3. Hashing Motivation and Terminology:
   a) Sequential search of an array or linked list follows the same search pattern for any
given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.
If \( n \) is the number of items being searched, what is the average and worst case big-oh notation for a sequential search?
   - average case \( O(\sqrt[2]{n}) \)
   - worst case \( O(n) \)

   b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given
   target value. For example, binary search always compares the target value with the middle element of the remaining
   portion of the array needing to be searched.
If \( n \) is the number of items being searched, what is the average and worst case big-oh notation for a search?
   - average case \( O(\log_2(n)) \)
   - worst case \( O(n) \)

Hashing tries to achieve average constant time (i.e., \( O(1) \)) searching by using the target's value to calculate where
in the array/Python list (called the hash table) it should be located, i.e., each target value gets its own search pattern.
The translation of the target value to an array index (called the target's home address) is the job of the hash function.
A perfect hash function would take your set of target values and map each to a unique array index.

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td></td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td>1</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td>2</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 8</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>hash(Sarah Diesburg) = 5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Philip East</td>
<td>3-2939</td>
<td></td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>3-5918</td>
<td></td>
</tr>
<tr>
<td>John Doe</td>
<td>3-4567</td>
<td></td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>3-2187</td>
<td></td>
</tr>
<tr>
<td>Ben Schafer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) If \( n \) is the number of items being searched and we had a perfect hash function, what is the average and worst case
big-oh notation for a search?
   - average case \( O(1) \)
   - worst case \( O(1) \)

4. Unfortunately, perfect hash functions are a rarity, so in general many target values might get mapped to the same
hash-table index, called a collision.

Collisions are handled by two approaches:
   - open-address with some rehashing strategy: Each hash table home address holds at most one target value. The
first target value hashed to a specify home address is stored there. Later targets getting hashed to that home
address get rehashed to a different hash table address. A simple rehashing strategy is linear probing where the
hash table is scanned circularly from the home address until an empty hash table address is found.
   - chaining, closed-address, or external chaining: all target values hashed to the same home address are stored in a
data structure (called a bucket) at that index (typically a linked list, but a BST or AVL-tree could also be used).
Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)
c) Assume double hashing, insert “Andrew Berns” and “Sarah Diesburg” into the hash table.

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td></td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td></td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td></td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 0</td>
<td></td>
</tr>
<tr>
<td>Andrew Berns</td>
<td>hash(Andrew Berns) = 3</td>
<td></td>
</tr>
<tr>
<td>(3-2740)</td>
<td>rehash offset(Andrew Berns) = 1</td>
<td></td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 3</td>
<td></td>
</tr>
<tr>
<td>(3-7395)</td>
<td>rehash offset(Sarah Diesburg) = 3</td>
<td></td>
</tr>
</tbody>
</table>

\[
(3 + 1 \times 3) \mod 8 = 6 \mod 8 = 6
\]
\[
(3 + 2 \times 3) \mod 8 = 9 \mod 8 = 1
\]

d) For the above double-hashing example, what would be the sequence of hashing and rehashing addresses tried for Sarah Diesburg if the table was full? For the above example, (home address + (rehash attempt #) * offset) % (hash table size) would be: 

\[
(3 + (\text{rehash attempt #}) \times 3) \mod 8
\]

<table>
<thead>
<tr>
<th>Rehash Attempt #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.

- primary clustering - keys mapped to a home address follow the same rehash pattern
- secondary clustering - rehash patterns from initially different home addresses merge together

<table>
<thead>
<tr>
<th>Rehash Strategy</th>
<th>Description</th>
<th>Suffers from:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)</td>
<td>Yes</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>Check a square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 + (rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2</td>
<td>Yes, No</td>
</tr>
<tr>
<td>double hashing</td>
<td>Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) \times offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size</td>
<td>No, No</td>
</tr>
</tbody>
</table>

6. Let \( \lambda \) be the load factor (# item/hash table size). The average probes with linear probing for insertion or unsuccessful search is:

\[
\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right)
\]

The average for successful search is:

\[
\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right)
\]

a) Why is an unsuccessful search worse than a successful search?

\[
\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right)
\]

\[
\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right)
\]
The average probes with quadratic probing for insertion or unsuccessful search is: \( \left( \frac{1}{1 - \lambda} \right) - \lambda - \log_e(1 - \lambda) \)

