Hashtables

Implementations for dictionaries and sets

Terence Parr
MSDS program
University of San Francisco

See https://github.com/parrt/msds692/blob/master/notes/hashtable.ipynb and https://github.com/parrt/msds692/blob/master/notes/dict.ipynb

UNIVERSITY OF SAN FRANCISCO

Speed matters

- In data science, data sets are often very large
- Being able to find elements of interest quickly is more challenging in this environment
- Searching through all elements can be prohibitively slow
- We will need to trade some increased complexity and set up time for a data structure in exchange for increased look up speed

How to search big collections quickly

- *Hashtables* are data structures that efficiently implement search/lookup operations for sets and dictionaries
- Sets and dictionaries are abstract data structures, hashtables are concrete implementations of those structures
- Simple lists of elements and lists of tuples work but are slow
- Hashtable's **key idea**: partition the search space into well-defined regions so we don't have to search linearly through the entire collection to find an element
- We use a (hash) function of the values to partition into buckets

Review: Sets

- A set is just an unordered, unique collection of elements; here is an example using integers:
  `ids = {100, 103, 121, 102, 113, 113, 113, 113}`
- We can do lots of fun set arithmetic:

```
{100,102}.union({109})
```

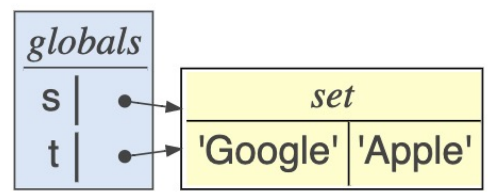
```
{100, 102, 109}
```

```
{100,102}.intersection({100,119})
```

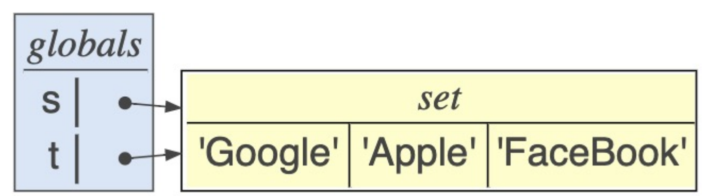
```
{100}
```

Watch out for aliasing!

```
s = {"Apple", "Google"}
t = s
```




```
s.add("FaceBook")
```



Review: Dictionaries map keys to values

- If we arrange two lists side-by-side and kind of glue them together, we get a *dictionary* (type is **dict**)
- Dictionaries map one value to another, just like a dictionary in the real world maps a word to a definition
- Here is a sample dictionary:

```
movies = {'Amadeus':1984, 'Witness':1985}
```

'Amadeus' → 1984
'Witness' → 1985

- Index by key to get the value; e.g., `movies['Amadeus']`

List implementation for sets

```
import numpy as np
n = 5_000_000
A = list(np.random.randint(low=0,high=1_000_000,size=n))
A[0:10]
```

```
[509385, 571020, 998421, 173251, 567339, 229005, 614066, 89806, 878866, 496601]
```

```
def lsearch(A,x):
    for a in A:
        if a==x:
            return True
    return False
```

Searching linearly is pretty slow

```
%time lsearch(A, 999)
```

```
CPU times: user 362 ms, sys: 8.02 ms, total: 370 ms
Wall time: 378 ms
True
```

```
%time for a in range(50): lsearch(A, a)
```

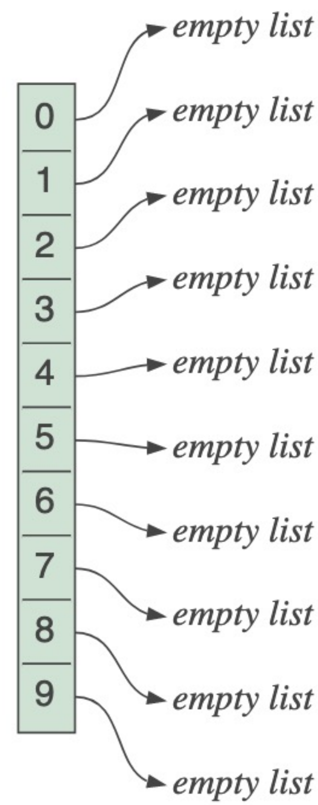
```
CPU times: user 8.65 s, sys: 102 ms, total: 8.75 s
Wall time: 8.9 s
```

Can we do better than linear search?

