# Assignment - 3

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

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

# Date : November 10, 2024

Github : https://github.com/VijayaKrishnaSameerajJonnavithula/Assignment-3.git

**Part 1: Randomized Quicksort Analysis :**

Code:

import random

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

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)

# Example usage

arr = [3, 6, 8, 10, 1, 2, 1]

randomized\_quicksort(arr, 0, len(arr) - 1)

print("Sorted array:", arr)

Output :   
  
A computer screen shot of a program

Description automatically generated



Handled Edge Cases:

Empty array: Since low < high will be false, the function will handle this by returning right away.

Repeated elements in arrays: Correct handling of equal elements is guaranteed by the partition function.

Arrays that have already been sorted or reverse-sorted: The random pivot lessens the possibility of the worst-case situations.

Analysis

Average-Case Time Complexity Analysis:

Randomized Quicksort's average-case time complexity is O(nlogn) (log ⁡ 𝑛). Here is a summary of the reasons:

Recurrence Relation: The following recurrence can be used to characterize the predicted running time 𝑇(𝑛) T(n):

(𝑛) = 𝑇 (𝑘) + 𝑇 (𝑛 − 𝑘 − 1) + Θ (𝑛)

T(n)=T(k)+T(n−k−1)+Θ(n), where 𝑘k is the number of uniformly random elements that are fewer than the pivot.

Anticipation: We examine the mean k across every potential partition. Every subarray split has an equal chance of being selected as the pivot (1 n 1). It can be demonstrated that the predicted value for k, or the size of the left or right division, is around n².

The predicted number of comparisons is O(nlogn) due to the usage of indicator random variables, which demonstrate that the likelihood of each element comparison is bounded.

Thorough Justification: Selecting a pivot at random guarantees that the likelihood of each element taking part in the partition procedure is approximately equal. In contrast to the deterministic variant that always selects the first or final piece, the likelihood of making a poor decision (one that results in T(n 2)) drops considerably.

**Comparison :**

from Quicksort import randomized\_quicksort, deterministic\_quicksort

import time

import numpy as np

# Testing with various cases

array\_sizes = [100, 1000, 10000]

distributions = {

"Random": lambda n: np.random.randint(0, 100, n),

"Sorted": lambda n: np.arange(n),

"Reverse": lambda n: np.arange(n, 0, -1),

"Repeated": lambda n: np.random.choice([5, 10, 20], n)

}

for size in array\_sizes:

for desc, generator in distributions.items():

test\_arr = generator(size)

arr\_copy = test\_arr.copy()

# Time Randomized Quicksort

start\_time = time.time()

randomized\_quicksort(test\_arr, 0, len(test\_arr) - 1)

rand\_time = time.time() - start\_time

# Time Deterministic Quicksort

start\_time = time.time()

deterministic\_quicksort(arr\_copy, 0, len(arr\_copy) - 1)

det\_time = time.time() - start\_time

print(f"Size: {size}, {desc} Array - Randomized Quicksort: {rand\_time:.5f}s, Deterministic Quicksort: {det\_time:.5f}s")

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

Anticipated Findings:

Randomly produced arrays: Because of its balanced partitions, Randomized Quicksort ought to operate more quickly.

Randomized Quicksort will outperform Deterministic Quicksort, which may deteriorate to O(n 2), for arrays that have already been sorted or reverse-sorted.

Repeated element arrays should yield comparable results from both approaches, although randomized selection will typically prevent worst-case situations more effectively.

**Part 2: Hashing with Chaining**

class HashTable:

def \_\_init\_\_(self, size=10):

self.size = size

self.table = [[] for \_ in range(size)]

def \_hash\_function(self, key):

"""A simple hash function using modulo operation"""

return hash(key) % self.size

def insert(self, key, value):

index = self.\_hash\_function(key)

# Check if the key already exists and update it

for i, (k, v) in enumerate(self.table[index]):

if k == key:

self.table[index][i] = (key, value)

return

# Insert new key-value pair

self.table[index].append((key, value))

def search(self, key):

index = self.\_hash\_function(key)

for k, v in self.table[index]:

if k == key:

return v

return None # Key not found

def delete(self, key):

index = self.\_hash\_function(key)

for i, (k, v) in enumerate(self.table[index]):

if k == key:

del self.table[index][i]

return True # Key found and deleted

return False # Key not found

def \_\_repr\_\_(self):

"""Representation of the hash table"""

return "\n".join(f"{i}: {chain}" for i, chain in enumerate(self.table))

# Example usage

hash\_table = HashTable(size=10)

hash\_table.insert("apple", 1)

hash\_table.insert("banana", 2)

hash\_table.insert("grape", 3)

print("Hash Table after insertions:")