The average probes with quadratic probing for successful search is: \( 1 - \left( \frac{1}{2} \right) - \log_e(1 - \lambda) \)

Consider the following table containing the average number probes for various load factors:

<table>
<thead>
<tr>
<th>Probing Type</th>
<th>Search outcome</th>
<th>Load Factor, ( \lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>unsuccessful</td>
<td>1.39, 2.50, 5.09, 13.00, 5000.50</td>
</tr>
<tr>
<td>Probing</td>
<td>successful</td>
<td>1.17, 1.50, 2.02, 3.00, 50.50</td>
</tr>
<tr>
<td>Quadratic</td>
<td>unsuccessful</td>
<td>1.37, 2.19, 3.47, 5.81, 103.62</td>
</tr>
<tr>
<td>Probing</td>
<td>successful</td>
<td>1.16, 1.44, 1.77, 2.21, 5.11</td>
</tr>
</tbody>
</table>

b) Why do you suppose the "general rule of thumb" in hashing tries to keep the load factor between 0.5 and 0.67?

**Bigger 0.67 for load factor causes too many probes.
Less than 0.5 wastes space.**

7. Allowing deletions from an open-address hash table complicates the implementation. Assuming linear probing we might have the following:

- **Set of Keys**
- **Hash function**
- **Hash Table Array**

- John Doe hash(John Doe) = 6
- Philip East hash(Philip East) = 3
- Mark Fienup hash(Mark Fienup) = 5
- Ben Schafer hash(Ben Schafer) = 8
- Andrew Berns hash(Andrew Berns) = 3
- Sarah Diesburg hash(Sarah Diesburg) = 3

a) If "Mark Fienup" is deleted, how will we find Sarah Diesburg?

b) How might we fix this problem? **Special Deleted value True**

**Special Empty value None**
1. The Map/Dictionary abstract data type (ADT) stores key-value pairs. The key is used to look up the data value.

<table>
<thead>
<tr>
<th>Method call</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = ListDict()</td>
<td><strong>init</strong>(self)</td>
<td>Constructs an empty dictionary</td>
</tr>
<tr>
<td>d[“Name”] = “Bob”</td>
<td><strong>setitem</strong>(self, key, value)</td>
<td>Inserts a key-value entry if key does not exist or replaces the old value with new value if key exists.</td>
</tr>
<tr>
<td>temp = d[“Name”]</td>
<td><strong>getitem</strong>(self, key)</td>
<td>Given a key return it value or None if key is not in the dictionary</td>
</tr>
<tr>
<td>del d[“Name”]</td>
<td><strong>delitem</strong>(self, key)</td>
<td>Removes the entry associated with key</td>
</tr>
<tr>
<td>if “Name” in d:</td>
<td><strong>contains</strong>(self, key)</td>
<td>Return True if key is in the dictionary; return False otherwise</td>
</tr>
<tr>
<td>for k in d:</td>
<td><strong>iter</strong>(self)</td>
<td>Iterates over the keys in the dictionary</td>
</tr>
<tr>
<td>len(d):</td>
<td><strong>len</strong>(self)</td>
<td>Returns the number of items in the dictionary</td>
</tr>
<tr>
<td>str(d):</td>
<td><strong>str</strong>(self)</td>
<td>Returns a string representation of the dictionary</td>
</tr>
</tbody>
</table>

```
from entry import Entry
class ListDict(object):
    """Dictionary implemented with a Python list."""
    def __init__(self):
        self._table = []
    def __getitem__(self, key):
        """Returns the value associated with key or returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            # NOTE: 'Python list index method
            # errors on unsuccessful search
            index = self._table.index(entry)
        except:
            return None
        return self._table[index].getValue()

    def __delitem__(self, key):
        """Removes the entry associated with key."""
        entry = Entry(key, None)
        try:
            # NOTE: Python list index method
            # errors on unsuccessful search
            index = self._table.index(entry)
        except:
            return
        self._table.pop(index)

    def __str__(self):
        """Returns string repr. of the dictionary"""
        resultStr = ""'
        for item in self._table:
            resultStr += " + str(item)"
        return resultStr + ""

    def __iter__(self):
        """Iterates over keys of the dictionary"""
        for item in self._table:
            yield item.getKey()
```

a) Complete the code for the __contains__ method.

```
def __contains__(self, key):
```

b) Complete the code for the __setitem__ method.

```
def __setitem__(self, key, value):
```