Weekly Lab

Hash Table

In this lab, we will implement a data structure that can store and query data quickly: Hash table.

1 Example

Before implementing our own hash table, let's look at how Python provides a built-in hash table structure via the `dict` type.

```python
# Create a hash table using Python's built-in dict,
# where student name is the key and the score is the value
scores = {
    "An": 8.5,
    "Binh": 7.0,
    "Chi": 9.0
}

# Add a key
scores["Dung"] = 6.5

# Update a key
scores["An"] = 9.5

# Search for a key
print(scores["An"])  # Output: 9.5

# Remove a key
del scores["Binh"]

# Check if a key exists
if "Chi" in scores:
    print("Chi's score is", scores["Chi"])
```

Python's `dict` is implemented as a hash table under the hood, using a combination of hashing and open addressing. It provides average-case constant time for insertion, deletion, and lookup. In this lab, you will implement a simplified version of such a structure from scratch to understand how it works internally.

2 Hash Table

Students are required to implement a hash table with 5 different collision handling: Linear Probing, Quadratic Probing, Chaining using Linked List, Chaining using AVL Tree and Double Hashing.

2.1 Linear Probing

Implement a class `HashTable` using the **linear probing** technique for collision resolution, with the following attributes:

- `table: list[HashNode or None]`: This is the list that stores the data of the hash table. Each element is either `None` or a `HashNode` object representing a key-value pair. For example:

```python
class HashNode:
    def __init__(self, key: str, value: int):
        self.key = key
        self.value = value
```

- `capacity: int`: The size of the hash table (equal to `len(table)`).

The following methods are required:

- `def __init__(self, hashSize: int) -> None`: This is a constructor for initializing an empty hash table with the specified size.
- `def __del__(self) -> None`: Destructor to clean up when the object is deleted.
- `def add(self, key: str, value: int) -> None`: Inserts a key-value pair into the table. If the key already exists, updates its associated value.
- `def searchValue(self, key: str) -> int`: Searches for the value associated with the given key. Returns the value if found; otherwise, returns `None`.
- `def removeKey(self, key: str) -> None`: Removes the key and its associated value from the table, if present.

You may define additional attributes or helper methods if needed.

You can use Python's built-in `hash()` function to compute the hash value of a key. The index for storing the key should be calculated as:

$$\text{index} = (\text{hash}(key) + i) \bmod \text{capacity}$$

where i is the number of probing attempts.

In `if __name__ == "__main__"`: Initialize the hash table with a small size (e.g., 10), and then perform add, search, and remove operations. No user input or menu is necessary.

2.2 Quadratic Probing

Implement a hash table using **quadratic probing** with the same attributes and methods as in Linear Probing. However, when a collision occurs, instead of checking the next slot (linear probing), use the formula:

$$\text{index} = (\text{hash}(key) + i^2) \bmod \text{capacity}$$

where i is the number of probing attempts.

2.3 Chaining using Linked List

Implement a hash table using **chaining** with the same attributes and methods as in Linear Probing. However, the `HashNode` class must be modified to support a **singly linked list**, as shown below:

```python
class HashNode:
    def __init__(self, key: str, value: int):
        self.key = key
        self.value = value
        self.next = None  # Add this line
```

Each slot in the table will point to the head of a linked list. You will also need to implement basic linked list operations (insertion, search, and deletion) inside each bucket.

2.4 Chaining using AVL Tree

Implement a hash table using **chaining**, but each bucket will store an **AVL tree** instead of a linked list. The `HashNode` class can be structured as follows:

```python
class HashNode:
    def __init__(self, key: str, value: int):
        self.key = key
        self.value = value
        self.left = None   # Left child in the AVL tree
        self.right = None  # Right child in the AVL tree
        self.height = 1    # Optional: for balancing
```

You will need to implement insertion, search, and deletion in an AVL tree for each bucket.

2.5 Double Hashing

Implement a hash table that using **double hashing**, with the same attributes and methods as in Linear Probing. In this method, two hash functions are used.

Primary hash:

$$h_1(s) = \text{hash}(s) \bmod \text{capacity}$$

Secondary hash:

$$h_2(s) = 1 + (\text{hash}(s) \bmod \text{prime})$$

where `prime` is a prime number less than `capacity` (e.g., 7 for a table size of 10).

Probing formula:

$$\text{index} = (h_1(s) + i \times h_2(s)) \bmod \text{capacity}$$

where i is the current probing attempt (starting from 0).

Regulations

Please follow these guidelines:

- You may use any Python IDE.
- After completing assignment, check your submission before and after uploading to Moodle.
- Do not use the following modules: `numpy`, `pandas`, `collections`, `heapq`, and `deque`.
- You may use `list`, `tuple`, and `set` but no external libraries.

Your submission must be contributed in a compressed file, named in the format `StudentID.zip`, with the following structure:

```
StudentID
├── Exercise_1.py
├── Exercise_2.py
├── Exercise_3.py
├── Exercise_4.py
└── Exercise_5.py
```

The end.