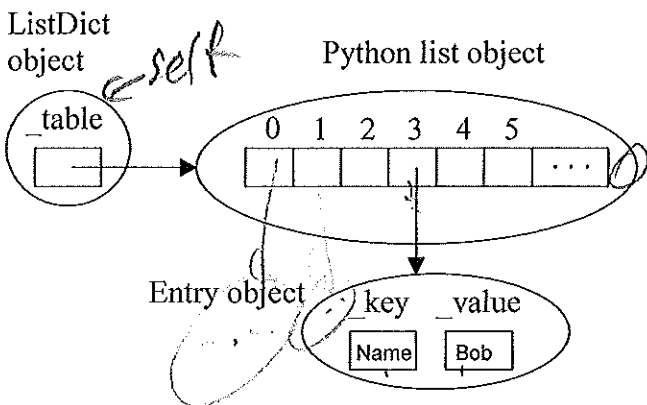


1. The Map/Dictionary abstract data type (ADT) stores key-value pairs. The key is used to look up the data value.

Method call	Class Name	Description
d = ListDict()	<code>__init__(self)</code>	Constructs an empty dictionary
d["Name"] = "Bob"	<code>__setitem__(self, key, value)</code>	Inserts a key-value entry if key does not exist or replaces the old value with value if key exists.
temp = d["Name"]	<code>__getitem__(self, key)</code>	Given a key return its value or None if key is not in the dictionary
del d["Name"]	<code>__delitem__(self, key)</code>	Removes the entry associated with key
if "Name" in d:	<code>__contains__(self, key)</code>	Return True if key is in the dictionary; return False otherwise
for k in d:	<code>__iter__(self)</code>	Iterates over the keys in the dictionary
len(d)	<code>__len__(self)</code>	Returns the number of items in the dictionary
str(d)	<code>__str__(self)</code>	Returns a string representation of the dictionary



```
class Entry(object):
    """A key/value pair."""

    def __init__(self, key, value):
        self._key = key
        self._value = value

    def getKey(self):
        return self._key

    def getValue(self):
        return self._value

    def setValue(self, newValue):
        self._value = newValue

    def __eq__(self, other):
        if not isinstance(other, Entry):
            return False
        return self._key == other._key

    def __str__(self):
        return str(self._key) + ":" + str(self._value)
```

```
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:
            index = self._table.index(entry)
            return self._table[index].getValue()
        except:
            return None

    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)
            self._table.pop(index)
        except:
            return

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

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

for k in d

a) Complete the code for the `__contains__` method.

```
def __contains__(self, key):
    for entry in self._table:
        if entry.getKey() == key:
            return True
    return False
```

b) Complete the code for the `__setitem__` method.

```
def __setitem__(self, key, value):
    entry = Entry(key, value)
    try:
        index = self._table.index(entry)
        self._table[index] = entry
    except:
        self._table.append(entry)
```

2. Dictionary implementation using hashing with chaining -- an UnorderedList object at each slot in the hash table.

```
from entry import Entry
from unordered_linked_list import UnorderedList
```

```
class ChainingDict(object):
    """Dictionary implemented using hashing with chaining."""
```

```
def __init__(self, capacity = 8):
    self._capacity = capacity
    self._table = []
    for index in range(self._capacity):
        self._table.append(UnorderedList())
    self._size = 0
    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
    entry = Entry(key, None)

    return self._table[self._index].search(entry)
```

```
def __getitem__(self, key):
    """Returns the value associated with key or
    returns None if key does not exist."""
    if key in self:
        entry = Entry(key, None)
        entry = self._table[self._index].remove(entry)
        self._table[self._index].add(entry)
        return entry.getValue()
    else:
        return None
```

```
def __delitem__(self, key):
    """Removes the entry associated with key."""
    if key in self:
        entry = Entry(key, None)
        entry = self._table[self._index].remove(entry)
        self._size -= 1
```

```
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)
    if key in self:
        entry = self._table[self._index].remove(entry)
        entry.setValue(value)
    else:
        self._size += 1
        self._table[self._index].add(entry)
```