- Rather than search through every element, let's partition the set of integers into 10 buckets so that, on average, we only need to search 1/10 of the elements
- We partition with a *hash* function that operates on set values
- We are effectively using something about the value to hint at the location (which bucket)
- E.g., where does Eric Erickson live in US? Imagine a hash function that gave the postal code given a name (set value). Faster to search a postal code region than entire country

Partitioning requires a new data structure

- Rather than a list of set values, we break the set into smaller groups, buckets, where each bucket is a list of values
- The hash of a value leads to the bucket index (this is a simplification and not technically correct)
- For sets of integers, let's use the value modulo 10 to uniquely place values into one of 10 buckets, indexed 0..9



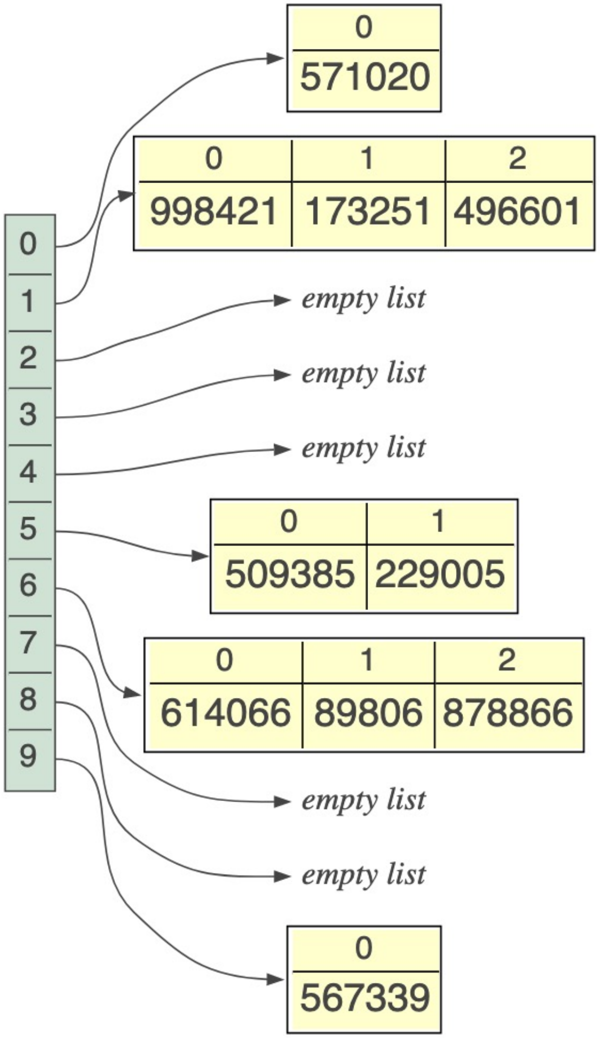
Partitioning with modulo hash

```python
def hash(x):
    return x % 10

[(a,hash(a)) for a in A[0:10]]
```

```
[(509385, 5),
 (571020, 0),
 (998421, 1),
 (173251, 1),
 (567339, 9),
 (229005, 5),
 (614066, 6),
 (89806, 6),
 (878866, 6),
 (496601, 1)]
```

We don't care about the meaning of the mapping from key to bucket except that it evenly distributes and is reproducible



