15-110 Principles of Computing – F19

Lecture 19: Dictionaries

Teacher:
Gianni A. Di Caro

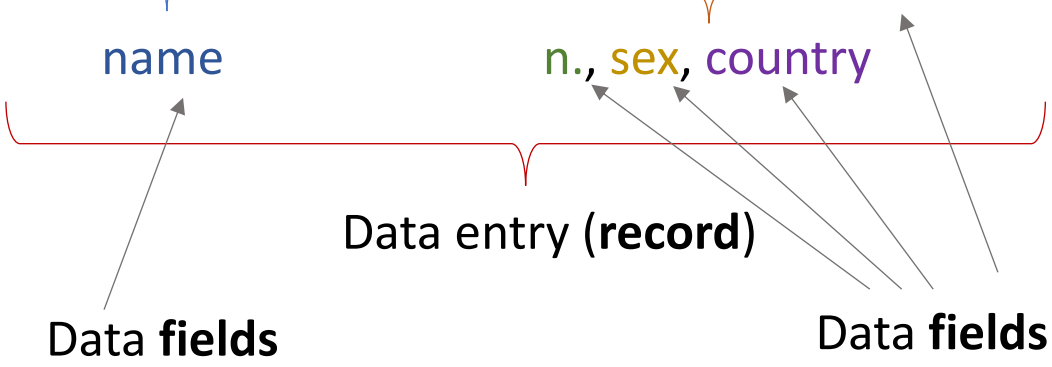
جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Storing and Manipulating structured data

- So far we have used lists to store and manipulate **structured data**

- Example: data about people, including account number, sex, country of origin

```
accounts = [ ['J. Smith', [35672, 'M', 'USA']],
             ['M. Saleh', [27623, 'M', 'Jordan']],
             ['F. Dupont', [17623, 'F', 'France']] ]
```



Storing and Manipulating structured data

- So far we have used lists to store and manipulate **structured data**

- Example: data about animals, including name, phylum, class, order

```
animals = [ ['tiger', 'vertebrate', 'mammal', 'carnivore'],
            ['orangu-tan', 'vertebrate', 'mammal', 'primate'],
            ['falcon', 'vertebrate', 'bird', 'falconiformes'] ]
```

- Example: data about countries, including name, population, GDP per capita, S&P's rating

```
countries = [ ['USA', 324459463, 59792, 'AAA'], ['Switzerland', 8476005, 80637, 'AAA'],
              ['Qatar', 2639211, 61024, 'AA-'], ['Luxembourg', 583455, 105863, 'AAA'] ]
```

Manipulating structured data

➢ How do we access and modify these type of data?

```
accounts = [ ['J. Smith', [35672, 'M', 'USA']],
             ['M. Saleh', [27623, 'M', 'Jordan']],
             ['F. Dupont', [17623, 'F', 'France']] ]
```

❑ Common queries / manipulation actions include:

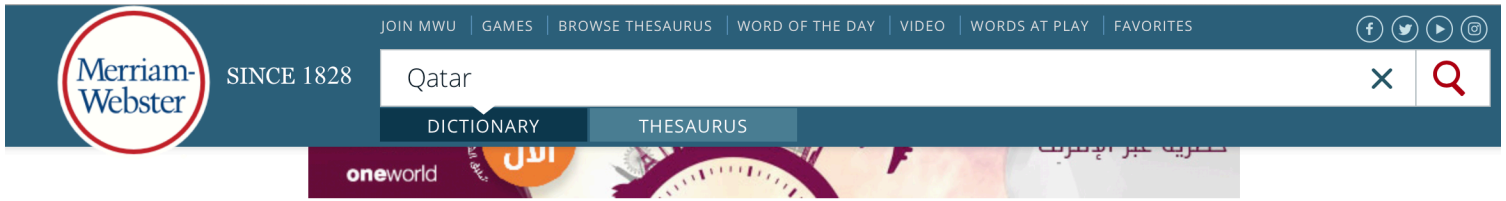
- **Get** the data of a *specific person* (e.g., Get all data about J. Smith)
- **Modify** the data of a *specific person* (e.g., Change the account of F. Dupont)
- **Get** the data of the citizens of a *specific country* (e.g., Get all data of USA citizens)

➢ No built-in method does directly the job, we need to write our own function to retrieve needed data ☹

❖ *Idea:* we need to provide a **search key** (e.g., 'J. Smith') and retrieve the **associate data**

Dictionary data structure

- ✓ Don't we have a more structured / built-in way to provide a **search key** and retrieve the **associate data?**
- ✓ Or, more in general**, to label data and access / search data using labels?**



Collection of *data resources* that can be accessed through specific keyword identifiers (e.g., Qatar)

Qatar geographical name

Qa·tar | \ ˈkä-tər, ˈgä-, ˈgə-; kə-ˈtär\

Definition of *Qatar*

country in eastern Arabia on a peninsula projecting into the Persian Gulf; an independent emirate; capital Doha *area* 4400 square miles (11,395 square kilometers), *population* 1,699,435

