1. Consider the following `ListDict` class implementation.

```python
# File: dictionary.py

class Entry(object):
    """A key/value pair."""
    def __init__(self, key, value):
        self.key = key
        self.value = value
    def __eq__(self, other):
        return self.key == other.key
    def __str__(self):
        return str(self.key) + ':' + str(self.value)

class ListDict(object):
    """A list-based implementation of a dictionary."""
    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:
            index = self._table.index(entry)
            return self._table[index].value
        except:
            return None
    def pop(self, key):
        """Removes the entry associated with key and returns its value or returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            return self._table.pop(index).value
        except:
            return None
    def __setitem__(self, key, value):
        """Inserts an entry with key/value if key does not exist or replaces the existing value with value if key exists."""
        entry = Entry(key, value)
        try:
            index = self._table.index(entry)
            self._table[index] = entry
        except:
            self._table.append(entry)

# The methods __len__(), __str__(), keys(),
# __contains__, and values() are exercises
```

a) Explain how the `__getitem__` method looks up a key?

b) What is the average-case theta notation of each operation?

- `__getitem__`: \( \Theta(1) \)
- `pop`: \( \Theta(n) \)
- `__setitem__`: \( \Theta(n) \)
- `__contains__`: \( \Theta(n) \)

---

**ListDict object**

<table>
<thead>
<tr>
<th>_table</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
</table>

**Python list object**

[Table 19.3] The Interface of the `ListDict` class
2. Consider the following HashDict class implementation.

```python
from arrays import Array

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

class HashDict(object):
    """A hashing implementation of a dictionary."""

    DEFAULT_CAPACITY = 3

    def __init__(self, capacity = None):
        if capacity is None:
            self._capacity = HashDict.DEFAULT_CAPACITY
        else:
            self._capacity = capacity
        self._table = Array(self._capacity)
        self._size = 0
        self._priorEntry = None
        self._foundEntry = None
        self._index = None

    def __contains__(self, key):
        """Returns True if key is in the dictionary or False otherwise."""
        self._index = abs(hash(key)) % self._capacity
        self._priorEntry = None
        self._foundEntry = self._table[self._index]
        while self._foundEntry != None:
            if self._foundEntry.key == key:
                return True
            else:
                self._priorEntry = self._foundEntry
                self._foundEntry = self._foundEntry.next
        return False

    def __getitem__(self, key):
        """Returns the value associated with key or returns None if key does not exist."""
        if key in self:
            return self._foundEntry.value
        else:
            return None

    def pop(self, key):
        """Removes the entry associated with key and returns its value or returns None if key does not exist."""
        if not key in self:
            return None
        else:
            if self._priorEntry is None:
                self._table[self._index] = self._foundEntry.next
            else:
                self._priorEntry.next = self._foundEntry.next
            self._size -= 1
            return self._foundEntry.value

    def __setitem__(self, key, value):
        """Inserts an entry with key/value if key does not exist or replaces the existing value with value if key exists."""
        if not key in self:
            newEntry = HashEntry(key, value, self._table[self._index])
            self._table[self._index] = newEntry
            self._size += 1
        else:
            returnValue = self._foundEntry.value
            self._foundEntry.value = value
            return returnValue

    def __len__(self):
        return self._size

    def __str__(self):
        result = "HashDict: capacity = " + str(self._capacity) + ", load factor = " + str(len(self) / float(self._capacity))
        for i in xrange(self._capacity):
            entry = self._table[i]
            while entry != None:
                rowStr = "" + str(entry) + " 
                entry = entry.next
            if rowStr != "":
                result += "Row " + str(i) + ": " + rowStr
        return result

# The methods keys() and values() are exercises
```

a) Explain how the __getitem__ method looks up a key?

