Hash Tables

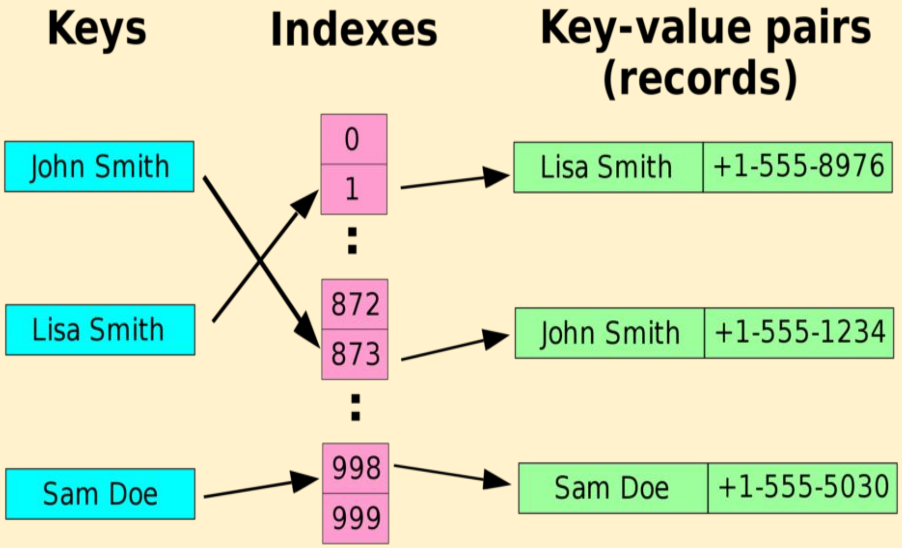
Presenters : Cassidy Newberry, Chahak Sethi, Amanda Li Luo, Chandan Nayak

Presentation Overview

- Introduction to hash tables
- Implementation of hash tables
    - Example data
    - Hashcode function
    - Put function
    - Get function
- Time complexity discussion
- Questions

Introduction

- What is a hash table?
- How are hash tables useful?
- How are hash tables implemented?
- Example