Other Words from *Qatar*

Qatari \ kə-ˈtär-ē, gə-\ *adjective or noun*

Collection of *pairs* of:

<key>, <data>

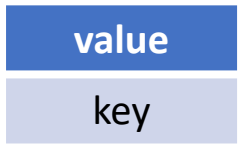
Definition of *dictionary*

1 : a reference source in print or electronic form containing words usually alphabetically arranged along with information about their forms, pronunciations, functions, etymologies, meanings, and syntactic and idiomatic uses

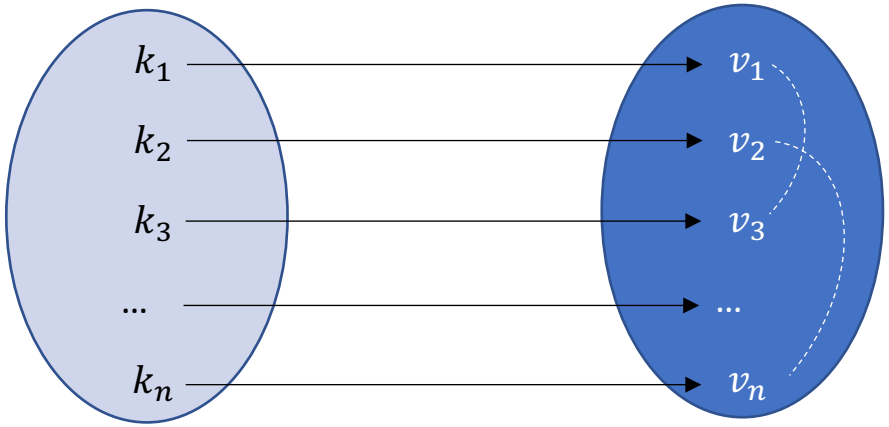
2 : a reference book listing alphabetically terms or names important to a particular subject or activity along with discussion of their meanings and applications

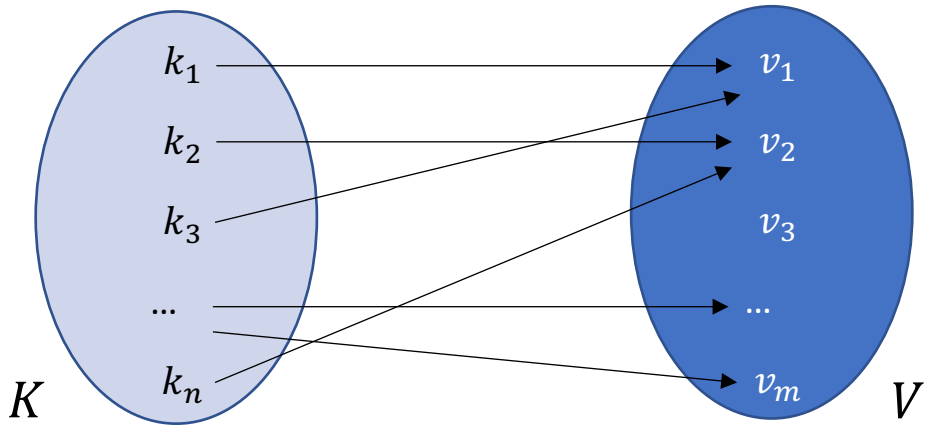
3 : a reference book listing alphabetically the words of one language and showing their meanings or translations in another language

Dictionary data structure: maps (associative, surjective)



key ↦ value





- A dictionary **maps** n keys into n values
- Keys are <u>all different / unique</u>
- Different keys might be associated to a <u>same value</u> (representing however *physically different data records*)

- In the example, the value v_1 associated to key k_1 is the same as the value v_3 associated to key k_3 (as shown by dashed lines), however they are physically different items
- E.g., k_1 = "John", k_3= "Ann", and they have the same age v_1 = 20, v_3 = 20

- Accounting for values that can be the same, we can represent a dictionary map as a **surjective map** in *mathematics*, where each element in the value set V (of size m) is associated to (is the co-image of) *at least* one element from the key set K (of size $n \geq m$), and all elements in K are associated to one element in V

$$dict : K \mapsto V$$

Dictionary data structure: associative maps

key ↦ value

Examples:

- SSNs ↦ Person information data
- Names ↦ phone numbers, email
- Usernames ↦ passwords, OS preferences
- ZIP codes ↦ Shipping costs and time
- Country names ↦ Capital, demographic info
- Sales items ↦ Quantity in stock, time to order
- Courses ↦ Student statistics
- Persons ↦ Friends in social network
- Animals ↦ Classification data
- Companies ↦ Rate, capital, investments
- …

- In all the examples, a **unique label** (***key***) can be associated to a (more or less complex) **piece of data** (the ***value***)
- This motivates the choice of a *dictionary data structure* to represent and manipulate these type of data

Dictionary data structure

- **Data type**: `dict`
- **Syntax:**

```
{ key_1: value_1,  key_2: value_2,  key_3: value_3 }
```

