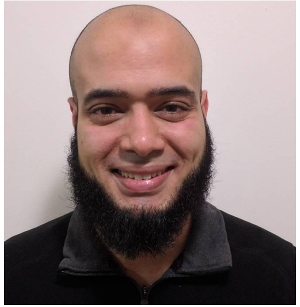
Data Structures

Chaining Implementation

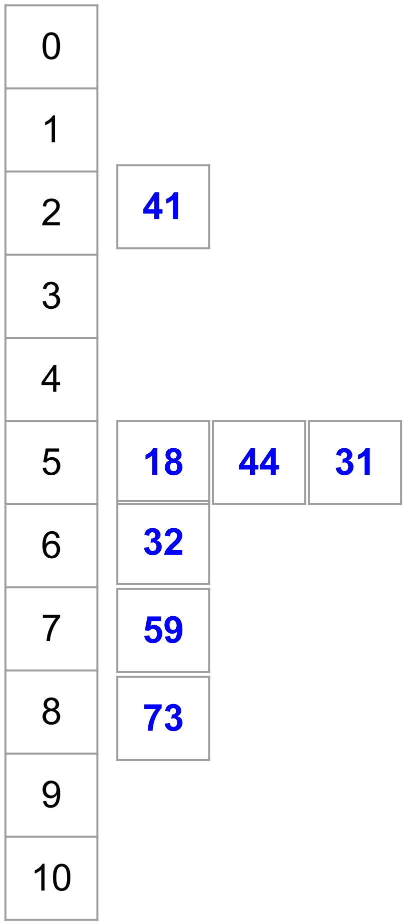
Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Hash-table inner DS

- An easy internal DS is a 2D structure like **list of list**
  - The first list is our array (table) of entries
    - This is where we use the hash_key to access bucket
  - The second internal list maintains items of the same keys
  - E.g. key 5 access the first list
    - Then the inner list is [18, 44, 31]



Let’s implement our own dictionary

- We build our own version of a dictionary
  - So mainly put key vs value
  - The key must support hash(key)
  - This index will be mapped in range [0, table-size -1]
  - We use it to access the table

```python
class OurDict:
    def __init__(self, table_size):
        self.table_size = table_size
        self.table = [None] * table_size

    def add(self, key, value):
        ...

    def print(self):
        ...

    def get(self, key):
        ...

    def exists(self, key):
        ...

    def remove(self, key):
        ...
```

Usage

```
dct = OurDict(table_size=5)

dct.add('Mostafa', 1)   # 2
dct.add('Ziad', 2)      # 2
dct.add('Ali', 5)       # 0
dct.add('Amal', 6)      # 0
dct.add('Hany', 8)      # 4
dct.add('Belal', 10)    # 1
dct.add('Safaa', 11)    # 3
dct.add('Safa', 3)      # 2
dct.add('Ashraf', 4)    # 2

dct.add('Ziad', 555)    # reassign

dct.print()
'''
Key 0 - Pairs [['Ali', 5], ['Amal', 6]]
Key 1 - Pairs [['Belal', 10]]
Key 2 - Pairs [['Mostafa', 1], ['Ziad', 2], ['Safa', 3], ['Ashraf', 4]]
Key 3 - Pairs [['Safaa', 11]]
Key 4 - Pairs [['Saad', 9], ['Hany', 8]]
'''
```

Put item

- Compute hash
  - Table index
- Table[hash]
  - list of items of the same hash

```python
def add(self, key, value):
    hkey = hash(key) % self.table_size  # [0, sz-1]
    new_item = [key, value]

    if self.table[hkey] is None:
        self.table[hkey] = [new_item]   # initial list of items
    else:
        items_equal_key = self.table[hkey]
        for item in items_equal_key:    # search
            if item[0] == key:          # key match
                item[1] = value         # update
                return

        self.table[hkey].append(new_item)
```

Print Item

```python
def print(self):
    for key_values in self.table:
        if key_values is None or len(key_values) == 0:
            continue

        hkey = hash(key_values[0][0]) % self.table_size
        print(f'Key {hkey} - Pairs {key_values}')
```

```python
dct.print()
'''
Key 0 - Pairs [['Ali', 5], ['Amal', 6]]
Key 1 - Pairs [['Belal', 10]]
Key 2 - Pairs [['Mostafa', 1], ['Ziad', 2], ['Safa', 3], ['Ashraf', 4]]
Key 3 - Pairs [['Safaa', 11]]
Key 4 - Pairs [['Saad', 9], ['Hany', 8]]
'''
```

Check existing

```python
def exists(self, key):
    hkey = hash(key) % self.table_size

    if self.table[hkey] is None:
        return False

    items_equal_key = self.table[hkey]
    for idx, item in enumerate(items_equal_key):
        if item[0] == key:
            return True

    return False
```

Get

- Assure it exists

```python
def get(self, key):
    hkey = hash(key) % self.table_size

    assert self.table[hkey] is not None, f'No such item {key}'

    items_equal_key = self.table[hkey]
    for item in items_equal_key:
        if item[0] == key:
            return item[1]

    assert False, f'No value attached with item {key}'
```

Remove

```python
def remove(self, key):
    hkey = hash(key) % self.table_size

    if self.table[hkey] is None:
        return False

    items_equal_key = self.table[hkey]
    for idx, item in enumerate(items_equal_key):
        if item[0] == key:
            items_equal_key.pop(idx)    # O(n) removal
            # more efficient way? Swap with last and pop in O(1)
            return True

    return False
```

```python
dct.print()
'''
Key 0 - Pairs [['Ali', 5], ['Amal', 6]]
Key 1 - Pairs [['Belal', 10]]
Key 2 - Pairs [['Mostafa', 1], ['Ziad', 2], ['Safa', 3], ['Ashraf', 4]]
Key 3 - Pairs [['Safaa', 11]]
Key 4 - Pairs [['Saad', 9], ['Hany', 8]]
'''

print(dct.exists('Ziad'))  # True
print(dct.exists('Saad'))  # False

print(dct.get('Amal'))  # 6
print(dct.get('Ziad'))  # 555
#print(dct.get('Saad'))  # AssertionError

print(dct.remove('Ziad'))   # True
print(dct.remove('Ziad'))   # False
print(dct.remove('Saad'))   # False

# Optional homework: OOP this class with special methods
```

Load Factor

- Assume we have 30 strings and a table of size 10
- The best case is if every table cell has 30/10 = 3 elements
  - Then we take O(3) to find an element
  - Total_elements / table_size ⇒ this is called the **load_factor**
  - This means our hashed keys are almost uniform in distribution
  - If so ⇒ very short chain of nodes
- What if a small number of buckets contain most of the elements, and the rest of the array is largely empty?
  - Then your hash function is very bad!

Rehashing

- What does it mean to have a hash table with a load factor of 1? Like in C#
  - If we expect N entries, our table size is N ⇒ N / N = 1
  - In Java, the default load factor is 0.75
- However, we may not know how many elements there are
  - Decide a load_factor_limit (e.g. 0.75)
  - Keep adding elements, if added/size > load_factor_limit: Do **rehashing**
    - Create a bigger hash-table: 2 * old **size**
    - Move data from old to new and delete old
    - *Now, we again have fewer elements per bucket ⇒ faster operations*
  - As you see: there's a clear trade-off between time and memory!
- With good choices, each hash entry can contain minimal elements ⇒ $\Theta(1)$
- *The chaining technique might not be efficient to store on a file*

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."