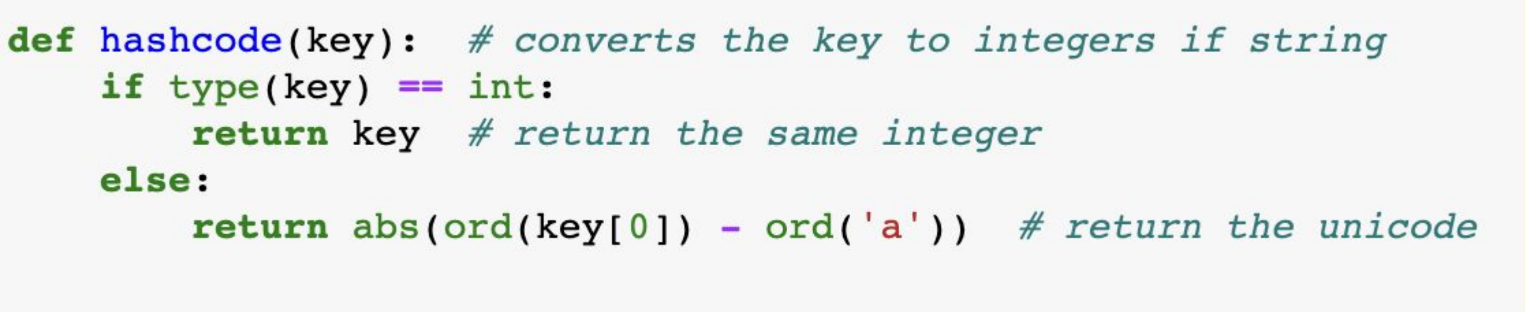


Implementation: Example Data

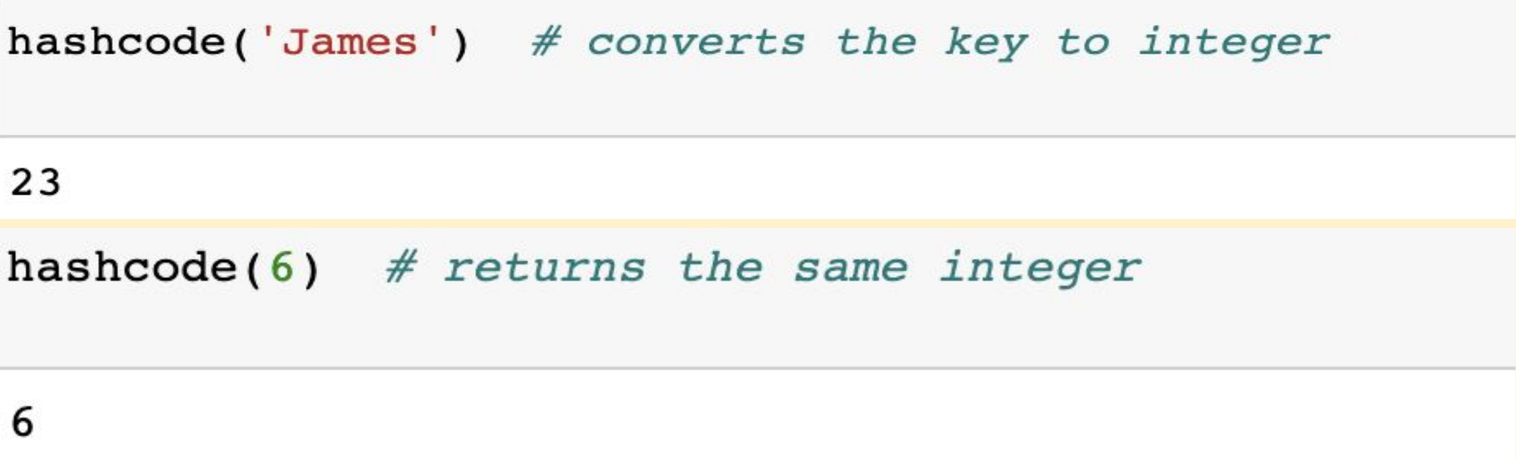
- 10 key, value pairs of Name & ID
- AIM: given a name, search for IDs

Name (Key)	ID (Value)
James	4700229
Robert	4455696
John	4453807
Michael	2454407
William	2335792
David	2084043
Richard	2038798
Joseph	3196385
Patricia	1558407
Linda	1448303
Elizabeth	1397635
Barbara	1103569
Susan	1046322
Sarah	991910
Karen	986057
Nancy	966867
Karen	986057
Nancy	966867

Implementation: Hashcode Function



```
def hashcode(key):  # converts the key to integers if string
    if type(key) == int:
        return key  # return the same integer
    else:
        return abs(ord(key[0]) - ord('a'))  # return the unicode
```



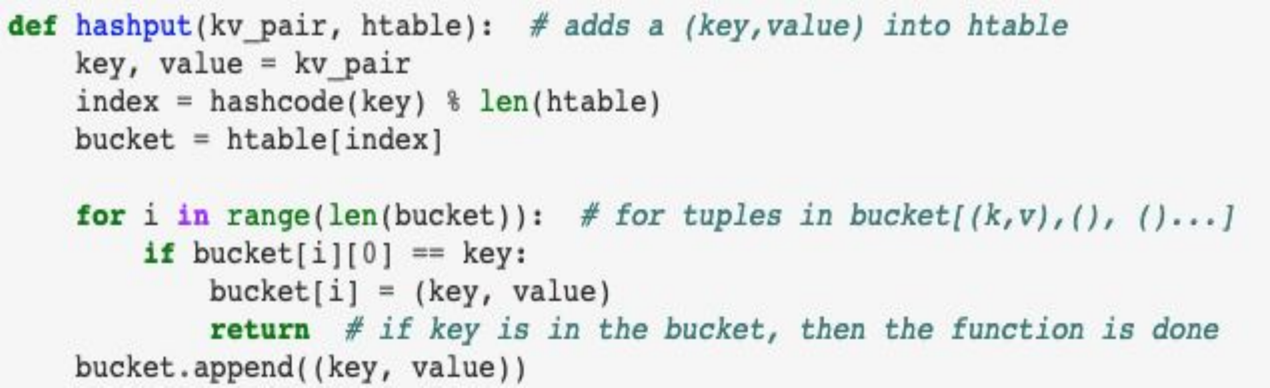
```
hashcode('James')  # converts the key to integer
```

```
23
```

```
hashcode(6)  # returns the same integer
```

```
6
```