`dict` literal object with three elements

```
d = { key_1: value_1,  key_2: value_2,  key_3: value_3 }
```

definition of a `dict` variable `d` with three elements

```
d = { }
```

definition of an empty `dict` variable `d`

Separator between key-value entries

Key(word) identifier

Data value (information data) associated to the key

Separator between key and value

Delimiters for literal object definition

Dictionary data structure: unordered, associative array (map)

- **Unordered:** it's *not a sequence*, rather a collection, where items are accessed through the keys, not by their position in a sequence

```
x = [20, 22, 29, 20]
```

Value	20	22	29	20
Position index	0	1	2	3

List

✓ A sequence type accesses values by their position in the sequence, i.e., values are *sequentially indexed*

index ↦ value

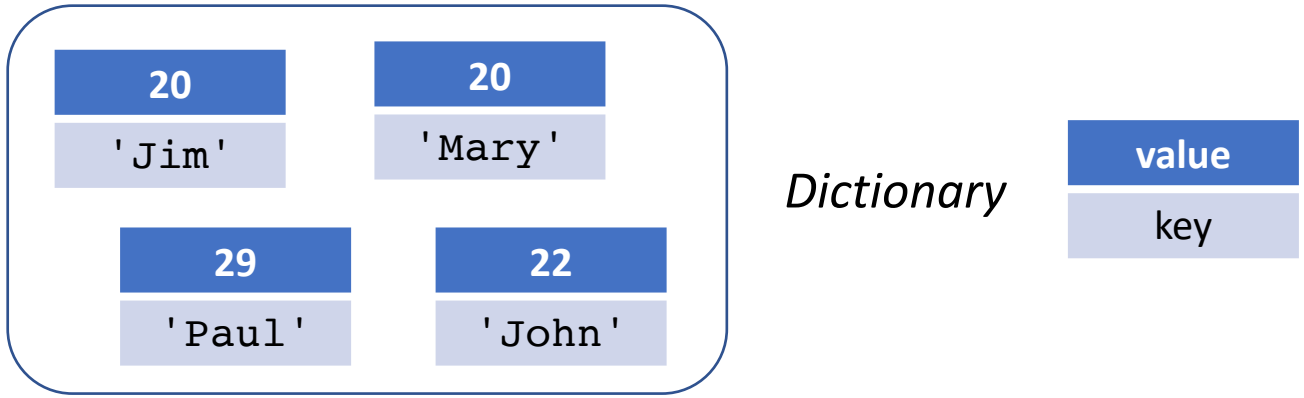
`x[1]` is the value of `x` at position 1, which is 22

✓ A dictionary represents data values by *using key labels*, and then accesses values by their keys, i.e., *associates* key labels to values (associative memory):

key ↦ value

Dictionary data structure: unordered, associative array (map)

```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```

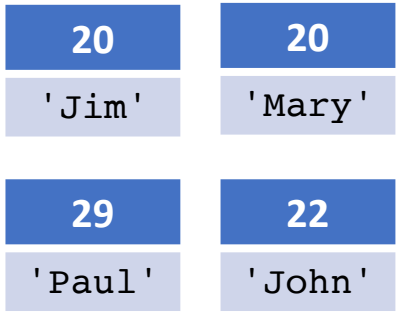


- `d['John']` is the value of **associated** to the keyword `'John'`, which is 22
- `d[1]` throws an error: there's no a key with value 1 in the dictionary

Dictionary data structure: non-scalar, mutable

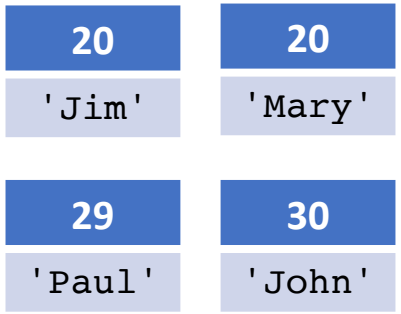
- **Non-scalar**: it's a composite data type, it has *internal structure*
- **Mutable:** values of dictionary's entries can be *updated* and items can be added and deleted (without changing dictionary identity), *aliases* can be created between variables

```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```



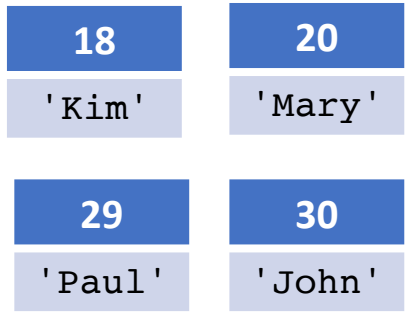
✓ Update value of existing keys

```
d['John'] = 30
```



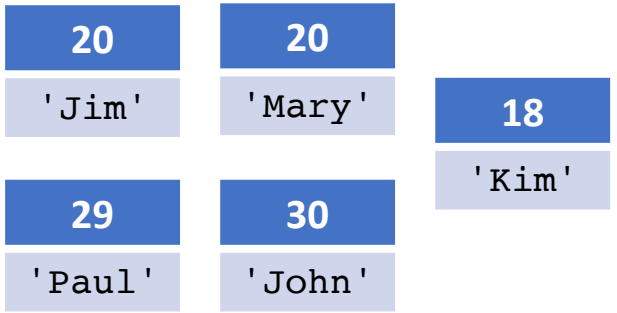
✓ Delete an existing item:

```
del d['Jim']
```



