# Project 03 Hash Tables

We will share few of the essential functions required by students to complete in their project.

# _hash function

```python
def _hash(self, key: str, inserting: bool = False) -> int:
    """Calculates the hash index for a given key to determine its position in the hash table...."""
    # region
    # Calculate the first hash value
    hash_1 = self._hash_1(key)
    # Calculate the second hash value
    hash_2 = self._hash_2(key)
    # Initial index using the first hash value
    index = hash_1
    # Initialize the probe count, used for double hashing
    probe = 0
    # Attempt to retrieve the node at the calculated index
    node = self.table[index]

    # Loop until an appropriate slot is found
    while node is not None:
        # If the node has been logically deleted and we're inserting, use this slot
        if node.deleted and inserting:
            return index
        # If the node is active and matches the key, return its index
        # This condition serves both for inserting (to update) and searching
        if not node.deleted and node.key == key:
            return index

        # Collision resolution through double hashing:
        # Increment probe and calculate new index using both hash functions
        probe += 1
        index = (hash_1 + probe * hash_2) % self.capacity
        # Retrieve the node at the new index
        node = self.table[index]

    # Return the final calculated index
    return index

    # endregion
```

# _insert

```python
def _insert(self, key: str, value: T) -> None:
    """
    Insert key and value into HashTable
    :param key: string to be used as key value
    :param value: int to be used as value mapped to key
    """
    # region
    # Call hash without inserting true the first time to make sure we get the current value if it exists somewhere
    index = self._hash(key)

    # Get the node at that index
    node = self.table[index]

    # If the node exists and the keys match, just update the value
    if node is not None and node.key == key:
        node.value = value  # type: ignore

    # Otherwise, we need to make a new node
    else:
        # Searching without inserting turned on must search past deleted nodes to make sure we find
        # any slot that could contain the key. However, we still want to insert into the first empty slot.
        # So, we call hash again with inserting true
        index = self._hash(key, inserting=True)
        self.table[index] = HashNode(key=key, value=value)
        self.size += 1

    if (self.size / self.capacity) >= 0.5:
        self._grow()
    # endregion
```

# _get

```python
def _get(self, key: str) -> Optional[HashNode]:
    """
    Returns the hash node in the table based on the key
    :param key: key of hash node to find in hash table
    :return item: value in table if key exists, else None
    """
    # region
    index = self._hash(key)

    # If the key is valid
    if index is not None:
        node = self.table[index]
        if not node:  # The key does not exist in this case
            return None
        # Check if the index contains the correct key
        if node.key == key:
            return node
    # endregion
```

# _delete

```python
def _delete(self, key: str) -> None:
    """
    Delete a key from the dictionary
    :param key: the key we are deleting from the hash table
    :return: None
    """
    # region
    index = self._hash(key)

    # If the key is valid
    if index is not None:
        node = self.table[index]
        if not node or node.deleted:  # added deleted check here 2022
            return
        # Check if the index contains the correct key
        # Create empty hashnode and set the deleted flag
        if node.key == key:
            self.table[index] = HashNode(None, None, True)
            self.size -= 1
    # endregion
```