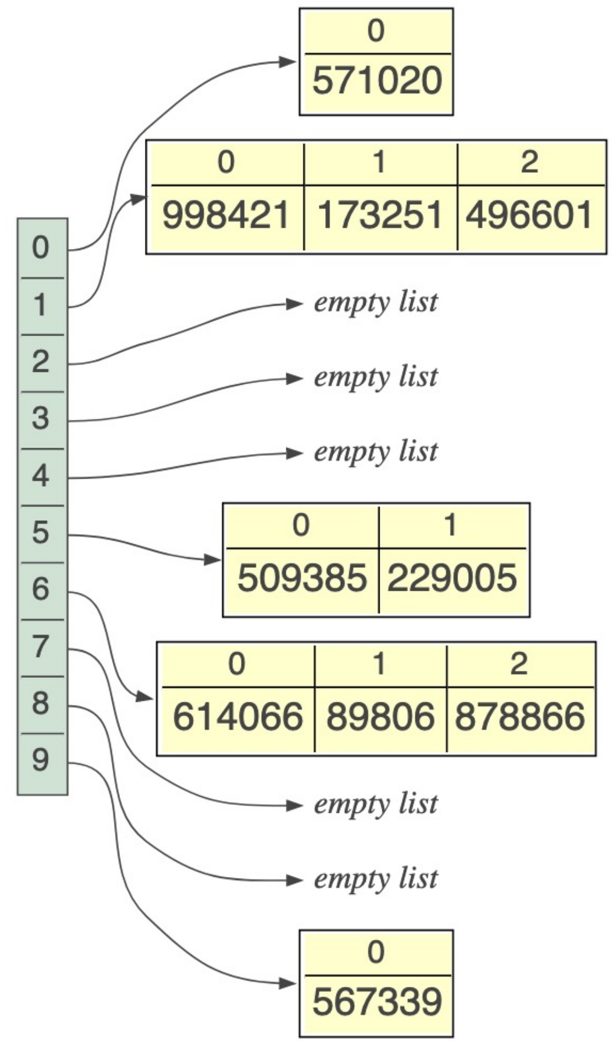
Hashtable construction

- Make 10 empty buckets (lists)
- Add each set element to correct bucket
- Amounts to appending each element to one of 10 lists

```
def hash(x):
    return x % 10
```

```
def htable(A):
    "Build hashtable for integer values"
    buckets = [[] for i in range(10)]
    for a in A:
        buckets[hash(a)].append(a)
    return buckets
```

```
buckets = htable(A)
```



Searching hashtable set implementation

- To find x, look in the bucket indicated by hash(x)

Linear

```python
def lsearch(A,x):
    for a in A:
        if a==x:
            return True
    return False
```

```python
%time lsearch(A, 999)
```

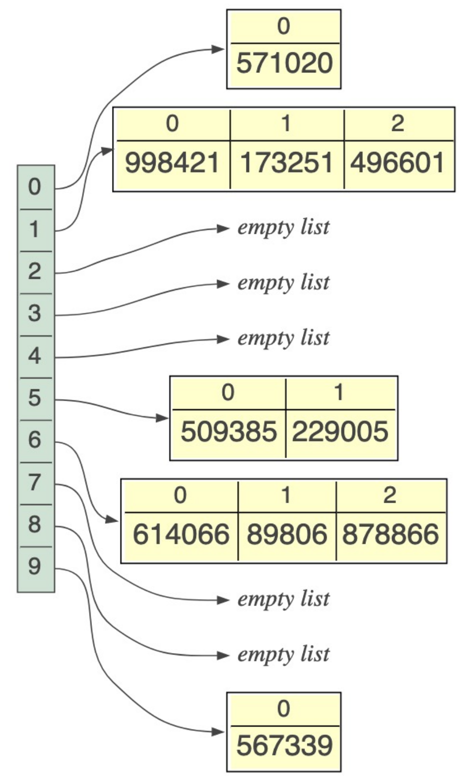
```
CPU times: user 190 ms, sys:
Wall time: 209 ms
```

Hashtable

```python
def hsearch(buckets,x):
    i = hash(x)
    for a in buckets[i]:
        if a==x:
            return True
    return False
```

```python
buckets = htable(A)
%time hsearch(buckets, 999)
```

```
CPU times: user 18 ms, sys:
Wall time: 18.7 ms
```



Speed difference is dramatic

Linear

```
%time for a in range(50): lsearch(A, a)
```

```
CPU times: user 6.97 s, sys: 93.9 ms, total: 7.06 s
Wall time: 7.38 s
```

Hashtable

```
%time for a in range(50): hsearch(buckets, a)
```

```
CPU times: user 823 ms, sys: 14.3 ms, total: 837 ms
Wall time: 861 ms
```

Exercise

- What would happen if we used a hash function like these?

