ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY
Department of Computer Science

**NAME:** MUHAMMAD HASEEB **CLASS:** BSCS
**ROLL NO:** 14861 **Sir:** JAMAL ABDUAL AHUD
**SUBJECT:** DSA **DATE:** 01-02-2025

# PROJECT PROPOSAL

# DSA
# HASH TABLE IMPLEMENTATION
# GitHub link

## 1. Introduction

### 1.1 Purpose

The purpose of this document is to define the requirements for the implementation of a **Hash Table** using a **Linked List (Chaining Method) in Python**. This implementation provides efficient insertion, deletion, and lookup operations for managing key-value pairs.

### 1.2 Document Conventions

- Code is written in **Python 3.x**.
- Comments are used to explain functionalities.
- CamelCase is used for class names, and snake case is used for function and variable names.

### 1.3 Project Scope

#### 1.3.1 Scope Definition

This project aims to implement a **Hash Table** data structure with the following features:

- **Chaining Method:** Handles collisions using linked lists.
- **Dynamic Resizing:** Automatically expands when the load factor exceeds **0.7**.
- **CRUD Operations:** Supports **Insertion, Search, and Deletion** of key-value pairs.

#### 1.3.2 Core Features

- **Custom Hash Function** using modulo operation.
- **Load Factor Monitoring** to determine when to resize the table.
- **Linked List-based Chaining** to handle collisions.

#### 1.3.3 Future Enhancements

- Implement **Open Addressing** as an alternative to chaining.
- Add **Graphical User Interface (GUI)** for visualization.

### 1.4 References

- **Data Structures and Algorithms in Python** – Michael T. Goodrich
- Python Documentation: https://docs.python.org/3/

## 2. Overall Description

### 2.1 Product Perspective

The Hash Table implementation is an **in-memory** data structure used for quick lookup operations. It serves as a foundational component for databases, caches, and other applications requiring fast data retrieval.

#### 2.1.1 Product Context

This implementation is a standalone module that can be integrated into larger projects requiring **efficient key-value storage**.

### 2.2 User Classes and Characteristics

- **Developers:** Use it as a module in applications.
- **Students:** Learn hash tables and chaining in Python.

### 2.3 Operating Environment

- **Programming Language:** Python 3.x
- **Hardware:** Works on standard machines (1GB RAM, 1GHz CPU)
- **Operating System:** Windows/Linux/Mac

### 2.4 Design and Implementation Constraints

- Uses **linked lists** for collision handling.
- Load factor threshold: **0.7** (resizes when exceeded).

## 3. System Features

### 3.1 Key Functionalities

- **Insert (key, value):** Adds key-value pairs.
- **Search(key):** Retrieves value for a given key.
- **Delete(key):** Removes a key-value pair.
- **Display ():** Prints the entire hash table.

### 4. External Interface Requirements

### 4.1 User Interface

- **Command-Line Interface (CLI)** for input/output.

### 4.2 Software Interfaces

- Python 3 Standard Library

### 5. Quality Attributes

### 5.1 Performance

- **Average Time Complexity:** O(1) for search, insert, and delete.
- **Worst Case Complexity:** O(n) (when all keys hash to the same bucket).

### 5.2 Reliability

- Ensures accurate **insertion, retrieval, and deletion**.

### 5.3 Usability

- Simple API with easy-to-use functions.

### 5.4 Security

- **Handles collisions** to prevent data loss.

## CODE:

```
class Node:
    """Node for storing key-value pairs in a linked list (for chaining)."""
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

```
class HashTable:
    """Custom Hash Table with Chaining and Dynamic Resizing."""
    def __init__(self, size=10):
        self.size = size
        self.table = [None] * self.size
        self.count = 0  # Track number of elements
        self.load_factor_threshold = 0.7  # Resize if exceeded

    def _hash(self, key):
        """Hash function using modulo operation."""
        return hash(key) % self.size

    def _resize(self):
        """Doubles the table size and rehashes all elements."""
        new_size = self.size * 2
        new_table = [None] * new_size
        old_table = self.table
        self.table = new_table
        self.size = new_size
        self.count = 0  # Reset count and reinsert

        for node in old_table:
            while node:
                self.insert(node.key, node.value)
                node = node.next