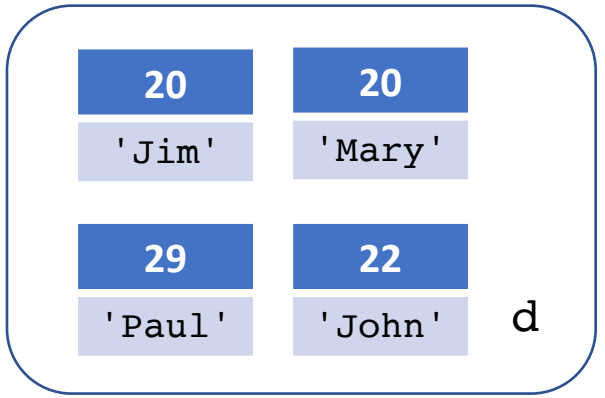
✓ Add a new key-value pair:

```
d['Kim'] = 18
```



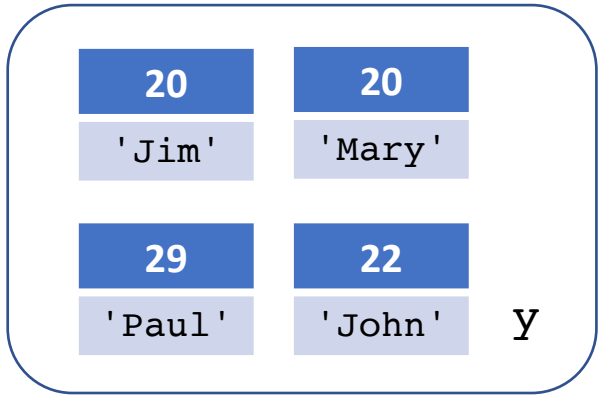
Dictionary data structure: mutable

```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```



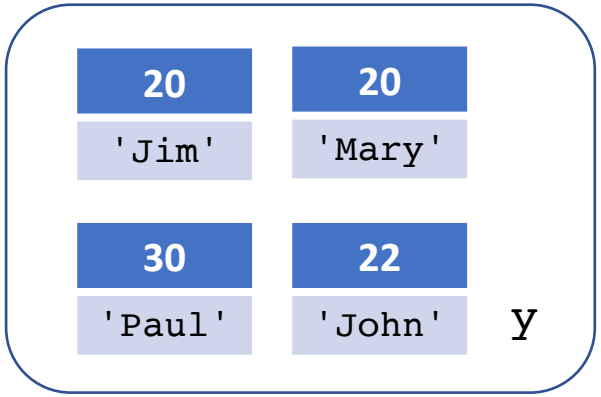
✓ Create an alias:

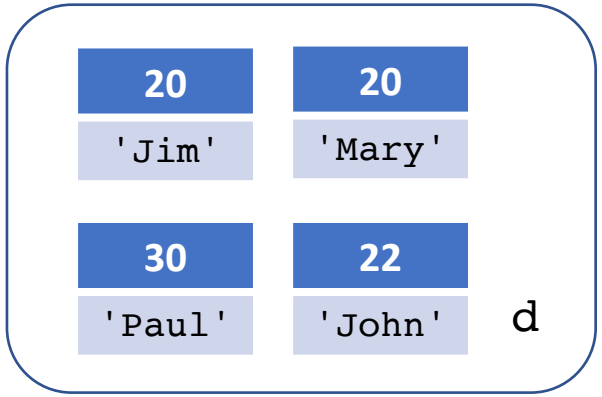
```
y = d
```



- Changing `y` changes `d` and vice versa:

```
y['Paul'] = 30
```





- The two dictionaries have the same identity:

`id(y)` is the same as `id(d)`

Restrictions and freedom on data types for keys and values

key ↦ value

- A **key** can only contain **immutable** data types: `int, float, bool, str, tuple`
- A **value** can be of **any type**
- **Keys and values** of the same dictionary can be of any (allowed) mixed type

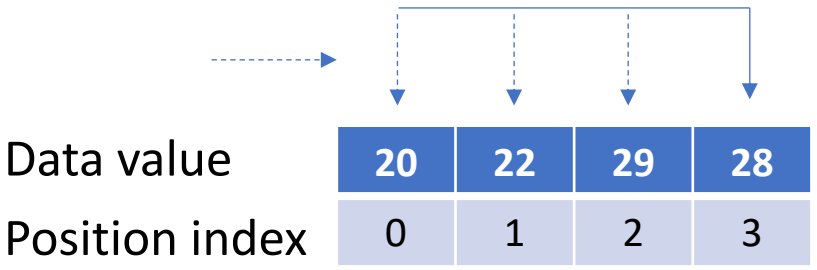
```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```

- ✓ `d[12] = 'New data value'`
- ✓ `d['Jim'] = True`
- ✓ `d['List data'] = [ [1,2,3], [4,5,6] ]`
- ✓ `d[(2,3)] = 7.8`
- ❖ `d[[2,3]] = 'Incorrect'`
- ❖ `d[([1,3], 2)] = 'Incorrect'`

Why do we need an associative data structure?