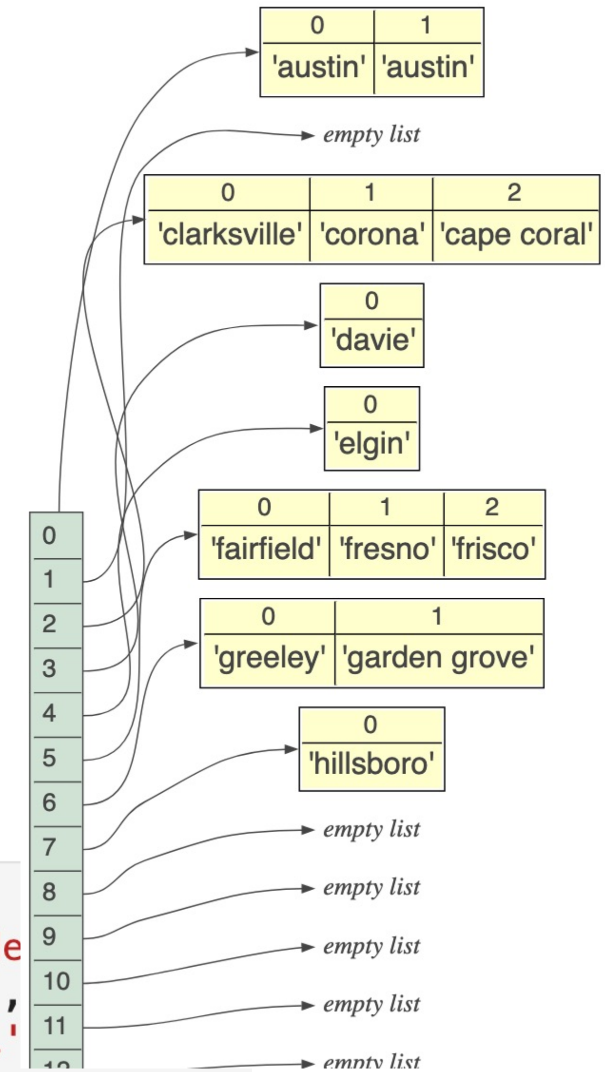
```python
def hash(x):
    return x % 2
```

```python
def hash(x):
    return 0
```

Sets of strings

- Hashtables work for any type of value for which we can define a good hash function, one that partitions the space evenly
- What key→bucket mapping am I using here?

Distance of char code from ‘a’s code



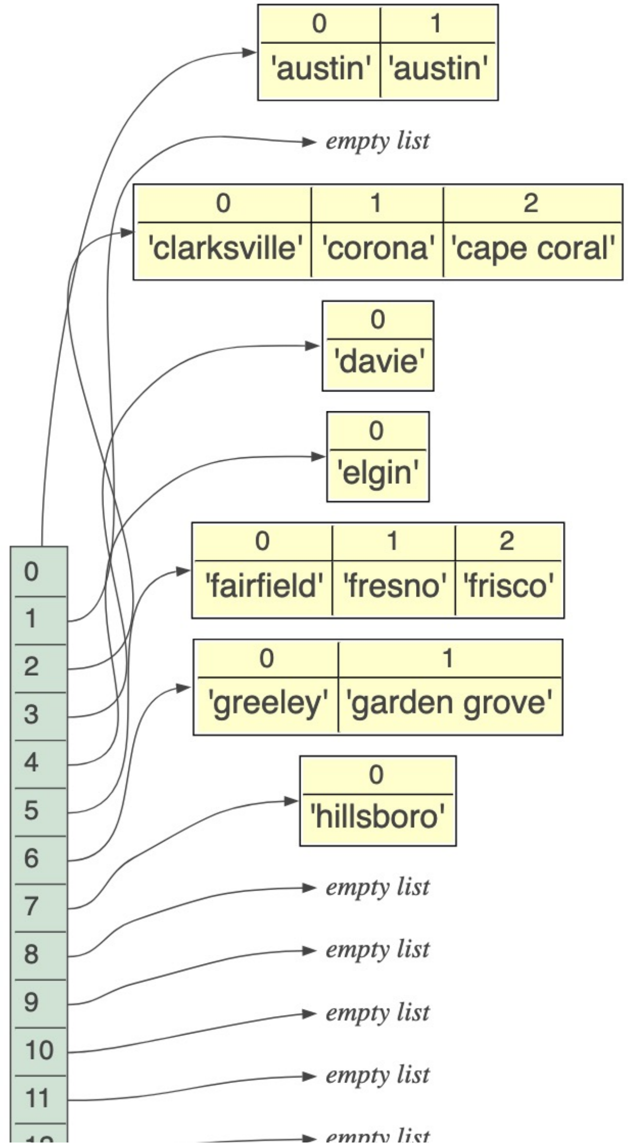

```
cities = ['elgin', 'tyler', 'austin', 'hillsboro', 'greeley',
          'davie', 'rockford', 'orange', 'sandy springs', 'garde
          'paterson', 'clarksville', 'fairfield', 'victorville',
          'palmdale', 'frisco', 'corona', 'austin', 'cape coral'
```

