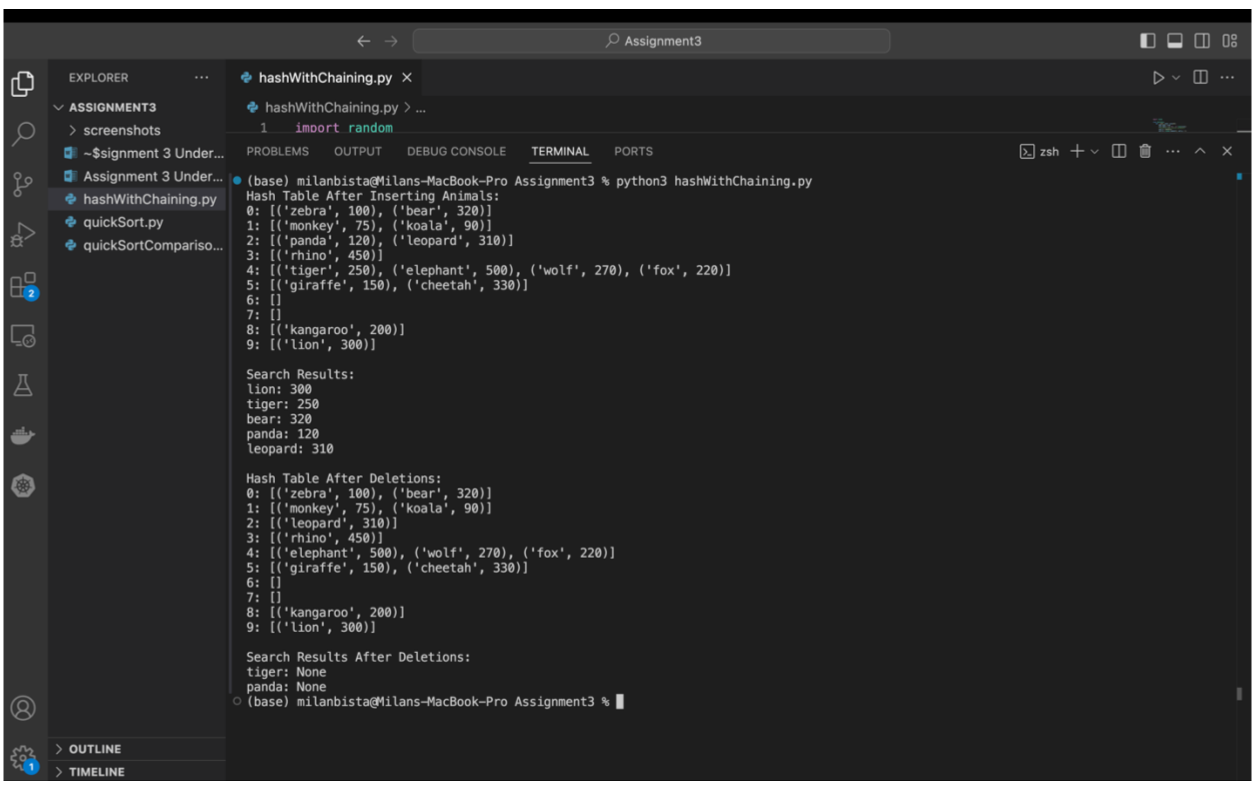
1. Implementation

- In the part, I am implementing a hash table using chaining for collision resolution. I am using a suitable hash function to minimize collisions. The function will support insert, delete and search along with to String method to print the content of the hash table.



```python
import random

class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
        # Universal hash function parameters
        self.p = 109345121
        self.a = random.randint(1, self.p - 1)
        self.b = random.randint(0, self.p - 1)

    def _hash_function(self, key):
        """Universal hashing function."""
        return ((self.a * hash(key) + self.b) % self.p) % self

    def insert(self, key, value):
        """Insert a key-value pair into the hash table, replac
        index = self._hash_function(key)
        bucket = self.table[index]

        # Check if the key already exists and replace its valu
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return

        # If key was not found, append a new key-value pair
        bucket.append((key, value))

    def search(self, key):
        """Search for a value with the given key in the hash t
        index = self._hash_function(key)
        bucket = self.table[index]

        # Search for the key in the bucket
```

```python
class HashTable:
    def search(self, key):
                return v
        # Return None if key is not found
        return None

    def delete(self, key):
        """Remove a key-value pair from the hash table."""
        index = self._hash_function(key)
        bucket = self.table[index]

        # Remove key if it exists in the bucket
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return

    def __str__(self):
        """String representation for the hash table."""
        return "\n".join(f"{i}: {bucket}" for i, bucket in enu


# Usage example:
hash_table = HashTable(size=10)

# Insert different animals with their respective values
hash_table.insert("lion", 300)
hash_table.insert("tiger", 250)
hash_table.insert("elephant", 500)
hash_table.insert("giraffe", 150)
hash_table.insert("zebra", 100)
hash_table.insert("monkey", 75)
hash_table.insert("panda", 120)
hash_table.insert("koala", 90)
hash_table.insert("kangaroo", 200)
```



```
(base) milanbista@Milans-MacBook-Pro Assignment3 % python3 hashWithChaining.py
Hash Table After Inserting Animals:
0: [('zebra', 100), ('bear', 320)]
1: [('monkey', 75), ('koala', 90)]
2: [('panda', 120), ('leopard', 310)]
3: [('rhino', 450)]
4: [('tiger', 250), ('elephant', 500), ('wolf', 270), ('fox', 220)]
5: [('giraffe', 150), ('cheetah', 330)]
6: []
7: []
8: [('kangaroo', 200)]
9: [('lion', 300)]

Search Results:
lion: 300
tiger: 250
bear: 320
panda: 120
leopard: 310

Hash Table After Deletions:
0: [('zebra', 100), ('bear', 320)]
1: [('monkey', 75), ('koala', 90)]
2: [('leopard', 310)]
3: [('rhino', 450)]
4: [('elephant', 500), ('wolf', 270), ('fox', 220)]
5: [('giraffe', 150), ('cheetah', 330)]
6: []
7: []
8: [('kangaroo', 200)]
9: [('lion', 300)]