- ✓ Because by using labels we can access to values much more **efficiently** than with lists, for instance
  - Dictionaries are in fact **hashed** data types, while lists (sequences) are *indexed* data types

`value in data_list ?`

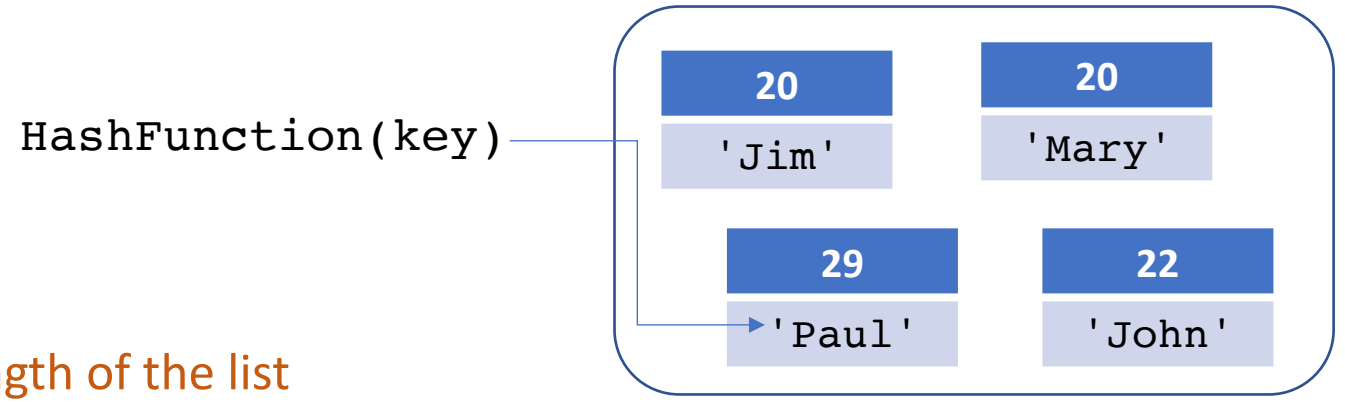



```
x = [20, 22, 29, 28]
```

Worst-case search time **linear** with the length of the list

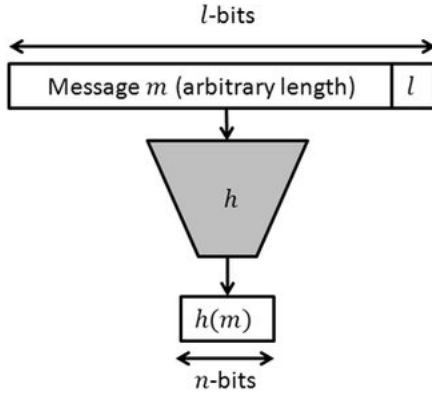
`key in dictionary ?`

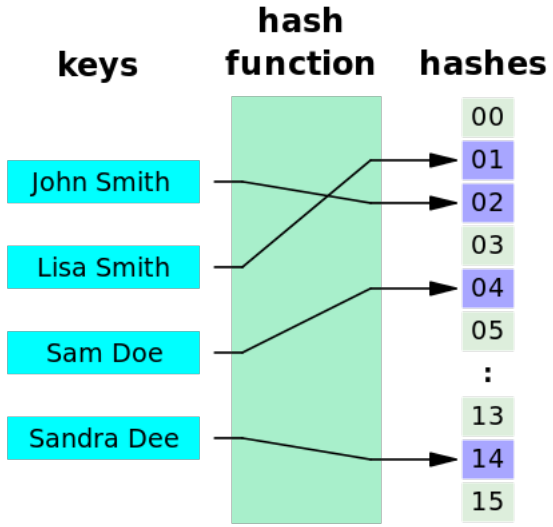
```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```

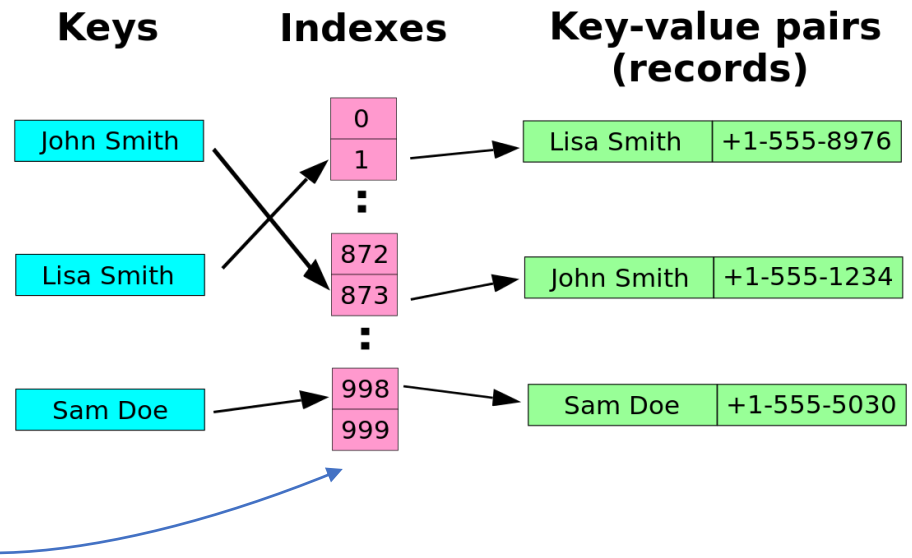


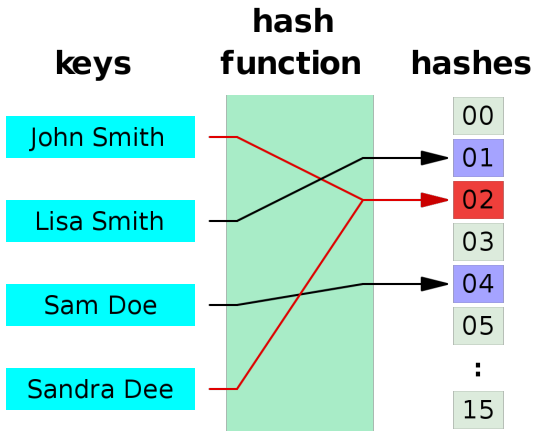
Constant search time
(independent of dictionary size)

Dictionary as hashed data type









- **Computation time:**
  Time to compute the hash
  + one direct memory access

- **Insert a new pair** `(key, value)` to a dictionary:
  - ✓ `hash(key)` computes an *index*
  - ✓ The pair `(key, value)` is stored directly at the *index* location

- **Access the value associated to key:**
  - ✓ `hash(key)` gives the *index*
  - ✓ The pair `(key, value)` at the *index* location is returned directly

Functions and operators for inspecting a dictionary

