Dictionaries: Takeaways

Syntax

- Entry class:

```
class Entry:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

- Dictionary class:

```
class Dictionary:
    def __init__(self, num_buckets):
        self.num_buckets = num_buckets
        self.buckets = [LinkedList() for _ in range(num_buckets)]
        self.length = 0
    def _get_index(self, key):
        hashcode = hash(key)
        return hashcode % self.num_buckets
    def put(self, key, value):
        index = self._get_index(key)
        found_key = False
        for entry in self.buckets[index]:
            if entry.key == key:
                entry.value = value
                found_key = True
        if not found_key:
            self.buckets[index].append(Entry(key, value))
            self.length += 1
    def get_value(self, key):
        index = self._get_index(key)
        for entry in self.buckets[index]:
            if entry.key == key:
```

```
                print(entry.key, entry.value)
                return entry.value
        raise KeyError(key)
    def delete(self, key):
        index = self._get_index(key)
        new_bucket = LinkedList()
        for entry in self.buckets[index]:
            if entry.key != key:
                new_bucket.append(entry)
        if len(new_bucket) < len(self.buckets[index]):
            self.length -= 1
        self.buckets[index] = new_bucket
    def __getitem__(self, key):
        return self.get_value(key)
    def __setitem__(self, key, value):
        self.put(key, value)
    def __len__(self):
        return self.length
```

Concepts

- Dictionaries work by mapping keys into a range of integers 0 to *B - 1*. The entries are stored by allocating an array-like data structure with *B* entries.
- To calculate the bucket index of a key, we use a **hash function** to convert the key to an integer and then a **compression function** to reduce that integer to the range 0 to *B - 1*.
- A common compression function is `h % B`, where `h` is the hash code of the key.
- A **collision** occurs when two keys map to the same bucket. One way to handle collisions is to use a linked list to store all entries that map to a given bucket index. We call this technique **separate chaining**.
- The time complexity of dictionary operations is *O(N)* in the worst case. However, if we use good hash functions and a good number of buckets, we can expect the complexity to be *O(1)* instead.
- The dictionary efficiency strongly depends on the **load factor** *N / B*. A general rule is to maintain the load factor below 0.75. We can do this by increasing *B* when *N* increases.
- We need to be careful when selecting or increasing *B*. A general rule is to use a prime number for the value of *B*.

Resources

- Hash Table
- Separate Chaining

- Load Factor
- Prime numbers