

Code Explanation and Help



i have viva and practical exam
if i provide you my code then can you help me by explaining the code line by line,also what is use of the code and how it works
and if possible can you give video to understand the concept in code

Yes, I can absolutely help you with that.

✓ If you provide your code (in Python, C++, or any other language you're using), I will:

1. **Explain it line by line** in simple language.
2. Describe **what the code is doing** (its purpose and how it works).
3. Mention where it might be used in real-world or academic situations.
4. If needed, point out **mistakes or improvements**.
5. ✓ **Include video links** (YouTube or others) that explain the concept in the code (like AVL trees, hashing, file handling, etc.).

Please upload or paste your code here to get started.



```
class HashTable1:
    """linear Probing without replacement"""

    def __init__(self, size: int) -> None:
        self.record = []
        self.m = size

        for _ in range(size):
            self.record.append([0, ""])

    def display_table(self) -> None:
        print("Hash table using linear probing (without replacement):\n")
        for i in range(len(self.record)):
            print(i, self.record[i])

    def hash_function(self, tel: int) -> int:
        key = (tel % self.m)
        return key

    def generate_table(self, recs: list[list]) -> None:
        for rec in recs:
            self.insert_rec(rec)
```

```
def insert_rec(self, rec: list) -> None:
    key = self.hash_function(rec[0])
    if (self.record[key][0] == 0):

        self.record[key][0] = rec[0]
        self.record[key][1] = rec[1]
    else:
        while (self.record[key][0] != 0):
            key = ((key+1) % self.m)

        self.record[key][0] = rec[0]
        self.record[key][1] = rec[1]

class HashTable2:
    """linear Probing with replacement"""

    def __init__(self, size: int) -> None:
        self.record = []
        self.m = size

        for _ in range(size):
            self.record.append([0, "", -1])

    def display_table(self) -> None:
        print("Hash table using linear probing (with replacement):\n")
        for i in range(len(self.record)):
            print(i, self.record[i])

    def hash_function(self, tel: int) -> int:
        key = (tel % self.m)
        return key

    def generate_table(self, recs: list[list]) -> None:
        for rec in recs:
            self.insert_rec(rec)

    def insert_rec(self, rec: list) -> None:
        key = self.hash_function(rec[0])
        if (self.record[key][0] == 0):

            self.record[key][0] = rec[0]
            self.record[key][1] = rec[1]
            self.record[key][2] = -1
        else:
            if (self.hash_function(self.record[key][0]) == key):

                last_elmt = key
                while (self.record[last_elmt][2] != -1):
                    last_elmt = self.record[last_elmt][2]
```

```

k = last_elmt
while (self.record[k][0] != 0):
    k = ((k+1) % self.m)
self.record[last_elmt][2] = k
self.record[k][0] = rec[0]
self.record[k][1] = rec[1]
self.record[k][2] = -1
else:
    for i in range(self.m):
        if (self.record[i][2] == key):
            prev_link_key = i

    old_rec_tel = self.record[key][0]
    old_rec_name = self.record[key][1]
    old_rec_link = self.record[key][2]

    self.record[key][0] = rec[0]
    self.record[key][1] = rec[1]
    self.record[key][2] = -1

    k = key
    while (self.record[k][0] != 0):
        k = ((k+1) % self.m)

    self.record[prev_link_key][2] = k
    self.record[k][0] = old_rec_tel
    self.record[k][1] = old_rec_name
    self.record[k][2] = old_rec_link

```

```

class HashTable3:
    """Double hashing"""

    def __init__(self, size: int) -> None:
        self.record = []
        self.m = size

        for _ in range(size):
            self.record.append([0, ""])

        if (size <= 3):
            self.prime = size
        else:
            prime = [2, 3]
            for i in range(size):
                for j in prime:
                    if (i % j == 0):
                        p = False
                        break
                if (p):
                    prime.append(i)