Search Results After Deletions:
tiger: None
panda: None
(base) milanbista@Milans-MacBook-Pro Assignment3 %
```

2. Analysis

In a hash table that assumes simple uniform hashing, the expected time complexity for search, insert, and delete operations is heavily influenced by the load factor $\alpha = n/m$, where n is the number of elements and m is the number of slots (or buckets). When the load factor is low, meaning there are fewer elements relative to the number of buckets, each bucket will contain only a small number of elements, resulting in O(1) time complexity for these operations. This ensures efficient performance.

However, as the load factor increases (i.e., when the table becomes more populated), the performance degrades. When α\alpha becomes large, multiple elements hash to the same bucket, resulting in longer chains (if using chaining) or clusters (if using open addressing). This leads to an increase in the time taken for operations, with the expected time complexity for search, insert, and delete becoming proportional to the number of elements in a bucket, which in the worst case can be O(n).

Impact of Load Factor and Collision Resolution

The **load factor** plays a crucial role in determining the efficiency of hash table operations. A low load factor generally results in a small number of collisions, meaning the hash table operates close to its expected O(1) time for each operation. As the load factor increases, however, the number of collisions also increases, causing the performance to deteriorate due to the longer chains or clusters formed in the hash table.

In practice, if the load factor exceeds a threshold (often around 0.7), performance can degrade significantly, especially with chaining, as longer linked lists form in each bucket. This results in slower operations, potentially reaching O(n) in the worst case, where all elements collide into the same bucket.

Strategies for Optimizing Hash Table Performance

To avoid performance degradation as the load factor increases, **dynamic resizing** is typically employed. When the load factor exceeds a predetermined threshold, the hash table is resized (usually doubled), and all elements are rehashed into the new table. This reduces the load factor and ensures that the number of collisions is minimized. Resizing helps keep the time complexity of operations close to O (1).

Additionally, using a **good hash function** is critical to evenly distribute the keys across the hash table's slots. A well-designed hash function minimizes the likelihood of collisions and helps maintain efficient operations, even as the load factor increases.

In summary, maintaining a low load factor and dynamically resizing the hash table as it grows are key strategies to ensure that the hash table remains efficient, with time complexities for search, insert, and delete operations staying close to O(1). These strategies, combined with a good hash function, minimize collisions and optimize the hash table's performance.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.