Hashing strings

```
def hash(s):
    # convert first char to int in [0,25]
    return ord(s[0]) - ord('a')

[(c,hash(c)) for c in cities[0:10]]
```

```
[('elgin', 4),
 ('tyler', 19),
 ('austin', 0),
 ('hillsboro', 7),
 ('greeley', 6),
 ('davie', 3),
 ('rockford', 17),
 ('orange', 14),
 ('sandy springs', 18),
 ('garden grove', 6)]
```



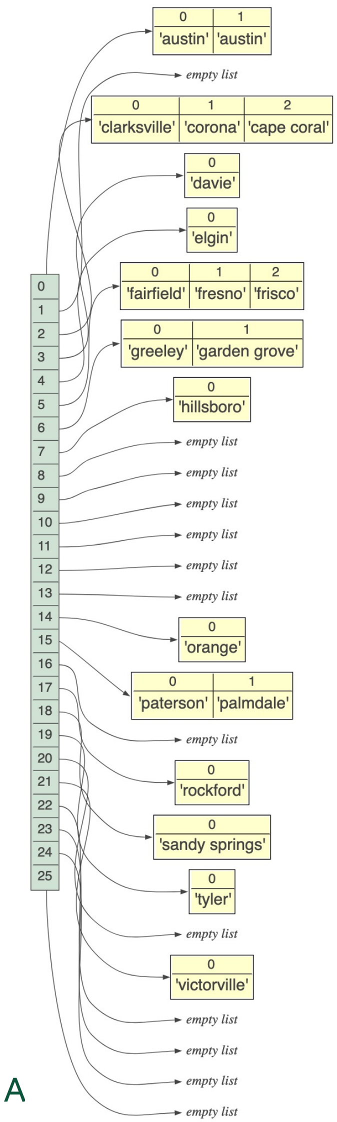
This impl of hashtable for strings vs ints differs only in number of buckets

```python
def htable(A):
    buckets = [[] for i in range(26)]
    for a in A:
        buckets[hash(a)].append(a)
    return buckets
```

```python
def hsearch(buckets,x):
    i = hash(x)
    for a in buckets[i]:
        if a==x:
            return True
    return False
```

```python
buckets = htable(cities)
%time hsearch(buckets, "austin")
```

```
CPU times: user 5 µs, sys: 0 ns, total: 5 µs
Wall time: 7.15 µs
```



UNIVERSITY OF SA

An important implementation detail

- The hash function does not directly give the bucket index: it converts values to integers, then we make sure that the hash value fits into the table by doing modulo the number of buckets
- For our int sets, the hash(x) is just x; the modulo 10 just puts it in one of 10 buckets, but we can use any number of buckets
- Same for strings; the hash(x) could be simply the character code for the first character, but we could squeeze all 26 English chars into 10 buckets by taking modulo 10