```

```

        self.prime = prime[-1]

def hash1(self, key: int) -> int:
    return (key % self.m)

def hash2(self, key: int) -> int:
    return (self.prime - (key % self.prime))

def display_table(self) -> None:
    print("Hash table using double hashing:\t")
    for i in range(len(self.record)):
        print(i, self.record[i])

def generate_table(self, recs: list[list]) -> None:
    for rec in recs:
        self.insert_rec(rec)

def insert_rec(self, rec: list) -> None:
    i = 0
    key = self.hash1(rec[0])
    k2 = (key + i*self.hash2(rec[0])) % self.m
    while (self.record[k2][0] != 0):
        k2 = (key + i*self.hash2(rec[0])) % self.m
        i += 1
    self.record[k2][0] = rec[0]
    self.record[k2][1] = rec[1]

def input_records(n: int) -> list[list]:
    records = []
    for i in range(n):
        print(f"--- PERSON {i+1} ---")
        name = input("Name of the person:\t")
        tel = int(input("Telephone number:\t"))
        records.append([tel, name])
        print("--- DETAILS SAVED ---")
    return records

n = int(input("Total number of records are:\t"))
records = input_records(n)
ch = 1
while(ch != 5):
    print("--- MAIN MENU")
    print("1 -> Enter records")
    print("2 -> Use linear Probing (without replacement)")
    print("3 -> Use linear Probing (with replacement)")
    print("4 -> Use double hashing")
    print("5 -> Exit")

    ch = int(input("Choose an option (1-5):\t"))
    match (ch):

```

```

case 1:
    n = int(input("Total number of records:\t"))
    records = input_records(n)
case 2:
    t1 = HashTable1(n)
    t1.generate_table(records)
    t1.display_table()
case 3:
    t2 = HashTable2(n)
    t2.generate_table(records)
    t2.display_table()
case 4:
    t3 = HashTable3(n)
    t3.generate_table(records)
    t3.display_table()
case 5:
    print("\n## END OF CODE\n")
case default:
    print("Please choose a valid option (1-5).")

```

Explanation of the Code:

This code implements a hash table using **linear probing without replacement**, **linear probing with replacement**, and **double hashing** techniques. It is designed to handle collisions in hash tables and store telephone records for various people. Let's go over the code and explain it step by step.

1. `class HashTable1`: Linear Probing Without Replacement

Constructor (`__init__`)

```

python

def __init__(self, size: int) -> None:
    self.record = []
    self.m = size
    for _ in range(size):
        self.record.append([0, ""])

```

- **Purpose:** Initializes the hash table of a given size (`size`).
- **Explanation:**
 - `self.record`: List representing the hash table.
 - Each entry in the table is a list `[0, ""]`, where `0` represents an empty slot, and `""` is a placeholder for the name.

`hash_function`

```

python

def hash_function(self, tel: int) -> int:
    key = (tel % self.m)

```

```
return key
```

- **Purpose:** A simple hash function that calculates the index by taking the modulus of the telephone number (`tel`) with the table size (`m`).
- **Explanation:** This ensures that the index is within the range of the hash table.

``insert_rec``

python

```
def insert_rec(self, rec: list) -> None:
    key = self.hash_function(rec[0])
    if (self.record[key][0] == 0):
        self.record[key][0] = rec[0]
        self.record[key][1] = rec[1]
    else:
        while (self.record[key][0] != 0):
            key = ((key+1) % self.m)
        self.record[key][0] = rec[0]
        self.record[key][1] = rec[1]
```

- **Purpose:** Inserts a record (`rec`) containing a telephone number and name into the hash table.
- **Explanation:**
 - It first computes the hash index (`key`) using ``hash_function``.
 - If the calculated index is empty (i.e., ``self.record[key][0] == 0``), it places the record there.
 - If the index is occupied, it uses **linear probing** to find the next available slot by incrementing the index by 1 (and using modulus to ensure it wraps around).

``display_table``

python

```
def display_table(self) -> None:
    print("Hash table using linear probing (without replacement):\n")
    for i in range(len(self.record)):
        print(i, self.record[i])
```

- **Purpose:** Displays the entire hash table.
- **Explanation:** Iterates through the hash table and prints each index with its contents.

2. ``class HashTable2``: Linear Probing With Replacement

This class is similar to ``HashTable1``, but it allows **replacement** when a collision occurs.

Constructor (``__init__``)

python

```
def __init__(self, size: int) -> None:
    self.record = []
    self.m = size
```

```
for _ in range(size):
    self.record.append([0, "", -1])
```

- **Purpose:** Initializes the hash table with an additional slot for a "linked list" (`-1` initially).
- **Explanation:** The table is now initialized with three components in each record: `0` for an empty slot, `` for a name, and `-1` for the linked list link.