Implementation: Put Function

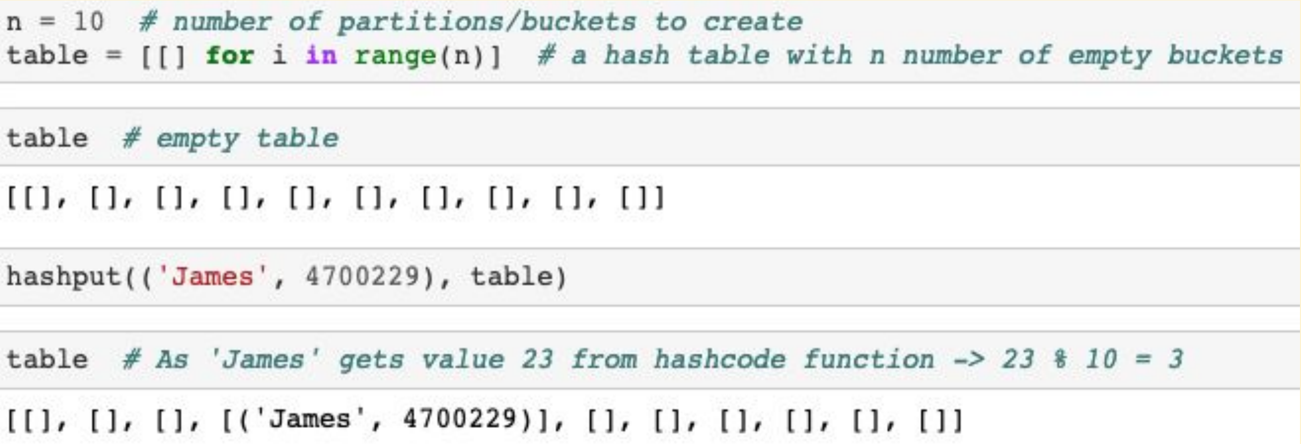


```python
def hashput(kv_pair, htable):  # adds a (key,value) into htable
    key, value = kv_pair
    index = hashcode(key) % len(htable)
    bucket = htable[index]

    for i in range(len(bucket)):  # for tuples in bucket[(k,v),(), ()...]
        if bucket[i][0] == key:
            bucket[i] = (key, value)
            return  # if key is in the bucket, then the function is done
    bucket.append((key, value))
```

Example:

- Putting in ('James', 4700229)



```python
n = 10  # number of partitions/buckets to create
table = [[] for i in range(n)]  # a hash table with n number of empty buckets
```

```python
table  # empty table
```

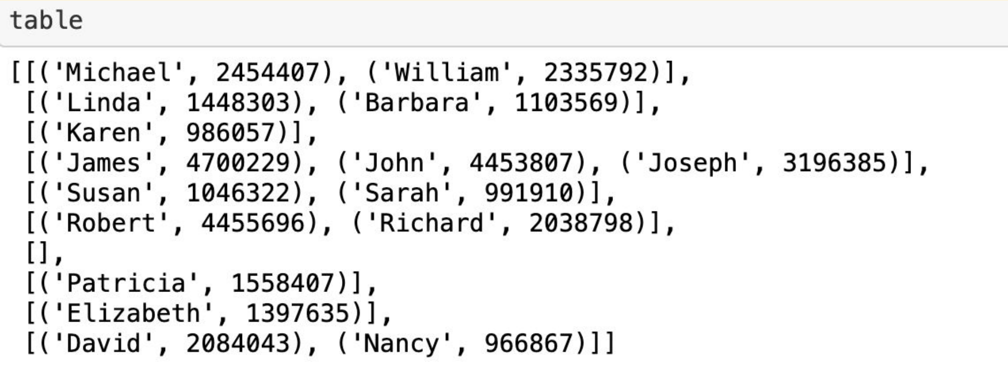
```
[[], [], [], [], [], [], [], [], [], []]
```

```python
hashput(('James', 4700229), table)
```

```python
table  # As 'James' gets value 23 from hashcode function -> 23 % 10 = 3
```

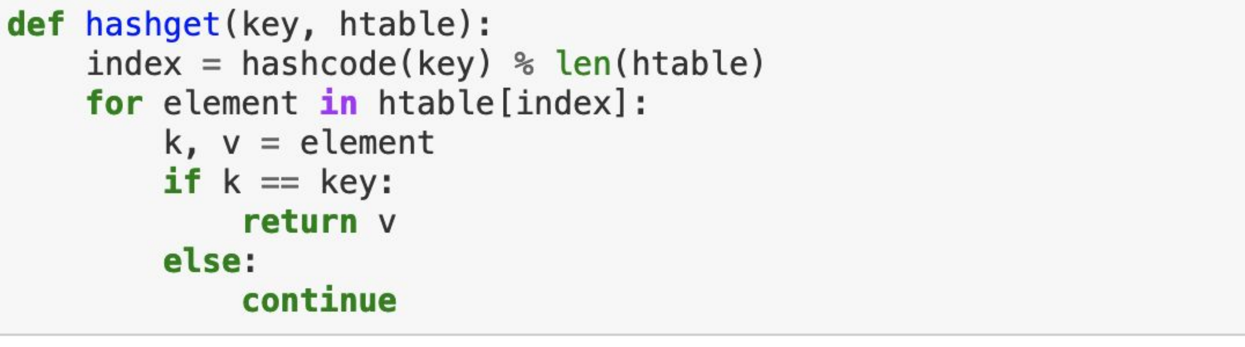
```
[[], [], [], [('James', 4700229)], [], [], [], [], [], []]
```

