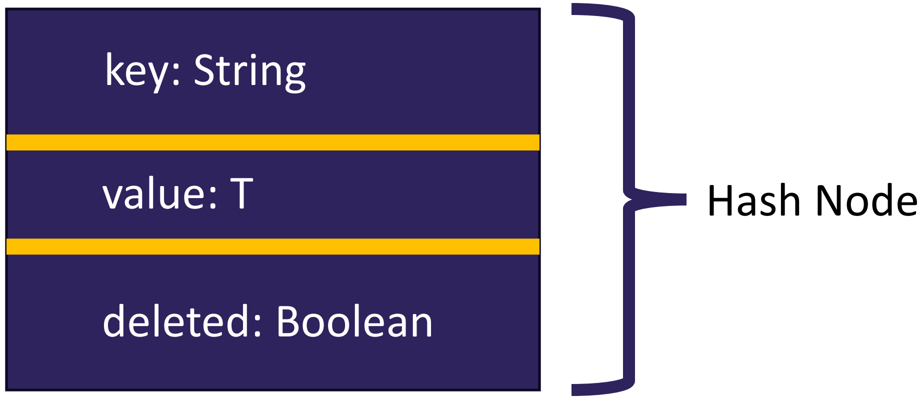
In your project

class HashNode:

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **key: str:** The key of the hash node (this is what is used in hashing)
  - **value: T:** Value being held in the node. Note that this may be any type, such as a `str`, `int`, `float`, `dict`, or a more complex object
  - **deleted: bool:** Whether or not the node has been deleted
- **__init__(self, key: str, value: T, deleted: bool = False) -> None**
  - Constructs a hash node
  - **key: str:** The key of the hash node
  - **value: T:** Value being held in the node
  - **deleted: bool:** Whether or not the node has been deleted. Defaults to false
  - **Returns:** `None`
  - *Time Complexity: O(1)*