`insert_rec`

python

```
def insert_rec(self, rec: list) -> None:
    key = self.hash_function(rec[0])
    if (self.record[key][0] == 0):
        self.record[key][0] = rec[0]
        self.record[key][1] = rec[1]
        self.record[key][2] = -1
    else:
        if (self.hash_function(self.record[key][0]) == key):
            last_elmt = key
            while (self.record[last_elmt][2] != -1):
                last_elmt = self.record[last_elmt][2]
            k = last_elmt
            while (self.record[k][0] != 0):
                k = ((k+1) % self.m)
            self.record[last_elmt][2] = k
            self.record[k][0] = rec[0]
            self.record[k][1] = rec[1]
            self.record[k][2] = -1
        else:
            for i in range(self.m):
                if (self.record[i][2] == key):
                    prev_link_key = i
                    old_rec_tel = self.record[key][0]
                    old_rec_name = self.record[key][1]
                    old_rec_link = self.record[key][2]
                    self.record[key][0] = rec[0]
                    self.record[key][1] = rec[1]
                    self.record[key][2] = -1
                    k = key
                    while (self.record[k][0] != 0):
                        k = ((k+1) % self.m)
                    self.record[prev_link_key][2] = k
                    self.record[k][0] = old_rec_tel
                    self.record[k][1] = old_rec_name
                    self.record[k][2] = old_rec_link
```

- **Purpose:** Handles collision resolution using linear probing with replacement.
- **Explanation:**
 - When a slot is occupied, the function checks if the key of the existing record matches the calculated hash index (`key`).
 - If they match, it tries to find the end of the linked list for the collided key and then performs linear probing for the next empty slot.
 - It also handles "replacement" by shifting old records to their next available slots.

3. `class HashTable3`: Double Hashing

Constructor (`__init__`)

python

```
def __init__(self, size: int) -> None:
    self.record = []
    self.m = size
    for _ in range(size):
        self.record.append([0, ""])
    if (size <= 3):
        self.prime = size
    else:
        prime = [2, 3]
        for i in range(size):
            for j in prime:
                if (i % j == 0):
                    p = False
                    break
            if (p):
                prime.append(i)
        self.prime = prime[-1]
```

- **Purpose:** Initializes the hash table and calculates a **prime number** larger than the table size for the second hash function.
- **Explanation:** `self.prime` is used for the second hash function, ensuring that the table size is coprime with the hash values.

`hash1` and `hash2`

python

```
def hash1(self, key: int) -> int:
    return (key % self.m)

def hash2(self, key: int) -> int:
    return (self.prime - (key % self.prime))
```

- **Purpose:**
 - `hash1`: A simple hash function based on modulus.
 - `hash2`: A secondary hash function used for double hashing to reduce collisions further by providing a different offset.

`insert_rec`

python

```
def insert_rec(self, rec: list) -> None:
    i = 0
    key = self.hash1(rec[0])
    k2 = (key + i*self.hash2(rec[0])) % self.m
    while (self.record[k2][0] != 0):
        k2 = (key + i*self.hash2(rec[0])) % self.m
        i += 1
    self.record[k2][0] = rec[0]
    self.record[k2][1] = rec[1]
```

- **Purpose:** Handles insertion of records using **double hashing** to resolve collisions.
- **Explanation:**

- `hash1` calculates the primary index.
 - `hash2` calculates the secondary offset to avoid clustering.
 - If the calculated index is already occupied, the index is re-calculated using both hash functions, incrementing a counter `i` until an empty slot is found.
-

4. Input and Main Program

- The code accepts **input records** (name and telephone number), stores them in the hash table, and allows the user to choose between linear probing without replacement, linear probing with replacement, and double hashing for collision resolution.
 - The program will keep displaying the table until the user chooses to exit.
-

Concept Explanation (Video Links)

1. Hashing and Hash Tables:

- [Hashing Concepts \(GeeksforGeeks\)](#)

2. Linear Probing:

- [Linear Probing Explained](#)

3. Double Hashing:

- [Double Hashing Explained](#)

These videos will help you grasp the underlying concepts of hashing and the collision resolution techniques used in this code.