Analysis of Randomized Quicksort and Hashing with Chaining

Introduction

In computer science, efficient data handling and organization are critical for the performance of software systems. Sorting and hashing are two fundamental techniques widely used to manage and retrieve data efficiently. This assignment focuses on analyzing and comparing the performance of **Randomized Quicksort** and **Hashing with Chaining**, two essential algorithms in the realms of sorting and hashing.

Randomized Quicksort is a variation of the classic Quicksort algorithm, where the pivot is chosen randomly from the subarray being partitioned. This random selection helps to avoid the worst-case scenario that can occur in Quicksort when the pivot is consistently chosen poorly, resulting in unbalanced partitions. The average-case time complexity of Randomized Quicksort is O(nlogn), making it an efficient choice for a wide range of datasets. In this assignment, we will implement Randomized Quicksort, analyze its time complexity using recurrence relations, and empirically compare its performance against Deterministic Quicksort, where the first element is consistently used as the pivot.

Hashing with Chaining is a method used to handle collisions in hash tables, where multiple keys may hash to the same index. This approach uses a linked list at each index in the hash table to store multiple key-value pairs that hash to the same slot. The efficiency of operations in a hash table with chaining depends on the **load factor**—the ratio of elements to available slots—and an effective hash function. In this assignment, we will implement a hash table with chaining, analyze its expected time complexity for operations, and discuss strategies to maintain efficient performance even as the load factor increases.

Through this document, we aim to develop a deeper understanding of the performance and scalability of sorting and hashing algorithms. The findings will demonstrate how Randomized Quicksort and Hashing with Chaining handle different input cases and provide insights into the conditions under which each algorithm is most efficient.

Part 1: Randomized Quicksort Analysis

Section 1: Implementation of Randomized Quicksort

Code and Explanation

Here's the implementation for **Randomized Quicksort**, where we choose the pivot randomly from the subarray to prevent consistently poor partitions. This helps maintain the average-case time complexity of

O(nlogn).

Randomized Quicksort Analysis

Section 1: Implementation of Randomized Quicksort

Code and Explanation

Here's the implementation for **Randomized Quicksort**, where we choose the pivot randomly from the subarray to prevent consistently poor partitions. This helps maintain the average-case time complexity of O(nlogn).

```
import random



def randomized_quicksort(arr, low, high):

  if low < high:

    # Partition the array and get the pivot index

    pivot_index = randomized_partition(arr, low, high)



    # Recursively sort elements before and after the partition

    randomized_quicksort(arr, low, pivot_index - 1)

    randomized_quicksort(arr, pivot_index + 1, high)



def randomized_partition(arr, low, high):

  # Step 1: Choose a random pivot and swap it with the last element

  pivot_index = random.randint(low, high)

  arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
```

```
    # Step 2: Standard partitioning around the pivot
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Explanation of Code

1. **randomized_quicksort function**:
    - This function performs the main sorting operation. It divides the array recursively by calling randomized_partition to get the index of the pivot.
    - The array is then split into two parts: elements left of the pivot and elements right of the pivot. The function recursively sorts each part until the base case is reached (when low >= high).
2. **randomized_partition function**:
    - **Pivot Selection**: A random pivot is chosen by selecting a random index between low and high, and this pivot element is swapped with the element at high to place it at the end.
    - **Partitioning Logic**: After setting the pivot, the function iterates through the array to partition it based on the pivot. Elements less than the pivot move to the left, and greater elements stay on the right. Finally, the pivot is swapped into its correct position in the sorted array.

Section 2: Testing and Edge Cases

To verify the robustness of the Randomized Quicksort implementation, we'll test it on a variety of cases, including edge cases. Here's a summary of test cases and their expected behaviors:

1. **Empty Array** ([]): The function should recognize that no sorting is needed and return the empty array without error.
2. **Array with Repeated Elements** ([3, 5, 3, 3, 7, 5]): Randomized Quicksort should handle duplicates properly, keeping elements in sorted order.
3. **Already Sorted Array** ([1, 2, 3, 4, 5]): Randomization helps prevent poor performance on already sorted data.
4. **Reverse-Sorted Array** ([5, 4, 3, 2, 1]): Randomization should help avoid the(n^2) worst-case complexity that might occur without it.

Here's how to test the function with edge cases:

```
from RandomizedQuickSort import randomized_quicksort




import time

import random

import sys



# Increase recursion limit

sys.setrecursionlimit(5000)  # Adjust this value as needed based on array size



# Define a function to measure execution time of randomized quicksort

def measure_sort_time(arr):

  start_time = time.time()

  randomized_quicksort(arr, 0, len(arr) - 1)

  end_time = time.time()

  return end_time - start_time



# Randomized Quicksort functions (same as before)
```

```
def randomized_quicksort(arr, low, high):
    if low < high:
        pivot_index = randomized_partition(arr, low, high)
        randomized_quicksort(arr, low, pivot_index - 1)
        randomized_quicksort(arr, pivot_index + 1, high)


def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


# Test arrays with different cases
array_sizes = [1000, 5000, 10000]  # Testing with different input sizes


# 1. Empty array test
empty_array = []
time_empty = measure_sort_time(empty_array)
```

```
print(f"Empty array sorted in: {time_empty:.6f} seconds")



# Loop through other test arrays with increasing sizes

for size in array_sizes:

  # 2. Randomly generated array

  random_array = [random.randint(1, 10000) for _ in range(size)]

  time_random = measure_sort_time(random_array)

  print(f"Random array of size {size} sorted in: {time_random:.6f} seconds")



  # 3. Already sorted array

  sorted_array = list(range(1, size + 1))

  time_sorted = measure_sort_time(sorted_array)

  print(f"Sorted array of size {size} sorted in: {time_sorted:.6f} seconds")



  # 4. Reverse-sorted array

  reverse_sorted_array = list(range(size, 0, -1))

  time_reverse_sorted = measure_sort_time(reverse_sorted_array)

  print(f"Reverse sorted array of size {size} sorted in: {time_reverse_sorted:.6f} seconds")



  # 5. Array with repeated elements

  repeated_elements_array = [random.choice([1, 2, 3, 4, 5]) for _ in range(size)]

  time_repeated = measure_sort_time(repeated_elements_array)

  print(f"Array with repeated elements of size {size} sorted in: {time_repeated:.6f} seconds")
```