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table."""
        index = self._hash(key)
        node = self.table[index]
        while node:
            if node.key == key:
                node.value = value  # Update existing key
                return
            node = node.next

        # Insert new node at the head (chaining)
        new_node = Node(key, value)
        new_node.next = self.table[index]
        self.table[index] = new_node
        self.count += 1

        # Check if resizing is needed
        if self.count / self.size > self.load_factor_threshold:
            self._resize()

    def search(self, key):
        """Searches for a key in the hash table."""
        index = self._hash(key)
        node = self.table[index]

        while node:
            if node.key == key:
                return node.value
            node = node.next

        return None  # Key not found

    def delete(self, key):
        """Deletes a key-value pair from the hash table."""
        index = self._hash(key)
        node = self.table[index]
        prev = None

        while node:
            if node.key == key:
                if prev:
                    prev.next = node.next
                else:
                    self.table[index] = node.next
                self.count -= 1
                return True  # Key deleted
            prev = node
            node = node.next

        return False  # Key not found

    def display(self):
        """Displays the hash table contents."""
        for i, node in enumerate(self.table):
            print(f"Index {i}: ", end="")
            while node:
                print(f"({node.key}: {node.value}) -> ", end="")
                node = node.next
            print("None")

# Example Usage
ht = HashTable()
ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("orange", 300)
```

ht.insert("grape", 400)

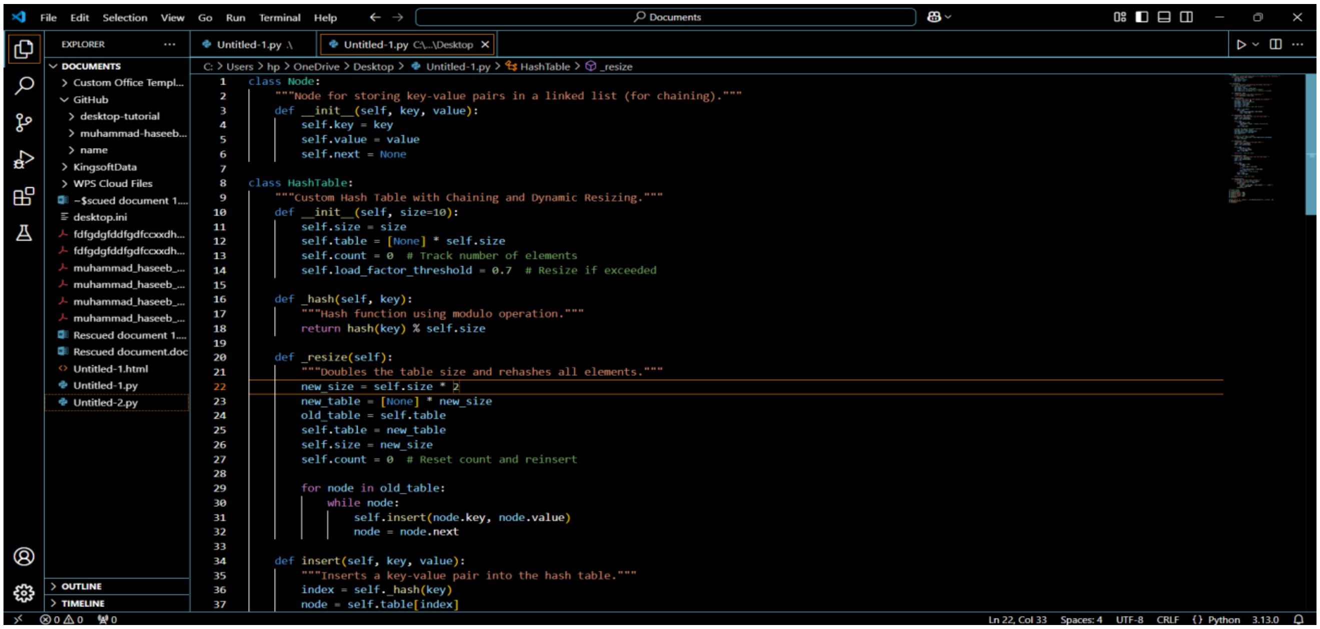
ht.display()

print("Search for 'banana':", ht.search("banana"))  # Output: 200

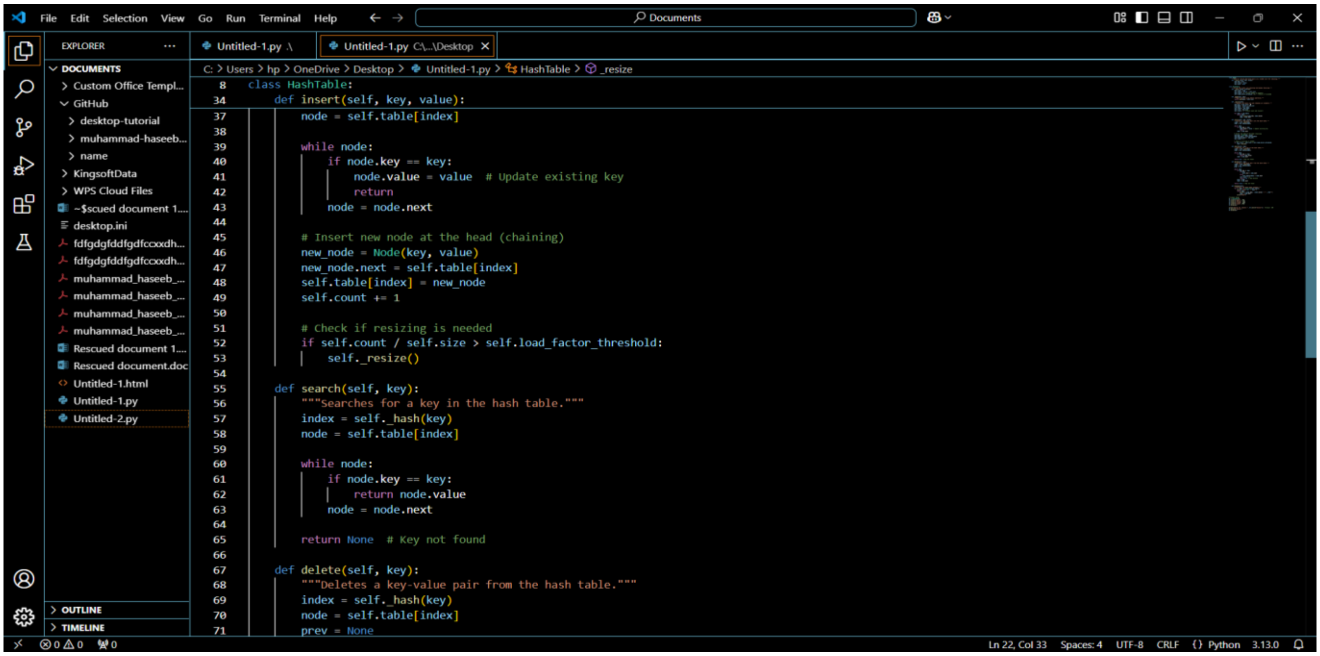
ht.delete("banana")

ht.display()

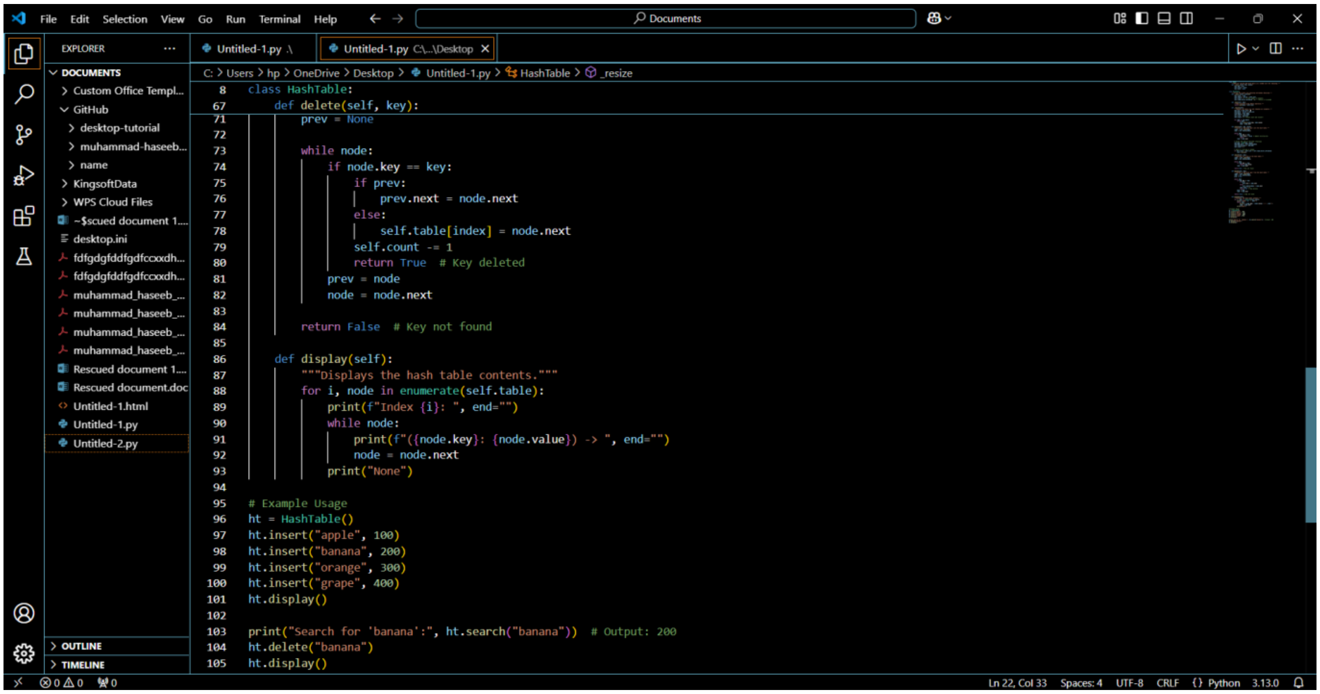
## CODE PICTURE:



```python
class Node:
    """Node for storing key-value pairs in a linked list (for chaining)."""
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None

class HashTable:
    """Custom Hash Table with Chaining and Dynamic Resizing."""
    def __init__(self, size=10):
        self.size = size
        self.table = [None] * self.size
        self.count = 0  # Track number of elements
        self.load_factor_threshold = 0.7  # Resize if exceeded

    def _hash(self, key):
        """Hash function using modulo operation."""
        return hash(key) % self.size

    def _resize(self):
        """Doubles the table size and rehashes all elements."""
        new_size = self.size * 2
        new_table = [None] * new_size
        old_table = self.table
        self.table = new_table
        self.size = new_size
        self.count = 0  # Reset count and reinsert

        for node in old_table:
            while node:
                self.insert(node.key, node.value)
                node = node.next

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table."""
        index = self._hash(key)
        node = self.table[index]
```

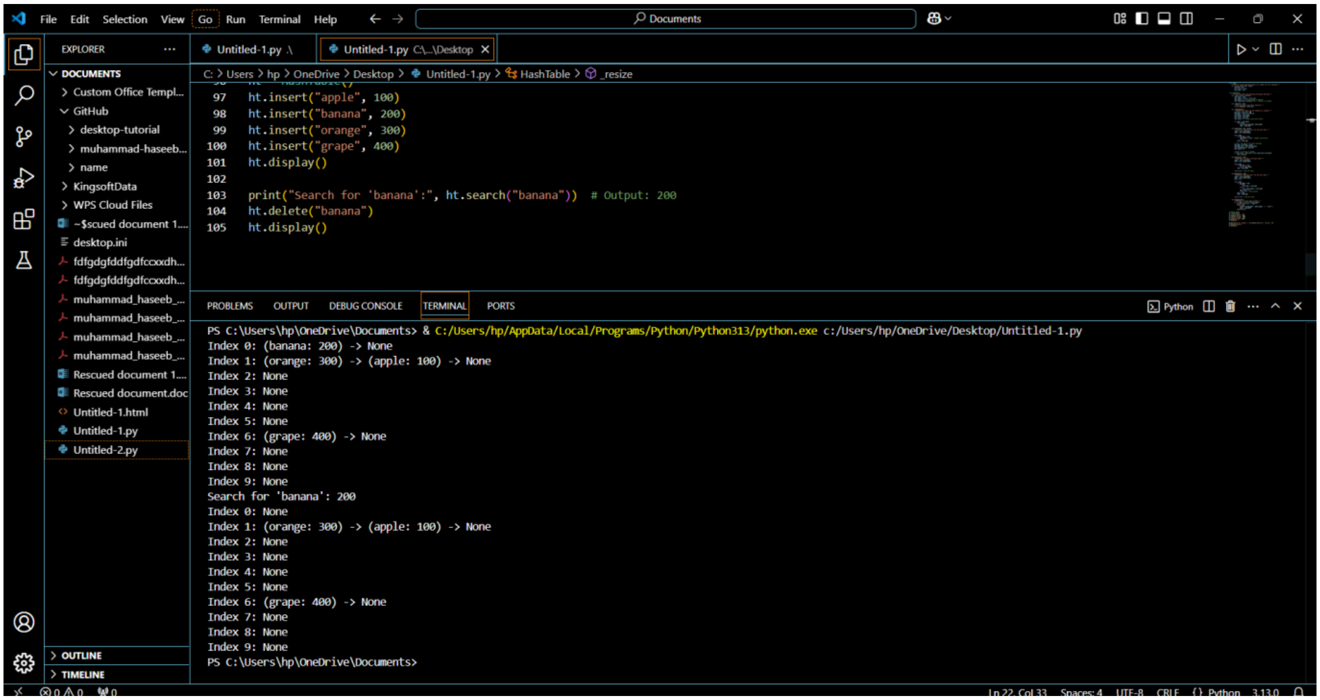