```
accounts = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
            'F. Dupont': [17623, 'F', 'France'] }
phone_numbers = {'Ann': 5461, 'Paul': 5472, 'Mark': 3541, 'Liz': 2451}
numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
```

- Count all the key-value items present in the dictionary: `len(dictionary)` function

  `len(accounts)` → `3` (int type)

  `len(numbers)` → `6` (int type)

- Get the list with the keys present in the dictionary: `list(dictionary)` function

  `list(accounts)` → `['J. Smith', 'M. Saleh', 'F. Dupont']` (list type)

  `list(phone_numbers)` → `['Ann', 'Paul', 'Mark', 'Liz']` (list type)

  `list(numbers)` → `[1, 2, 3, 4, 5, 6]` (list type)

- Check whether a key exists or not in the dictionary: membership operators `in`, `not in`

  `3 in numbers` → `True` (bool type)

  `'Jim' not in phone_numbers` → `True` (bool type)

Methods for inspecting a dictionary: `keys()`

- Get a *dynamic view* on the dictionary keys: `dict.keys()` method, returns a **view object**

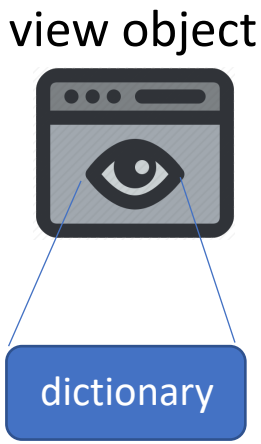
```
numbers.keys()    →  dict_keys([1, 2, 3, 4, 5, 6])    (view object)
```

vs.

```
list(numbers)  → [1, 2, 3, 4, 5, 6]                   (list object)
```

→ The keys() method doesn't return a *physical list* with the current keys, as list() does, instead it provides a **view object,** a window view on the dictionary which is dynamically kept up-to-date

- ✓ Save memory
- ✓ If things changes in the dictionary, these can be seen through the view object



Methods for inspecting a dictionary: `keys ( )`

```
numbers = {1: 'p', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r’}
keys_now_in_dict = list(numbers)
keys_view = numbers.keys()
numbers[13] = 'p'
print("Is 13 in dict? From static list copy:", (13 in keys_now_in_dict) )
print("Is 13 in dict? From dynamic view:", (13 in keys_view) )
```

Methods for inspecting a dictionary: `values()`, `items()`

- Get a *dynamic view* on the dictionary **values** : `dict.values()` method, returns a **view object**

```
numbers.values()  →  dict_values(['p', 'p', 'p', 'r', 'p', 'r'])
```

- Get a *dynamic view* on the **entire dictionary**: `dict.items()` method, returns a **view object**

```
numbers.items()  → dict_items([(1, 'p'), (2, 'p'), (3, 'p'),
                               (4, 'r'), (5, 'p'), (6, 'r')])
```

Methods for inspecting a dictionary: iterations

- Iterate over all dictionary elements :

```
for k in numbers:
  print('Key:', k)

for i in numbers.items():
  print('Pair (key, value):', i[0], i[1])
```

Observations:

- A dictionary is "*identified*" by its collection of keys, this is why directly iterating over the dictionary in the first example is in practice equivalent to iterate over the keys, that are the returned sequence values
- Iterations over `dict.items()` return the pairs (key, value) as tuples, where the key has index 0 and the value has index 1

Relational and arithmetic operators for dictionaries