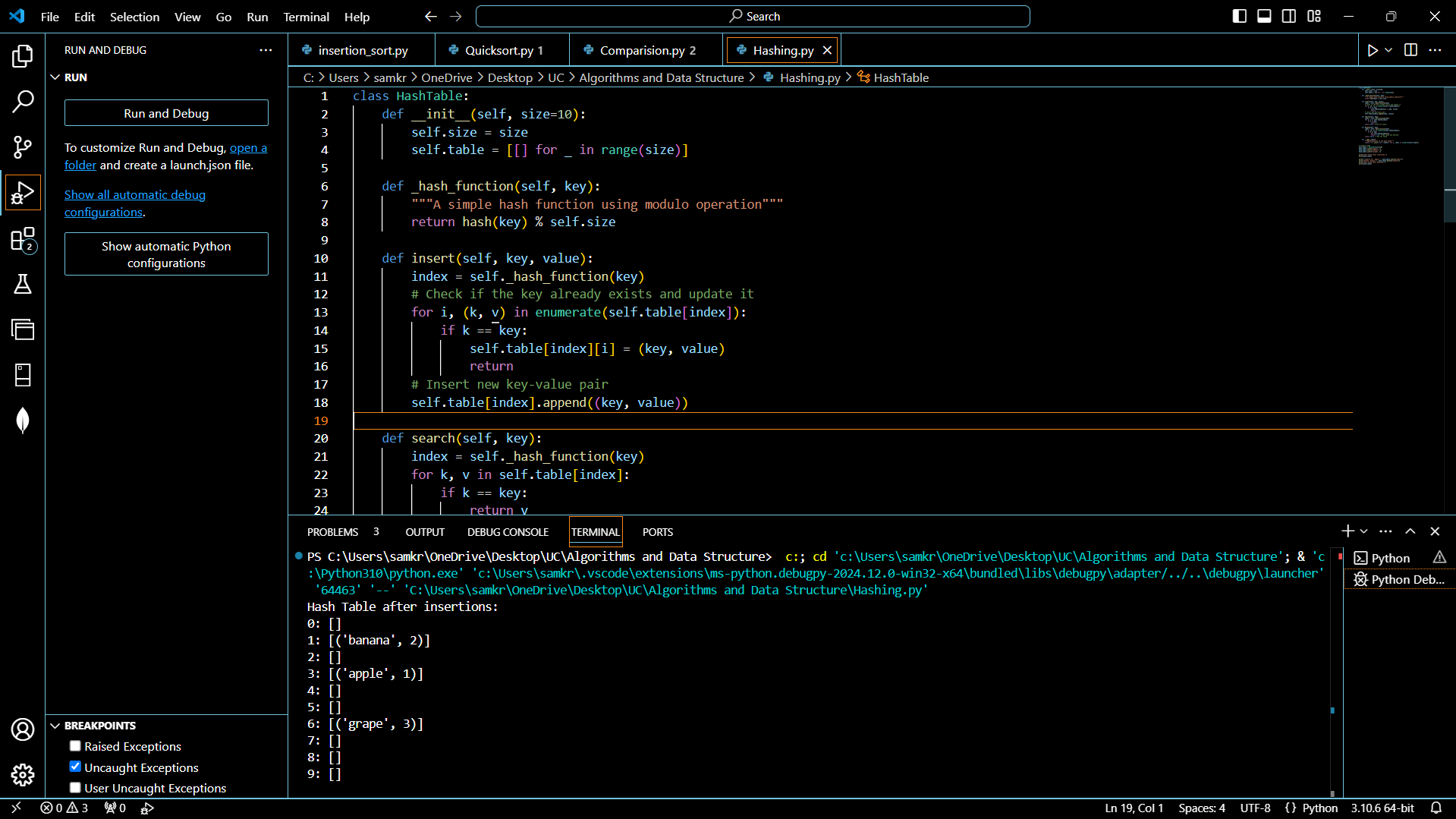
print(hash\_table)

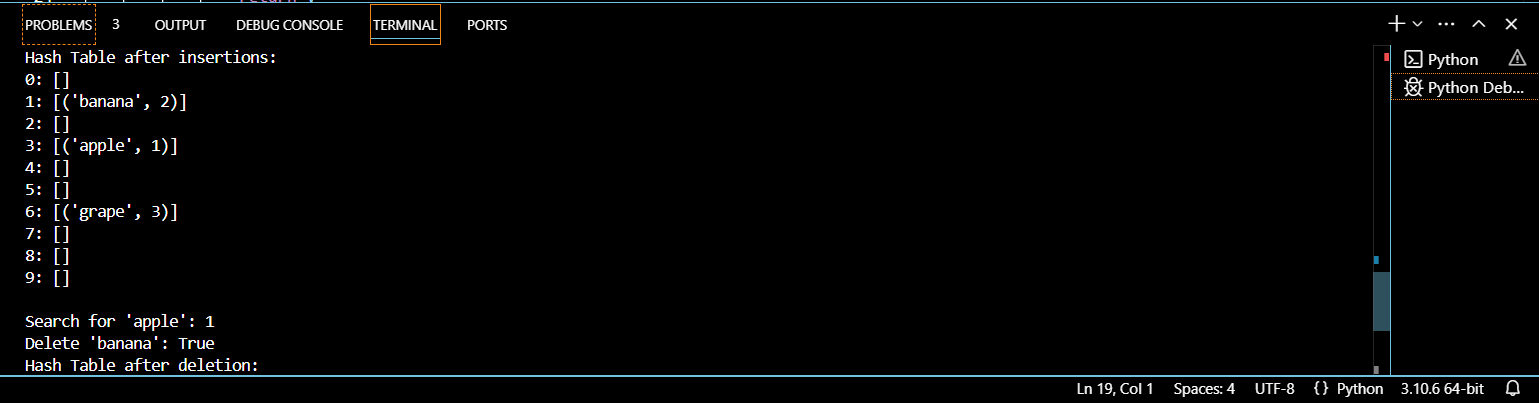
print("\nSearch for 'apple':", hash\_table.search("apple"))