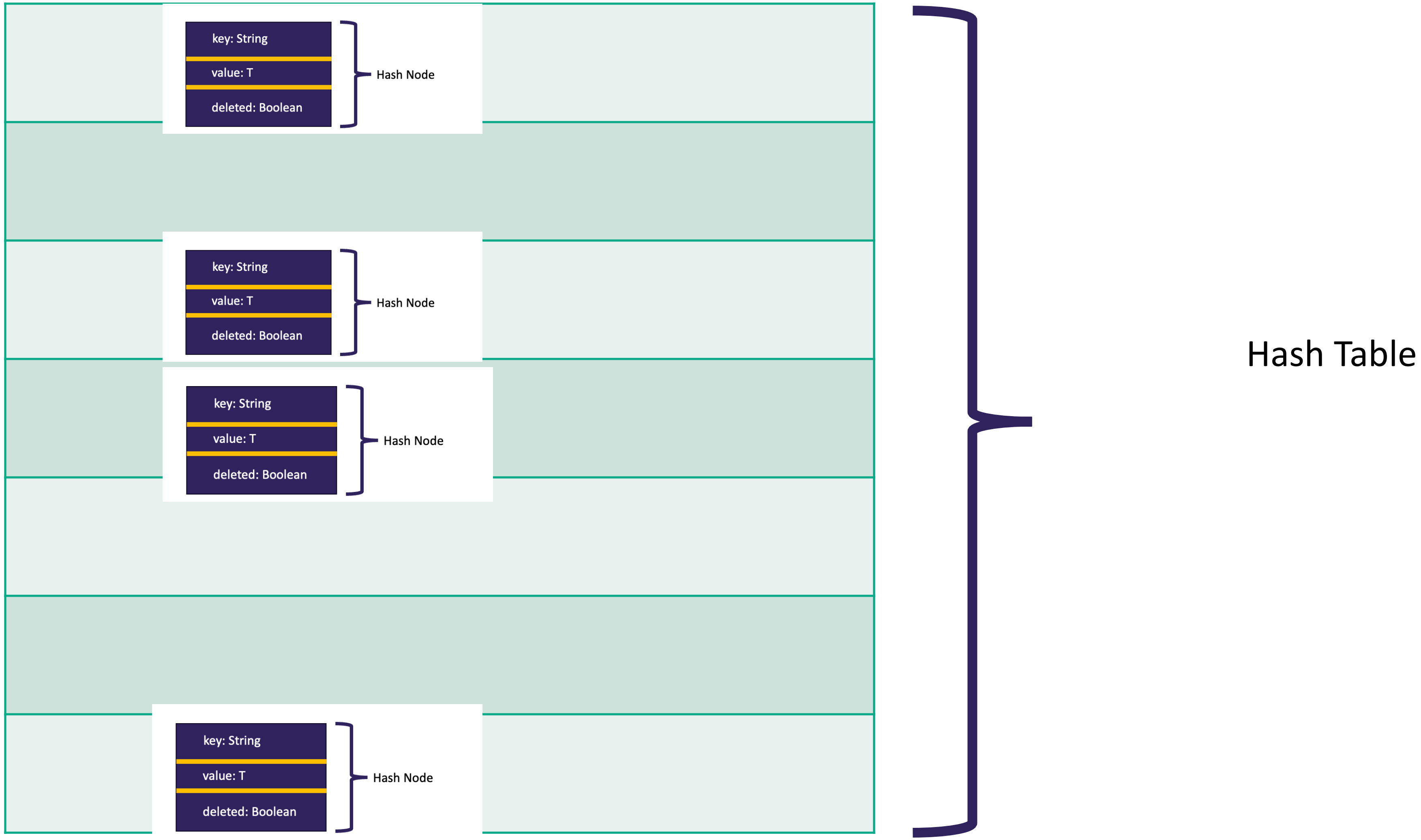
Hash Table

class HashTable:

DO NOT MODIFY the following attributes/functions

- **Attributes** (you may edit the values of attributes but do not remove them)
  - **capacity: int:** Capacity of the hash table
  - **size: int:** Current number of nodes in the hash table
  - **table: List:** This is where the actual data for our hash table is stored
  - **prime_index: int:** Current index of the prime numbers we are using in _hash_2()

Numerous Hash Nodes in Hash Table



This project works with double hashing

- *Time Complexity: O(N)*

- **_hash_1(self, key: str) -> int**
  - The first of the two hash functions used to turn a key into a bin number
  - Assume this is O(1) time/space complexity
  - **key: str:** key we are hashing
  - **Returns:** `int` that is the bin number
  - *Time Complexity: O(1)* (assume)
- **_hash_2(self, key: str) -> int**
  - The second of the two hash functions used to turn a key into a bin number. This hash function acts as the tie breaker.
  - Assume this is O(1) time/space complexity
  - **key: str**: key we are hashing
  - **Returns:** `int` that is the bin number
  - *Time Complexity: O(1)* (assume)

Guide to Hash Function

Implement a hash function that uses probing with double hashing for collision resolution.

Step 1:Primary & Secondary Hash Functions:

- **Primary Hashing**: Converts the key into an index within the hash table's capacity. The objective is to distribute the data evenly across the hash table.

- **Secondary Hashing** (Double Hashing):
Used as a "step size" when collisions occur.
Instead of stepping linearly (i.e., `index+1`), we step by the result of this secondary hash function. This ensures a better distribution during collision resolution.

Step 2: Implement a hash function that uses probing with double hashing for collision resolution.

Initialization:

- Start with the result of the **primary hash function** as your initial index.
- Initialize a probe counter to track the number of steps taken in collision resolution.

Step 3:Probing the Hash Table:

- Check the node at the current index.

Based on the node's status (**None, occupied, deleted**), decide the next action.

- Scenarios to consider:
  - **If the node is None:** This means the key doesn't exist in the table. If you're inserting, you've found the right spot.
  - **If the node is marked as deleted and you're inserting**: Again, this is an available spot for your new entry.
  - **If the node contains the desired key and is not marked as deleted**: If you're searching, you've found the key. If you're inserting, this is where you'd overwrite the value (or decide your next action).
- If none of the above scenarios match, there's a collision. You must then resolve it.