```
def __len__(self):
    return self._size
```

```
def __str__(self):
    result = "HashDict: capacity = " + \
        str(self._capacity) + ", load factor = " + \
        str(len(self) / self._capacity)
    for i in range(self._capacity):
        result += "\nRow " + str(i) + ": " + str(self._table[i])
    return result
```

```
def __iter__(self):
    """Iterates over the keys of the dictionary"""
```

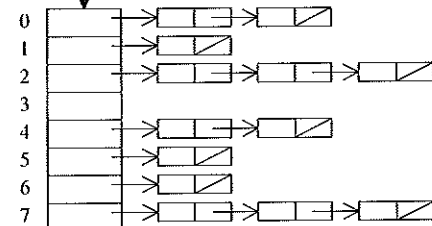
```
for slotList in self._table:
    for item in slotList:
        yield item.getKey()
    raise StopIteration
```

ChainingDict Object

_size 13 _capacity 8

_table [] _index 4

Python list of UnorderList objects containing Entries



a) In `__getitem__`, why is the `entry = Entry(key, None)` object created?

Entry objects stored in UnorderedList
so we need one with correct key to

b) In `__getitem__`, where does `self._index` receive its value?

"if key in self:" calls `__contains__`
which sets `self._index`.

c) What single modification was needed to the UnorderedList's `remove` method?

Original textbook description
did not return the removed item

d) Complete the `__iter__` method.

1. The Dictionary implementation using open-address hashing was the OpenAddrHashDict class in lab7.zip.

```

from entry import Entry

class OpenAddrHashDict(object):
    EMPTY = None # class variables shared by all objects of the class
    DELETED = True

    def __init__(self, capacity = 8, hashFunction = hash,
                 linear = True):
        self._table = [OpenAddrHashDict.EMPTY] * capacity
        self._size = 0
        self._hash = hashFunction
        self._homeIndex = -1
        self._actualIndex = -1
        self._linear = linear
        self._probeCount = 0

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

    def __delitem__(self, key):
        """Removes the entry associated with key."""
        if key in self:
            self._table[self._actualIndex] = OpenAddrHashDict.DELETED
            self._size -= 1

    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)
        if key in self:
            self._table[self._actualIndex] = entry
        else:
            self._table[self._actualIndex] = entry
            self._size += 1

    def __contains__(self, key):
        """Return True if key is in the dictionary; return False otherwise"""
        entry = Entry(key, None)
        self._probeCount = 0
        # Get the home index
        self._homeIndex = abs(self._hash(key)) % len(self._table)
        rehashAttempt = 0
        index = self._homeIndex

        # Stop searching when an empty cell is encountered
        while rehashAttempt < len(self._table):
            self._probeCount += 1
            if self._table[index] == OpenAddrHashDict.EMPTY:
                self._actualIndex = index
                return False # An empty cell is found, so key not found
            elif self._table[index] == entry:
                self._actualIndex = index
                return True

            # Calculate the index and wrap around to first position if necessary
            rehashAttempt += 1
            if self._linear:
                index = (self._homeIndex + rehashAttempt) % len(self._table)
            else: # Quadratic probing
                index = (self._homeIndex + (rehashAttempt ** 2 + rehashAttempt) // 2) % len(self._table)

        return False # tried all the slots in the hash table and did not find key

    def __len__(self):
        return self._size

    def __str__(self):
        resultStr = "{"
        for item in self._table:
            if not item in (OpenAddrHashDict.EMPTY, OpenAddrHashDict.DELETED):
                resultStr = resultStr + " " + str(item)
        return resultStr + "}"

    def __iter__(self):
        """Iterates over the keys of the dictionary"""
        for item in self._table:
            if isinstance(item, Entry):
                yield item.getKey()
            raise StopIteration

```

a) Complete the `__iter__` method.

```
                                unordered_linked_list.py
return resultStr + " (tail)"

def __iter__(self):
    """Iterates over the items of the list"""
    current = self._head
    while current != None:
        yield current.getData()
        current = current.getNext()
    raise StopIteration
```