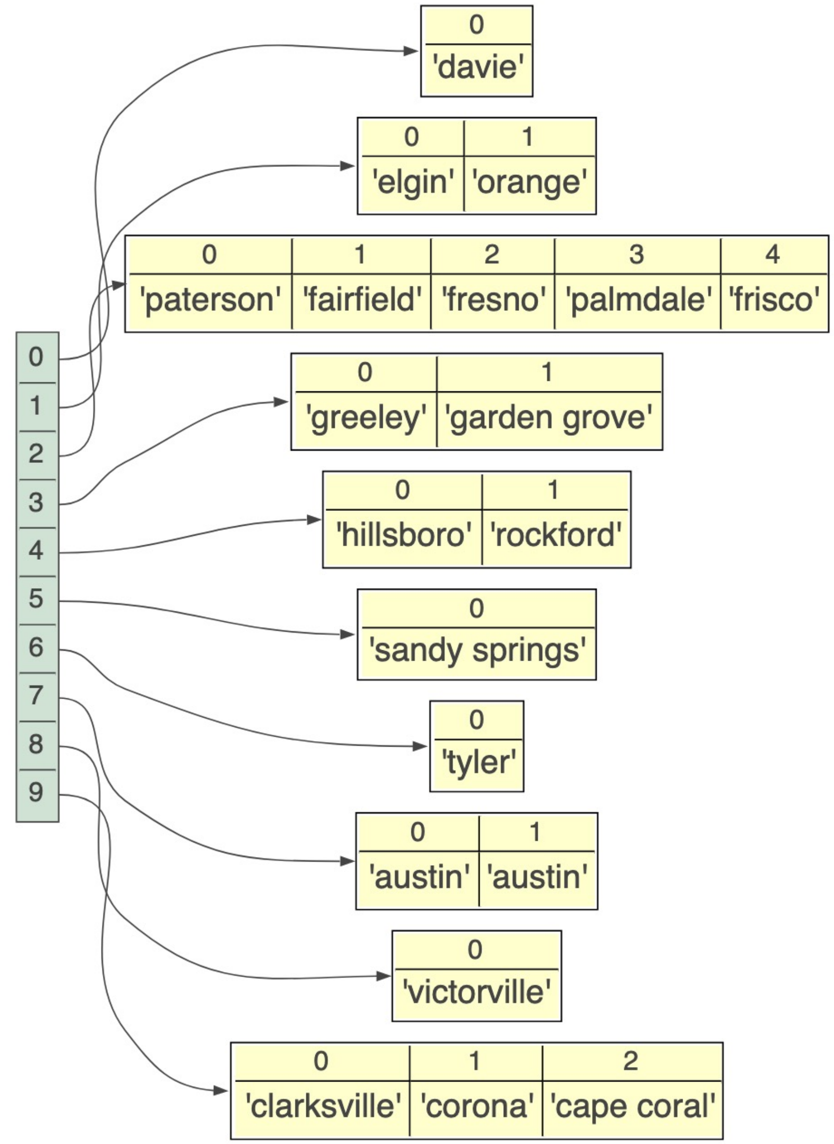
Hash vs bucket index

- Compute the hash and mod with the number of buckets we have

```python
def hash(s):
    return ord(s[0])

def htable(A):
    buckets = [[] for i in range(10)]
    for a in A:
        # fit in 10 buckets
        b = hash(a) % 10
        buckets[b].append(a)
    return buckets
```

Previously: `buckets[hash(a)].append(a)`



How much faster are hash tables?