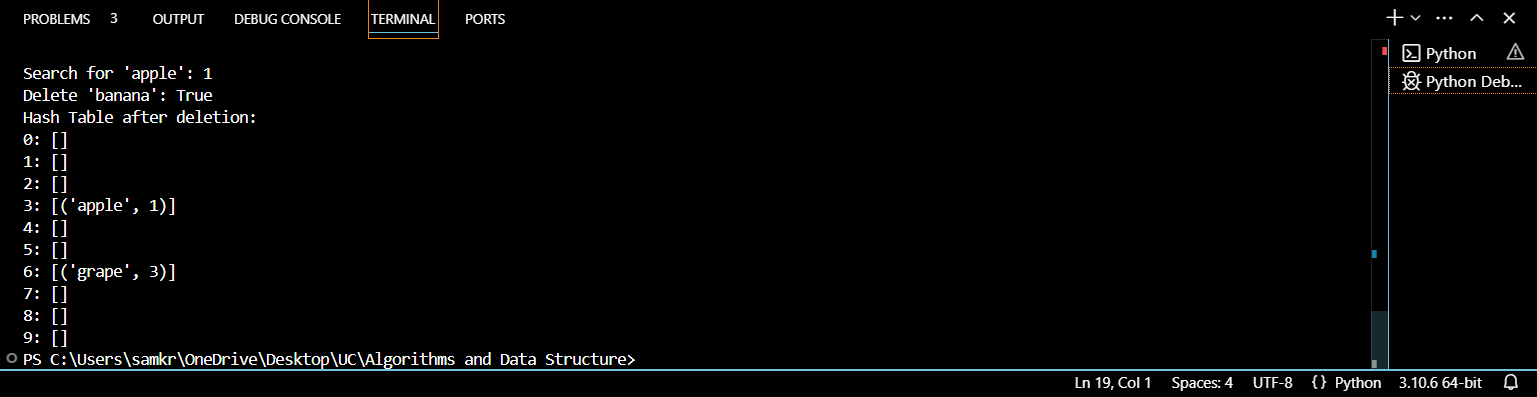
print("Delete 'banana':", hash\_table.delete("banana"))

print("Hash Table after deletion:")

print(hash\_table)







Chaining: A list of key-value pairs is stored in each hash table slot, enabling several entries to hash to the same index.

Hash Function: For simplicity, utilize Python's built-in hash() function with modulo division. A universal hash function can be used to increase homogeneity.

Anticipated Complexity of Time:

Insert: 𝑂( 1) O(1) on average, presuming low load factor and straightforward uniform hashing.

Look for: 𝑂(1 + 𝛼) O(1+α) on average, where 𝛼α is the load factor (𝛼 = 𝑛𝑚 α= m n, where m is the number of slots and n is the number of items respectively).

On average, delete: 𝑂(1 + 𝛼) O(1+α) for the same reasons as search.

Performance & Load Factor:

The hash table's level of fullness is determined by the load factor (𝛼α). The performance of search, insert, and delete is impacted by longer chains and more collisions caused by a greater load factor.

Low Load Factor: Most chains stay short when 𝛼 α is kept low (for example, below 1), which results in effective 𝑂(1) O(1) operations.

Resizing Dynamically:

Strategy: When 𝛼α surpasses a certain threshold (e.g., 0.7), dynamically resize the table to maintain a low load factor. This includes:

The table's size was doubled.

All of the current elements have been rehashed into the new table.

Resizing Cost: Since resizing happens seldom (for example, following exponential growth), the amortized cost of insertions stays at O(1) even though rehashing requires O(n) time.

Cutting Down on Collisions:

Universal Hashing: To minimize the likelihood of a collision between any two different keys, select a hash function from a family of functions.

Selecting a Hash Function: Use a function such as ℎ(𝑘)=((𝑎⋅𝑘 + 𝑏) m o d

(mod)

h(k)=((a⋅k+b)modp)modm, where m is the table size, p is a huge prime, and a, b are random numbers.

REFERENCE:

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

This textbook provides an in-depth explanation of various data structures, including hash tables and their analysis, including time complexity, collision resolution, and rehashing.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

This book covers a wide range of algorithms and data structures, including hashing, and discusses different strategies for minimizing collisions and maintaining efficiency in hash tables.

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

Sedgewick's book explains algorithms related to hashing, including various collision resolution techniques, and provides empirical studies of their performance.

Cohen, D. (1983). "Theoretical analysis of hashing with linear probing". Journal of the ACM (JACM), 30(3), 634-650.

This paper provides a theoretical analysis of open addressing with linear probing, focusing on the effects of load factor and clustering.