```python
class HashTable:
    def insert(self, key, value):
        node = self.table[index]

        while node:
            if node.key == key:
                node.value = value  # Update existing key
                return
            node = node.next

        # Insert new node at the head (chaining)
        new_node = Node(key, value)
        new_node.next = self.table[index]
        self.table[index] = new_node
        self.count += 1

        # Check if resizing is needed
        if self.count / self.size > self.load_factor_threshold:
            self._resize()

    def search(self, key):
        """Searches for a key in the hash table."""
        index = self._hash(key)
        node = self.table[index]

        while node:
            if node.key == key:
                return node.value
            node = node.next

        return None  # Key not found

    def delete(self, key):
        """Deletes a key-value pair from the hash table."""
        index = self._hash(key)
        node = self.table[index]
        prev = None
```


File Edit Selection View Go Run Terminal Help
Documents
EXPLORER
Untitled-1.py .\
Untitled-1.py C:\...\Desktop
C: > Users > hp > OneDrive > Desktop > Untitled-1.py > HashTable > _resize
Ln 22, Col 33 Spaces: 4 UTF-8 CRLF {} Python 3.13.0

```python
class HashTable:
    def delete(self, key):
        prev = None

        while node:
            if node.key == key:
                if prev:
                    prev.next = node.next
                else:
                    self.table[index] = node.next
                self.count -= 1
                return True  # Key deleted
            prev = node
            node = node.next

        return False  # Key not found

    def display(self):
        """Displays the hash table contents."""
        for i, node in enumerate(self.table):
            print(f"Index {i}: ", end="")
            while node:
                print(f"({node.key}: {node.value}) -> ", end="")
                node = node.next
            print("None")

# Example Usage
ht = HashTable()
ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("orange", 300)
ht.insert("grape", 400)
ht.display()

print("Search for 'banana':", ht.search("banana"))  # Output: 200
ht.delete("banana")
ht.display()
```

## OUT PUT:



```python
ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("orange", 300)
ht.insert("grape", 400)
ht.display()

print("Search for 'banana':", ht.search("banana"))  # Output: 200
ht.delete("banana")
ht.display()
```

```
PS C:\Users\hp\OneDrive\Documents> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe c:/Users/hp/OneDrive/Desktop/Untitled-1.py
Index 0: (banana: 200) -> None
Index 1: (orange: 300) -> (apple: 100) -> None
Index 2: None
Index 3: None
Index 4: None
Index 5: None
Index 6: (grape: 400) -> None
Index 7: None
Index 8: None
Index 9: None
Search for 'banana': 200
Index 0: None
Index 1: (orange: 300) -> (apple: 100) -> None
Index 2: None
Index 3: None
Index 4: None
Index 5: None
Index 6: (grape: 400) -> None
Index 7: None
Index 8: None
Index 9: None
PS C:\Users\hp\OneDrive\Documents>
```