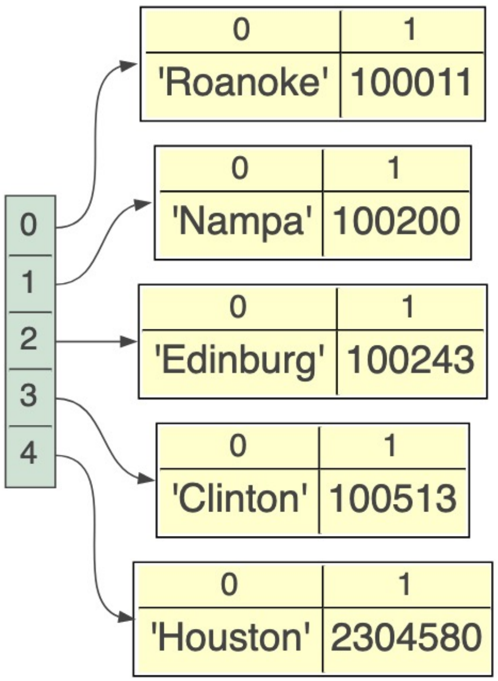
- With a uniform distribution, we would expect roughly N/B associations in each bucket for B buckets and N total elements in the dictionary
- A complexity of N/B is much better than N and, with sufficiently large B, we would say that N/B approaches 1, giving complexity $O(1)$ versus $O(n)$

Dictionary implementations

A simple dictionary implementation

- Let's represent the set of key→value pairs as a list of tuples:

```
pop = [
    ('Roanoke', 100011),
    ('Nampa', 100200),
    ('Edinburg', 100243),
    ('Clinton', 100513),
    ('Houston', 2304580)
]
```



- The key operation is to look up a value by key
- How would you implement this?

Linearly search through the list of tuples and compare the first value and the tuple to the key of interest; return the associated value if key is found

Linear lookup

- Looking for a key is a simple matter of examining the first element of every tuple stored in the list
- Return None if the key is not found

```python
def llookup(A,x):
    for k,v in A:
        if k==x:
            return v
    return None
```

```python
llookup(pop, 'Clinton')
```

```
100513
```

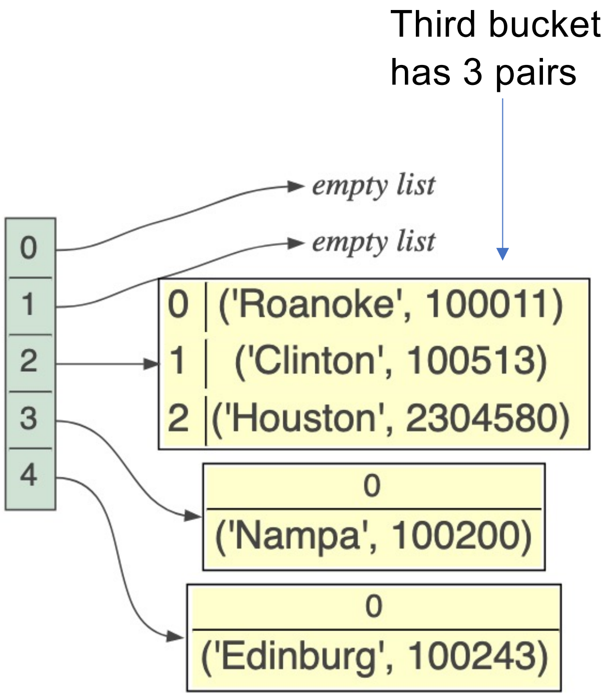
```python
llookup(pop, 'SF')
```

Hashtable dictionary implementation

- Split pairs into, say, 5 buckets (use our string hash function)

```python
def hash(s):
    return ord(s[0])

def htable_dict(A,nbuckets):
    buckets = [[] for i in range(nbuckets)]
    for k,v in A:
        b = hash(k) % nbuckets
        buckets[b].append((k,v))
    return buckets
```



Hashtable key look up

- Compute the hash, modulo the number of buckets, to get the bucket index
- Linear search within the bucket
- If key found, return value
- Else return None

```
def hlookup(buckets,x,nbuckets):
    i = hash(x) % nbuckets
    for k,v in buckets[i]:
        if k==x:
            return v
    return None
```

```
buckets = htable_dict(pop, 3)
```

```
hlookup(buckets, 'Clinton', nbuckets=3)
```