```
print("-" * 40)
```

Python3 TestRadomizedQuickSort.py

Empty array sorted in: 0.000001 seconds

Random array of size 1000 sorted in: 0.001356 seconds

Sorted array of size 1000 sorted in: 0.001315 seconds

Reverse sorted array of size 1000 sorted in: 0.001140 seconds

Array with repeated elements of size 1000 sorted in: 0.004848 seconds

---

Random array of size 5000 sorted in: 0.006831 seconds

Sorted array of size 5000 sorted in: 0.006079 seconds

Reverse sorted array of size 5000 sorted in: 0.005670 seconds

Array with repeated elements of size 5000 sorted in: 0.066649 seconds

---

Random array of size 10000 sorted in: 0.011609 seconds

Sorted array of size 10000 sorted in: 0.010886 seconds

Reverse sorted array of size 10000 sorted in: 0.010783 seconds

Array with repeated elements of size 10000 sorted in: 0.251815 seconds

---

Time Complexity Analysis of Randomized Quicksort

Objective: To demonstrate that Randomized Quicksort has an average-case time complexity of O(nlogn).

1. How Randomized Quicksort Works

Randomized Quicksort achieves its efficiency by selecting a **random pivot** for each partitioning step. This randomness helps balance the partitions on average, preventing the consistently poor performance seen with deterministic pivot choices (like choosing the first element).

The algorithm proceeds as follows:

1. **Divide Step**: It picks a random pivot, partitions the array around it, and creates two subarrays—one with elements less than the pivot and the other with elements greater than the pivot.
2. **Recursive Sort**: It recursively sorts the left and right subarrays.
3. **Combine Step**: Once the subarrays are sorted, they combine with the pivot to form the fully sorted array.

Each **partitioning step** has an O(n) cost, as it involves scanning the array once to place elements in the appropriate subarray.

2. Deriving the Average-Case Complexity Using Recurrence Relation

To analyze the average-case complexity, we use the recurrence relation. Let:

- T(n) is the time complexity for sorting an array of size n.
- Ideally, each partition splits the array into two equal parts, so we assume, on average, the pivot divides the array into two halves of size n/2.

The recurrence relation for the average case can then be expressed as

$$T(n) = 2T(2n) + O(n)$$

- $2T(2n)$: Recursively sorting the two halves.
- O(n): The partitioning step, which requires O(n) time to scan the array.

Using this recurrence relation and solving it, we expand T(n) as follows:

1. Substitute T(n/2) in terms of T(n/4):

   $$T(n) = 2(2T(4n) + 2n) + n = 4T(4n) + 2n$$

2. Continuing this process for kkk steps, we get:

   $$T(n) = 2kT(2kn) + k{\cdot}n$$

3. When $k = logn$, $T(n/2^k) = T(1)$, giving us:

   $$T(n) = nlogn$$

Thus, the **average-case time complexity** of Randomized Quicksort is O(nlogn).

3. Using Indicator Random Variables to Support the O(nlogn)O(n \log n)O(nlogn) Complexity

Let's approach the analysis with **indicator random variables** to support the expected O(nlogn) complexity.

Define an indicator variable Xij for each pair (i, j) of elements, where:

- Xij=1 if elements i and j are compared during the algorithm's execution.
- Xij=0 otherwise.

The total number of comparisons made by the algorithm, X, is the sum of these indicator variables:

The total number of comparisons made by the algorithm, XXX, is the sum of these indicator variables:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Xij$$

Each comparison E[Xij] is performed with a probability 2/(j−i+1) (the probability that either element i or j is chosen as the pivot before any element between them).

The expected number of comparisons, E[X], is then:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/\text{j-i+1}$$

This sum evaluates to O(nlogn), confirming that the average number of comparisons made is consistent with our recurrence-based analysis.

4. Interpreting the Empirical Results

Your empirical results further support the theoretical O(n logn) complexity. Here's how:

- **Random Arrays**: For sizes 1000, 5000, and 10000, the times are roughly proportional to n logn.
- **Sorted and Reverse-Sorted Arrays**: These cases, which can perform poorly in deterministic Quicksort, exhibit similar times to random arrays, demonstrating that randomized pivot selection effectively avoids worst-case scenarios.
- **Arrays with Repeated Elements**: These take longer as duplicates add to the comparisons, but still fall within the O(nlogn) range.

For instance, the time for a random array of size 1000 is around 0.0014 seconds, while for size 10000, it's about 0.0109 seconds, roughly consistent with an O(nlogn) growth rate.

Summary

- **Average-Case Complexity**: The average-case time complexity of Randomized Quicksort is O(nlogn), derived using both recurrence relations and expected comparisons.

- **Empirical Verification**: The timing results align with O(nlogn), especially for random and sorted inputs.
- **Effectiveness of Randomization**: The random pivot selection helps avoid worst-case behavior, keeping the performance consistent across various input types.

Comparison with Deterministic Quicksort

Objective

- **Empirical Comparison**: We aim to measure and compare the execution times of Randomized Quicksort and Deterministic Quicksort across various input scenarios. This comparison will highlight how the choice of a random pivot affects performance consistency, especially on sorted and reverse-sorted arrays.

Testing Setup

Here's a recap of the input cases we'll be using:

1. **Empty Array**: []
2. **Random Array**: An array of random integers within a range.
3. **Already Sorted Array**: An array sorted in ascending order.
4. **Reverse-Sorted Array**: An array sorted in descending order.
5. **Array with Repeated Elements**: An array containing many duplicate values.

Comparison Code

This code performs sorting using both Randomized Quicksort and Deterministic Quicksort and prints their execution times for each input scenario. We'll leverage the same array for each test to ensure fair comparison.

```
from RandomizedQuickSort import randomized_quicksort




import time

import random

import sys



# Increase recursion limit

sys.setrecursionlimit(5000)  # Adjust this value as needed based on array size
```

```
# Define a function to measure execution time of randomized quicksort
def measure_sort_time(arr):
    start_time = time.time()
    randomized_quicksort(arr, 0, len(arr) - 1)
    end_time = time.time()
    return end_time - start_time

# Randomized Quicksort functions (same as before)
def randomized_quicksort(arr, low, high):
    if low < high:
        pivot_index = randomized_partition(arr, low, high)
        randomized_quicksort(arr, low, pivot_index - 1)
        randomized_quicksort(arr, pivot_index + 1, high)

def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
```

```
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# Test arrays with different cases
array_sizes = [1000, 5000, 10000]  # Testing with different input sizes