b) What is the average-case theta notation of each operation? (Let α be load factor (n/Array size))
   __getitem__(self, key):
   pop(self, key):
   __setitem__(self, key, value):
   __contains__(key):
3. Consider the following `HashTable` class implementation.

```python
from arrays import Array

class HashTable(object):
    """Represents a hash table."""

    EMPTY = None
    DELETED = True

    def __init__(self, capacity = 29,
                 hashFunction = hash,
                 linear = True):
        self._table = Array(capacity, HashTable.EMPTY)
        self._size = 0
        self._hash = hashFunction
        self._homeIndex = -1
        self._actualIndex = -1
        self._linear = linear
        self._probeCount = 0

    def insert(self, item):
        """Inserts item into the table
        Preconditions: There is at least one empty cell or
        one previously occupied cell.
        There is not a duplicate item."""
        self._probeCount = 0
        # Get the home index
        self._homeIndex = abs(self._hash(item)) % len(self._table)
        distance = 1
        index = self._homeIndex

        # Stop searching when an empty cell is encountered
        while not self._table[index] in (HashTable.EMPTY,
                                          HashTable.DELETED):
            # Increment the index and wrap around to first
            # position if necessary
            if self._linear:
                increment = index + 1
            else:
                # Quadratic probing
                increment = self._homeIndex + distance ** 2
                distance += 1
            index = increment % len(self._table)
            self._probeCount += 1

        # An empty cell is found, so store the item
        self._table[index] = item
        self._size += 1
        self._actualIndex = index

    def search(self, item):
        """Search for item in the table."""
        self._probeCount = 0
        # Get the home index
        self._homeIndex = abs(self._hash(item)) % len(self._table)
        distance = 1
        index = self._homeIndex

        # Stop searching when an empty cell is encountered
        while not self._table[index] in (HashTable.EMPTY,
                                          HashTable.DELETED):
            # Increment the index and wrap around to first
            # position if necessary
            if self._linear:
                increment = index + 1
            else:
                # Quadratic probing
                increment = self._homeIndex + distance ** 2
                distance += 1
            index = increment % len(self._table)
            self._probeCount += 1

        # An empty cell is found, so return None
        if self._table[index] == HashTable.EMPTY:
            return None
        else:
            self._actualIndex = index
            return self._table[index]

    # Methods __len__(), __str__(), loadFactor(), homeIndex(),
    # actualIndex(), and probeCount() are exercises.
```

Let $\alpha$ be the load factor. The average probes with **linear probing** for insertion or unsuccessful search is:

$$\left(\frac{1}{2}\right)(1 + \frac{1}{(1-\alpha)^2})$$

The average probes with linear probing for successful search is:

$$\left(\frac{1}{2}\right)(1 + \frac{1}{(1-\alpha)})$$

If $\alpha = 0.8$, what is the average probes for:

a) unsuccessful search?

b) successful search?

c) Why is unsuccessful worse than successful search?
4. Let $\alpha$ be the load factor. The average probes with **quadratic probing** for insertion or unsuccessful search is:

$$\frac{1}{\frac{1}{1-a}} - a - \log_e (1 - a)$$

The average probes with quadratic probing for successful search is:

$$1 - \left(\frac{1}{2}\right) - \log_e (1 - a)$$

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>0.25</th>
<th>0.5</th>
<th>0.67</th>
<th>0.8</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>unsuccessful</td>
<td>1.39</td>
<td>2.50</td>
<td>5.09</td>
<td>13.00</td>
<td>5000.50</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td>1.17</td>
<td>1.50</td>
<td>2.02</td>
<td>3.00</td>
<td>50.50</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>unsuccessful</td>
<td>1.37</td>
<td>2.19</td>
<td>3.47</td>
<td>5.81</td>
<td>103.62</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td>1.16</td>
<td>1.44</td>
<td>1.77</td>
<td>2.21</td>
<td>5.11</td>
</tr>
</tbody>
</table>

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

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

<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>0</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>Paul Gray</td>
<td>hash(Paul Gray) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Kevin O'Kane</td>
<td>hash(Kevin O'Kane) = 4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

The Hash Table Array:

- 0: No entry.
- 1: Philip East (3-2939)
- 2: No entry.
- 3: Philip East (3-2939), Kevin O'Kane (3-7322)
- 4: Paul Gray (3-5917)
- 5: Mark Fienup (3-5918)
- 6: John Doe (3-4567)
- 7: Kevin O'Kane (3-7322), Ben Schafer (3-2187)
- 8: Ben Schafer (3-2187)
- 9: No entry.
- 10: No entry.

a) If "Mark Fienup" is deleted, how will we find Kevin O'Kane?

b) How might we fix this problem?