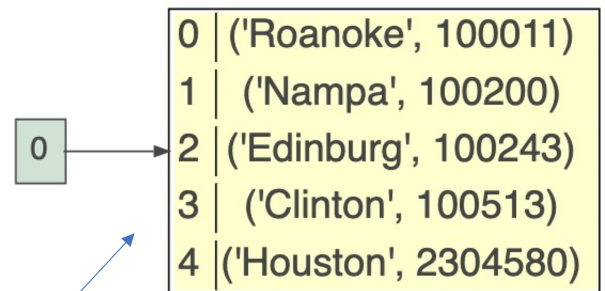
```
100513
```

```
hlookup(buckets, 'SF', nbuckets=3)
```

Degenerate case of one bucket

- With only one bucket, all pairs hash to the same bucket, which means doing a linear search of all elements to look up a key

```
buckets = htable_dict(pop, 1)
```



One and only bucket

Some details relevant to the search project

Review: tuples

- A tuple is an *immutable* list and uses parentheses rather than square brackets for notation
- Tuples are often used to group related elements:

```python
me = ('parrt',607)
userid,office = me
print(userid)
print(office)
print(me[0], me[1])
```

```
parrt
607
parrt 607
```

```python
me = ('parrt', 607)
me[1] = 525 # change office
```

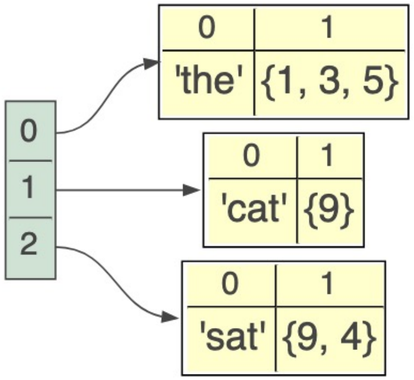
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-e02b2267f45c> in <module>
      1 me = ('parrt', 607)
----> 2 me[1] = 525 # change office

TypeError: 'tuple' object does not support item assignment
```

Values can be anything including sets

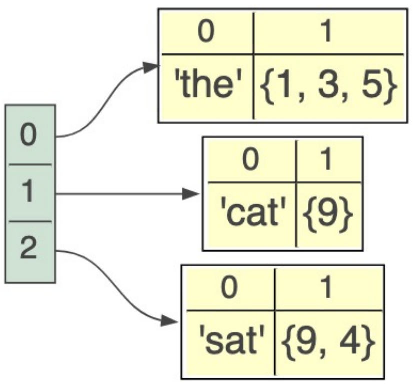
- The tuples used to represent key and value pairs are immutable, but the pair's value can point at a mutable data structures such as a set
- Consider a simple list of tuples implementation that maps words to sets of integers

```
words = [('the', {3,1,5}), ('cat',{9}), ('sat',{4,9})]
```



Modifying dictionary set values

- If you extract a mutable value from a data structure, you can modify it without having to delete and add an updated version



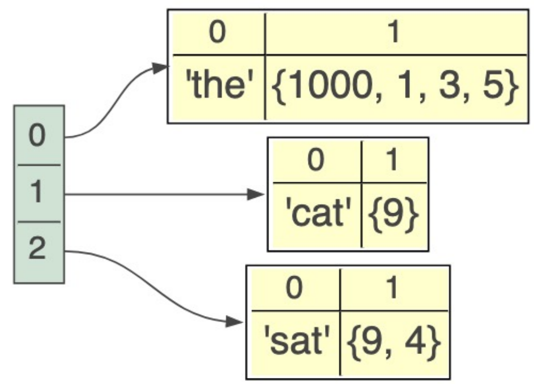

```
the = llookup(words, 'the')
the
```

```
{1, 3, 5}
```

```
the.add(1000)
```

```
llookup(words, 'the')
```

```
{1, 3, 5, 1000}
```



Summary

- Dictionaries and sets are typically implemented with a form of hashtable because the key lookup operation is so much faster
- The speed comes from a partitioning of the search space into a large number of small regions, which are searched linearly
- If we make enough buckets so that at most there are three keys in each bucket, lookup takes three operations no matter how many keys have been added to the dictionary