# 1. Empty array test
empty_array = []
time_empty = measure_sort_time(empty_array)
print(f"Empty array sorted in: {time_empty:.6f} seconds")

# Loop through other test arrays with increasing sizes
for size in array_sizes:
    # 2. Randomly generated array
    random_array = [random.randint(1, 10000) for _ in range(size)]
    time_random = measure_sort_time(random_array)
    print(f"Random array of size {size} sorted in: {time_random:.6f} seconds")

    # 3. Already sorted array
    sorted_array = list(range(1, size + 1))
    time_sorted = measure_sort_time(sorted_array)
    print(f"Sorted array of size {size} sorted in: {time_sorted:.6f} seconds")

    # 4. Reverse-sorted array
```

```
reverse_sorted_array = list(range(size, 0, -1))

time_reverse_sorted = measure_sort_time(reverse_sorted_array)

print(f"Reverse sorted array of size {size} sorted in: {time_reverse_sorted:.6f} seconds")



# 5. Array with repeated elements

repeated_elements_array = [random.choice([1, 2, 3, 4, 5]) for _ in range(size)]

time_repeated = measure_sort_time(repeated_elements_array)

print(f"Array with repeated elements of size {size} sorted in: {time_repeated:.6f} seconds")



print("-" * 40)
```

Output:
Empty array sorted in (Randomized): 0.000001 seconds

Empty array sorted in (Deterministic): 0.000001 seconds

Random array of size 1000 sorted in (Randomized): 0.001057 seconds

Random array of size 1000 sorted in (Deterministic): 0.000780 seconds

Sorted array of size 1000 sorted in (Randomized): 0.000985 seconds

Sorted array of size 1000 sorted in (Deterministic): 0.014761 seconds

Reverse sorted array of size 1000 sorted in (Randomized): 0.000861 seconds

Reverse sorted array of size 1000 sorted in (Deterministic): 0.022120 seconds

Array with repeated elements of size 1000 sorted in (Randomized): 0.003232 seconds

Array with repeated elements of size 1000 sorted in (Deterministic): 0.002756 seconds

---

Random array of size 5000 sorted in (Randomized): 0.005032 seconds

Random array of size 5000 sorted in (Deterministic): 0.004070 seconds

Sorted array of size 5000 sorted in (Randomized): 0.004865 seconds

Sorted array of size 5000 sorted in (Deterministic): 0.310081 seconds

Reverse sorted array of size 5000 sorted in (Randomized): 0.004574 seconds

Reverse sorted array of size 5000 sorted in (Deterministic): 0.537417 seconds

Array with repeated elements of size 5000 sorted in (Randomized): 0.065865 seconds

Array with repeated elements of size 5000 sorted in (Deterministic): 0.062036 seconds

---

Random array of size 10000 sorted in (Randomized): 0.011351 seconds

Random array of size 10000 sorted in (Deterministic): 0.009318 seconds

Sorted array of size 10000 sorted in (Randomized): 0.010933 seconds

Sorted array of size 10000 sorted in (Deterministic): 1.239562 seconds

Reverse sorted array of size 10000 sorted in (Randomized): 0.010456 seconds

Reverse sorted array of size 10000 sorted in (Deterministic): 2.145808 seconds

Array with repeated elements of size 10000 sorted in (Randomized): 0.254597 seconds

Array with repeated elements of size 10000 sorted in (Deterministic): 0.248305 seconds

1. **Empty Array**:
    - Both Randomized and Deterministic Quicksort perform similarly, taking negligible time, as expected. An empty array requires no comparisons or swaps, so it completes immediately.
2. **Random Array**:
    - For random arrays of various sizes, **Deterministic Quicksort** generally performs slightly faster than **Randomized Quicksort**. This is likely due to the lack of overhead from random pivot selection, as the inherent randomness in the array already provides a balanced partition on average.
    - Both algorithms demonstrate O(nlogn) scaling as the input size grows, consistent with theoretical expectations for average-case performance.
3. **Already Sorted Array**:

    - Here, the advantage of **Randomized Quicksort** becomes evident. For sorted arrays, Randomized Quicksort maintains near O(nlogn) performance by effectively balancing partitions with random pivot selection.
    - **Deterministic Quicksort**, on the other hand, performs poorly with sorted arrays. When the first element is used as the pivot on an already sorted array, it leads to extremely unbalanced partitions, resulting in (n^2) time complexity. This is apparent in the significantly higher times as array size increases (e.g., 1.24 seconds for size 10,000).
4. **Reverse-Sorted Array**:
    - Similar to the sorted array case, **Randomized Quicksort** efficiently handles reverse-sorted arrays due to its balanced partitioning, maintaining O(nlogn) performance.
    - **Deterministic Quicksort**, however, struggles significantly with reverse-sorted data, taking even longer than on sorted data due to the unbalanced partitions created by using the first element as the pivot. For instance, at size 10,000, Deterministic Quicksort takes over 2 seconds compared to Randomized Quicksort's 0.01 seconds.
5. **Array with Repeated Elements**:
    - For arrays with many repeated elements, both algorithms demonstrate similar performance. Although the presence of duplicates increases the number of comparisons in each partition, both algorithms maintain approximately O(nlogn) performance.
    - Both Randomized and Deterministic Quicksort show slightly longer times with increasing array size but do not differ significantly from each other in this case. For instance, at size 10,000, both Randomized and Deterministic Quicksort perform comparably (0.254 seconds vs. 0.248 seconds).

Summary of Findings

1. **Effectiveness of Randomized Quicksort**:
    - Randomized Quicksort consistently achieves O(nlogn)O(n \log n)O(nlogn) performance across all input types, including sorted and reverse-sorted arrays. By randomizing pivot selection, it avoids the pitfalls of unbalanced partitions, providing efficient and stable performance across diverse data distributions.
    - The overhead of random pivot selection is minimal and generally outweighed by the benefits of more balanced partitioning, particularly in worst-case scenarios.
2. **Limitations of Deterministic Quicksort**:
    - Deterministic Quicksort performs well on random and repeated-element arrays but performs poorly on already sorted or reverse-sorted data due to the unbalanced partitions created by choosing the first element as the pivot.
    - This vulnerability to sorted data highlights the importance of pivot selection in Quicksort. In cases where input data is likely to be sorted, Randomized Quicksort is preferable to avoid O(n^2) performance.
3. **Practical Recommendation**:

- For general-purpose sorting where input data characteristics are unknown, Randomized Quicksort is a better choice due to its consistency and resistance to worst-case scenarios.
- Deterministic Quicksort can be suitable for random or highly shuffled data where the choice of a fixed pivot does not lead to unbalanced partitions.

Part 2: Hashing with Chaining

Section 1: Implementation of Hash Table with Chaining

In this section, we'll implement a **hash table** that uses **chaining** to resolve collisions. In chaining, each slot in the hash table contains a list (or linked list) that holds all key-value pairs that hash to the same slot.

Code Implementation

Here's the Python implementation of the hash table with chaining, supporting insert, search, and delete operations:

```
class HashTable:

  def __init__(self, size=10):

      self.size = size  # Size of the hash table

      self.table = [[] for _ in range(self.size)]  # Initialize table with empty lists



  def hash_function(self, key):

      """Hash function that maps keys to table indices."""

      return hash(key) % self.size



  def insert(self, key, value):

      """Insert a key-value pair into the hash table."""

      index = self.hash_function(key)

      # Check if the key already exists and update the value if it does

      for item in self.table[index]:
```

```
        if item[0] == key:
            item[1] = value  # Update existing value
            return
    # If the key is not found, append it to the chain
    self.table[index].append([key, value])

def search(self, key):
    """Search for a value associated with the given key."""
    index = self.hash_function(key)
    for item in self.table[index]:
        if item[0] == key:
            return item[1]  # Return the associated value
    return None  # Key not found

def delete(self, key):
    """Delete a key-value pair from the hash table."""
    index = self.hash_function(key)
    for i, item in enumerate(self.table[index]):
        if item[0] == key:
            del self.table[index][i]  # Remove the item from the chain
            return True
    return False  # Key not found
```

Explanation of Code

1. **Initialization**:
    - The hash table is initialized with a specific size, and each slot contains an empty list to store multiple items in case of collisions.
2. **Hash Function**:
    - hash_function calculates the index for each key by taking hash(key) % self.size, ensuring that the key maps to a valid index within the table size.
3. **Insert Operation**:
    - insert finds the correct slot using the hash function and checks if the key already exists in the chain. If it does, it updates the value; otherwise, it adds the new key-value pair to the list at that slot.
4. **Search Operation**:
    - search finds the appropriate slot using the hash function and searches the list for the specified key, returning the value if found.
5. **Delete Operation**:
    - delete locates the key and removes it from the chain if found. It returns True if successful, False otherwise.

Section 2: Testing the Hash Table with Edge Cases

Let's test the hash table on some typical cases:

1. **Insert and Search**: Add multiple key-value pairs and search for them to ensure they were inserted correctly.
2. **Delete Operation**: Remove keys and check if they were deleted successfully.
3. **Collision Handling**: Insert keys that hash to the same slot (use a small table size to simulate this).

Here's the testing code:

```
# Create a hash table

hash_table = HashTable(size=5)  # Small size to force collisions



# Insert items

hash_table.insert("apple", 1)

hash_table.insert("banana", 2)

hash_table.insert("cherry", 3)
```

```
hash_table.insert("banana", 4)  # Update value for "banana"

hash_table.insert("grape", 5)

hash_table.insert("mango", 6)  # Collision likely with small table size



# Search for items

print("Search 'banana':", hash_table.search("banana"))  # Should print updated value 4

print("Search 'cherry':", hash_table.search("cherry"))  # Should print 3

print("Search 'grape':", hash_table.search("grape"))    # Should print 5

print("Search 'nonexistent':", hash_table.search("nonexistent"))  # Should print None



# Delete items

print("Delete 'apple':", hash_table.delete("apple"))    # Should return True

print("Delete 'nonexistent':", hash_table.delete("nonexistent"))  # Should return False

print("Search 'apple' after deletion:", hash_table.search("apple"))  # Should print None
```

Output

Search 'banana': 4

Search 'cherry': 3

Search 'grape': 5

Search 'nonexistent': None

Delete 'apple': True

Delete 'nonexistent': False

Search 'apple' after deletion: None

This code will confirm that:

- Keys can be inserted, searched, and deleted correctly.
- Updates to existing keys are handled.
- Collisions are managed effectively by chaining.

Section 3: Time Complexity Analysis

The performance of hash table operations depends on the **load factor** α , defined as the ratio of the number of elements n to the number of slots mmm:

$$\alpha = n/m$$

1. **Insertion, Search, and Deletion**:
    - In an ideal case (simple uniform hashing), each slot in the hash table has an expected number of entries proportional to α\alphaα.
    - For each operation (insert, search, delete), the expected time complexity is O(1+α), as it requires computing the hash function and scanning a linked list proportional to α\alphaα in the worst case.
2. **Effect of Load Factor**:
    - When α is low, each slot has few elements, and operations remain efficient.
    - As α increases (more elements relative to slots), each slot's chain becomes longer, degrading performance.
    - A high load factor leads to longer chains, increasing the time for each operation to O(α) in the worst case. To avoid this, the hash table can be resized when α exceeds a threshold.
3. **Resizing the Hash Table**:
    - One strategy to keep α\alphaα low is **dynamic resizing** (e.g., doubling the table size and rehashing all elements) when α\alphaα exceeds a set threshold. Resizing maintains O(1) average-case complexity for each operation, even as the dataset grows.

Summary of Findings for Hashing with Chaining

- **Efficiency**: Hashing with chaining provides average-case $O(1)$ operations for insert, search, and delete under simple uniform hashing, making it efficient for lookups.
- **Impact of Load Factor**: The load factor directly influences performance, with higher values resulting in longer chains. Dynamic resizing helps maintain efficiency by keeping α low.
- **Collision Resolution**: Chaining is an effective method for handling collisions, as each slot can hold multiple items, avoiding data overwrites.

Overall Findings

The experiments conducted provide valuable insights into the efficiency and scalability of Randomized Quicksort and Hashing with Chaining. While Randomized Quicksort excels at maintaining balanced partitions for sorting tasks, Hashing with Chaining efficiently resolves collisions in data storage and retrieval, offering constant-time operations under a manageable load factor.

Final Recommendations:

- **Randomized Quicksort** is ideal for sorting applications with unknown input distributions, thanks to its resistance to worst-case performance.
- **Hashing with Chaining** is well-suited for data storage systems requiring rapid insertions and lookups, especially when managing a high volume of data with potential collisions.

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.

Goodrich, M. T., & Tamassia, R. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Mitzenmacher, M., & Upfal, E. (2005). *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press.