```
accounts = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
            'F. Dupont': [17623, 'F', 'France'] }
numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
```

- == operator: check whether two dictionary are the same, same (key , value) pairs

```
x = accounts == numbers   → False

accounts2 = accounts.copy()
x = accounts == accounts2   → True
```

- Other relational operators `>, >=, <, <=` do not apply to dictionary operands
- Arithmetic operators do not apply to dictionary operands

Methods for accessing and modifying a dictionary: pop( )

- Remove and return dictionary element associated to passed key: `dict.pop(key, <value>)`

  `key` is the key to be searched, and `value` is the value to return if the specified key is not found in the dictionary. If `value` is not passed, an error is thrown in the case `key` is not in the dictionary

```
key = 3
x = numbers.pop(key, None)
if x != None:
   print('Removed pair (', key, ':', x, ')')
```

- Advantage over the use of `del` and `[ ]` operators:

```
key = 11
del numbers[key]
```

→ Throws an Error since the selected key is not in the dictionary!

Practice:

Implement the function `add_pair(k, v, d)` that returns the dictionary `d` modified such that the key `k` is associated with value `v`. If the key is already in the dictionary, its value may be modified. Otherwise, a new key needs to be added to the dictionary.

```
def add_pair(k, v, d):
    d[k] = v
    return d
```

Practice:

Implement the function `is_key_in(k, d)` that returns `True` if the key `k` is in the dictionary `d`, or `False` otherwise.

```
def is_key_in(k, d):
    return k in d
```

Practice:

Implement the function `is_value_in(v, d)` that returns `True` if the value `v` is in the dictionary `d`, or `False` otherwise.

```
def is_value_in(v, d):
    for k in d:
        if d[k] == v:
            return True
    return False
```

Practice:

Implement the function `get_value(k, d)` that returns the value associated with key `k` in the dictionary `d`, if it exists. If the dictionary does not contain such key, return `None`.

```
def get_value(k, d):
    if k in d:
        return d[k]
    else:
        return None
```

Practice:

Implement the function `get_key(k, d)` that returns a key which contains value `v` in the dictionary `d`, if it exists. If the dictionary does not contain a key with this value, return `None`.

```
def get_value(k, d):
    if k in d:
        return d[k]
    else:
        return None
```

Practice:

Implement the function `count(l)` that takes a list and returns a dictionary where the keys are elements of the list and the values are the number of times that element occurred in the list.

For example, count(['a','b','b','a','c','b'])
should return the dictionary: {'a': 2, 'b': 3, 'c': 1}.

```
def count(l):
    d = {}
    for e in l:
        if e in d:
            d[e] += 1
        else:
            d[e] = 1
    return d
```

Practice:

Implement the function `get_middle(d)` that takes a dictionary and returns value of the middle key (if the dictionary was sorted).

For example, `get_middle({'b': 5, 'a': 3, 'c': 1})` should return 5.

```
def get_middle(d):
    items = d.items()
    items = sorted(items)
    middle = items[len(items)//2]
    return middle[1]
```

Methods for accessing and modifying a dictionary: `get()`

- Get the value for a specified key if key is in dictionary: `dict.get(key, <value>)`

  `key` is the key to be searched, and `value` is the value to return if the specified key is not found in the dictionary. The `value` parameter is optional. If `value` is not passed, `None` is returned.

```
key = 3
x = numbers.get(key)
if x != None:
   print('Value associated to key', key, 'is:', x)
```

- Advantage over the use of the `[]` operator:

```
x = numbers[key]
```

→ Throws an Error if key is not in the dictionary!

Methods for accessing and modifying a dictionary: `popitem()`

- Remove and return the last inserted dictionary element: `dict.popitem()`

  A pair (key, value) is removed from the dictionary following a LIFO order (last-in, first-out). The removed pair is returned as a tuple

```
x = numbers.popitem()
if len(numbers) > 0:
    print('Removed the last inserted key-value pair (', x, ')')
    print('New size of the dictionary:', len(numbers))
```

Methods for accessing and modifying a dictionary: `clear()`

- Remove *all* elements from a dictionary element: `dict.clear()`

  All elements are removed, no values are returned, after the call `dict` is equivalent to `{}`

```
numbers.clear()
print('Removed all elements')
```

Methods for accessing and modifying a dictionary: `update( )`

- Update the dictionary with the elements from the another dictionary object (or from an iterable of key/value pairs (e.g., tuple)): `dict.update([other])`

  Takes as input a dictionary (or an iterable of tuples) and use the input to update `dict`

```
some_primes = {13:'p', 17:'p', 23:'p'}
numbers.update(some_primes)
print('Updated dictionary:', numbers)

new_entry= {12:'r'}
numbers.update(new_entry)
print('Dictionary updated with a new single entry')

l = [(10, 'r'), (12, 'r')]
numbers.update(l)
print('Dictionary updated with a list of entries')
```

Methods for accessing and modifying a dictionary: `setdefault()`

- Insert a new (key, value) pair only if key doesn't already exist, otherwise return the current value: `dict.setdefault(key, <value>)`

  The function aims to *update* the dictionary with a new (`key, value`) pair only if the given `key` is not already in the dictionary, otherwise the dictionary (i.e., the existing value of `key`) is *not updated* and the current value associated to the specified `key` is returned instead. If `value` is not passed as input, the default `None` value is used.