Implementation: Get Function



```
table
```

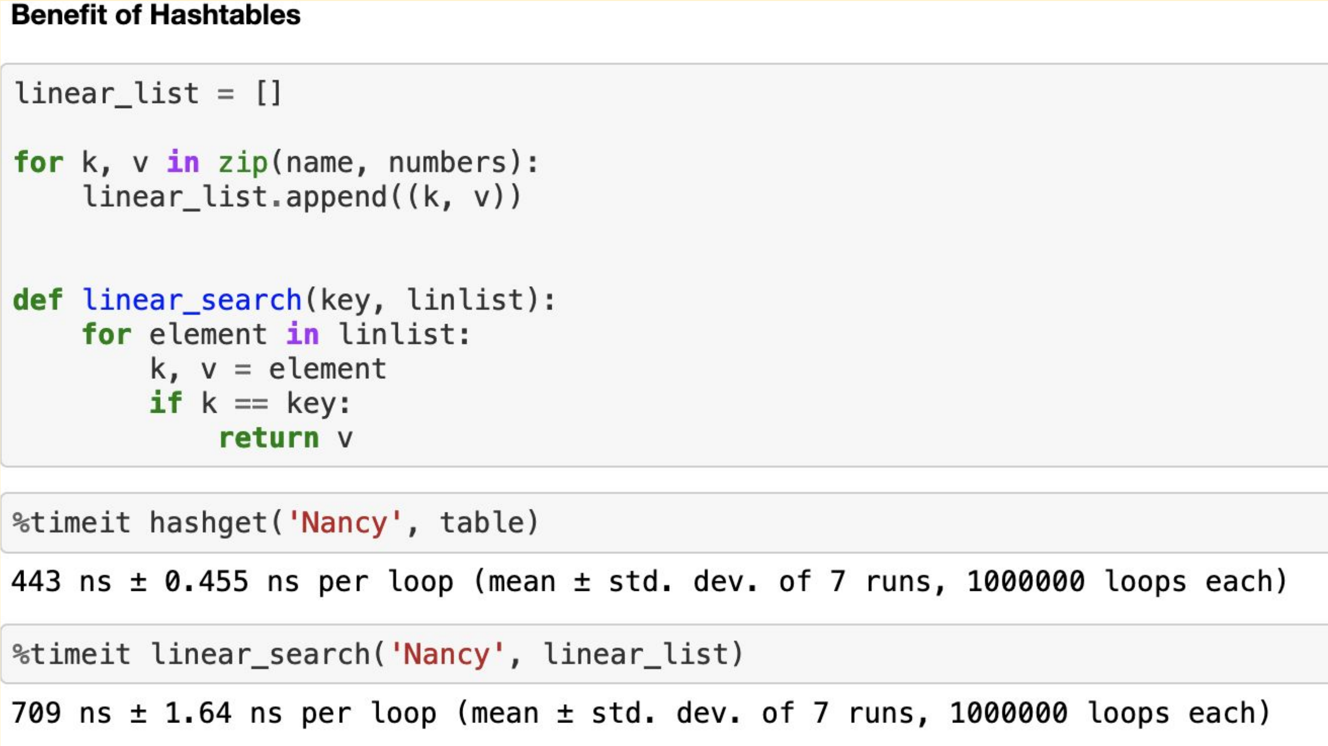
```
[[('Michael', 2454407), ('William', 2335792)],
 [('Linda', 1448303), ('Barbara', 1103569)],
 [('Karen', 986057)],
 [('James', 4700229), ('John', 4453807), ('Joseph', 3196385)],
 [('Susan', 1046322), ('Sarah', 991910)],
 [('Robert', 4455696), ('Richard', 2038798)],
 [],
 [('Patricia', 1558407)],
 [('Elizabeth', 1397635)],
 [('David', 2084043), ('Nancy', 966867)]]
```



```python
def hashget(key, htable):
    index = hashcode(key) % len(htable)
    for element in htable[index]:
        k, v = element
        if k == key:
            return v
        else:
            continue
```

Time Complexity Discussion

- Hashtables: O(n/b); b = # buckets. When b->n, then we get close to constant time access.



Benefit of Hashtables

```
linear_list = []

for k, v in zip(name, numbers):
    linear_list.append((k, v))


def linear_search(key, linlist):
    for element in linlist:
        k, v = element
        if k == key:
            return v
```

```
%timeit hashget('Nancy', table)
```

```
443 ns ± 0.455 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%timeit linear_search('Nancy', linear_list)
```

```
709 ns ± 1.64 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Conclusion

Benefits

- Almost constant time access!
- Almost constant time delete, update operations!

Potential drawbacks

- Memory usage - maintaining the index consumes additional space, might not be optimal for resource constrained use cases.

Questions