```
val = numbers.setdefault(30, 'r')      # key 30 doesn't exist, pair (30,'r') is inserted in numbers
val = numbers.setdefault(30, 'rr')     # key 30 it exists now, its value isn't updated, val gets 'r'


new_dict = {}
for i in range(10):
    new_dict.setdefault(i)     # new_dict gets initialized with 10 keys and None values
```

Useful operations on key and value sets: `sorted()`, `sort()`

- Get the **sorted list of keys** from the dictionary items:

```
sorted_keys = sorted(numbers) → [1, 2, 3, 4, 5, 6]
sorted_keys = sorted(numbers.keys())
```

- Get the **sorted list of values** from the dictionary items:

```
sorted_values = sorted(numbers.values()) →  ['p', 'p', 'p', 'p', 'r', 'r']
```

- Equivalent way, using the `list()` function:

```
keys_to_be_sorted = list(numbers.keys())
keys_to_be_sorted.sort()
```

- Get the **sorted list of keys, paired with their associated values** :

```
sorted_dict_list = sorted(numbers.items())
                  → [(1, 'p'), (2, 'p'), (3, 'p'), (4, 'r'), (5, 'p'), (6, 'r')]
```

Useful operations on key and value sets

- Watch out! The `sorted()` function and the `sort()` method could have been used without a comparison function given that in these example all keys / values are homogeneous (`int` or `str`) and python knows how to perform comparisons among these homogeneous data types

- In the general case, the use of sort functions/methods might require the additional definition of a comparison function, based on the characteristics of the keys / values to sort

- This applies also to `min(), max(), sum()`

Useful operations on key and value sets: `min()`, `max()`, `sum()`

- Find **min / max** of key/values from the dictionary items:

```
max_key_val = max(numbers)      → 6
min_key_val = min(numbers)      → 1
max_key_val = max(numbers.keys())      → 6
min_key_val = min(numbers.keys())      → 1

max_values = max(numbers.values())   → r
min_values = min(numbers.values())   → p
```

- Find **sum** of key/values from the dictionary items:

```
key_sum = sum(numbers) → 34
key_sum = sum(numbers.keys()) → 34
values_sum = sum(numbers.values()) → Error, sum not defined over strings!
```

Creation of dictionary variables: use literals

- Empty dictionary:

```
v = {}
```

- Creation of a dictionary with literals:

```
phone_numbers = {'Ann': 5461, 'Paul': 5472, 'Mark': 3541, 'Liz': 2451}

numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}

by_name = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
           'F. Dupont': [17623, 'F', 'France'] }

by_country = {'USA': ['J. Smith', 35672, 'M'], 'Jordan': ['M. Saleh', 27623, 'M'],
              'France': ['F. Dupont', 17623, 'F'] }
```

Creation of dictionary variables: use a list of tuples

- Use a list of tuples and the built-in function `dict(key_val_list)`
  that builds a dictionary directly from *sequences* of input key-value pairs:

```
word_list = [ ('Hello', 5), ('this', 4), ('is', 2), ('a', 1), ('list', 4) ]
word_dict = dict(word_list)

parabola = dict( [(0,0), (0.5, 0.25), (1,1), (1.5, 2.25)] )

v = dict( [ ('USA', [35672, 'M', 'J. Smith']),
            ('Jordan', [27623, 'M', 'M. Saleh']),
            ('France', [17623, 'F', 'F. Dupont']) ] )
```

Creation of dictionary variables: use list of keys with default values

- Use a list of keys and assign a common optional value to the keys by using the method `fromkeys(key_list, <value>)`

```
list_of_words = ["This", "is", "a", "list", "of", "key", "strings"]
dict_of_words = dict.fromkeys(list_of_words, 0)

primes = [2, 3, 5, 7, 11, 13]
dict_of_primes = dict.fromkeys(primes, 'p')
```

Creation of dictionary variables: use of two lists

- Use two lists of the same length, one containing the keys and the other the values, and pair them using the function `zip(key_list, value_list)`

```
list_of_keys = [1, 2, 3, 4, 5, 6]

list_of_values = ['r', 'p', 'p', 'r', 'p', 'r' ]

numbers = dict(zip(list_of_keys, list_of_values))
```

Creation of dictionary variables: cloning and aliasing

- **Cloning**: make a *shallow copy* of the content of a dictionary using method `copy()`

```
new_dict_same_content = numbers.copy()
new_dict_same_content[36] = 'r'
if 36 not in numbers:
    print("Change in the new dictionary didn't affected previous dictionary")
```

- **Aliasing:** make a copy of the content of a dictionary and establish an alias

```
alias_dict = numbers
alias_dict[36] = 'r'
if 36 in numbers:
    print("Change in the new dictionary affected previous dictionary!")
```