DASC / CSE 5300

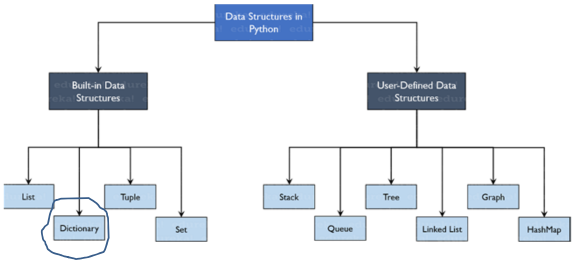
Module II

Sharma Chakravarthy
Information **T**echnology **Lab**oratory (IT Lab)
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76019
Email: sharmac@cse.uta.edu
URL: http://itlab.uta.edu/sharma

In Module II

- I will cover the following with examples so you can practice further on your own
  - Big-O complexity
  - Python data structures for analysis
  - Algorithms for data structures
  - Graphs
  - Hash table/Dictionary
  - Sets
  - Stack, Queue

Python



Data Structures in Python

Built-in Data Structures

User-Defined Data Structures

List

Dictionary

Tuple

Set

Stack

Queue

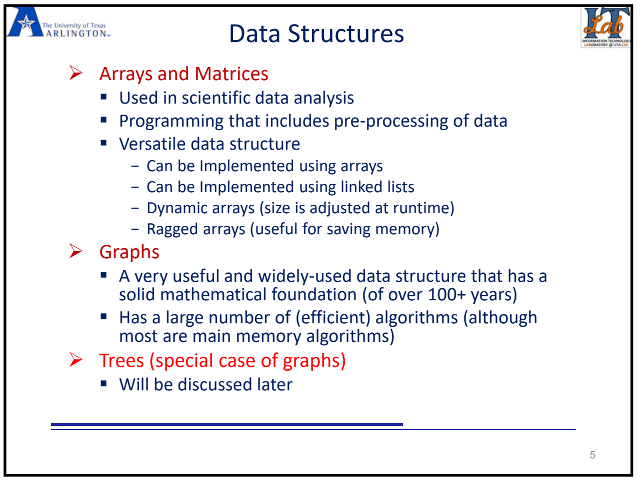
Tree

Linked List

Graph

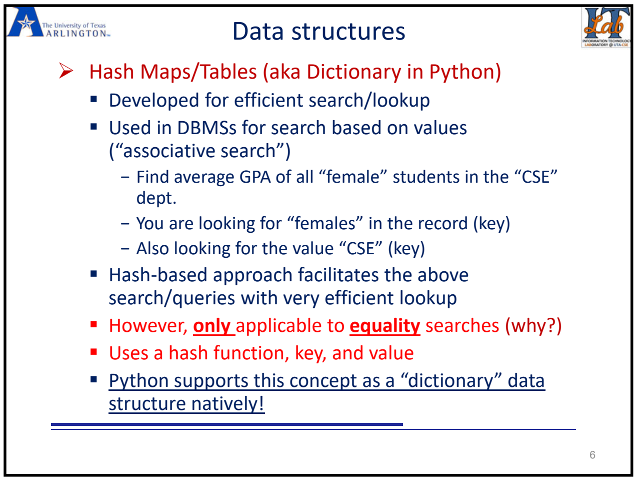
HashMap

Hash maps (Python Dictionary)



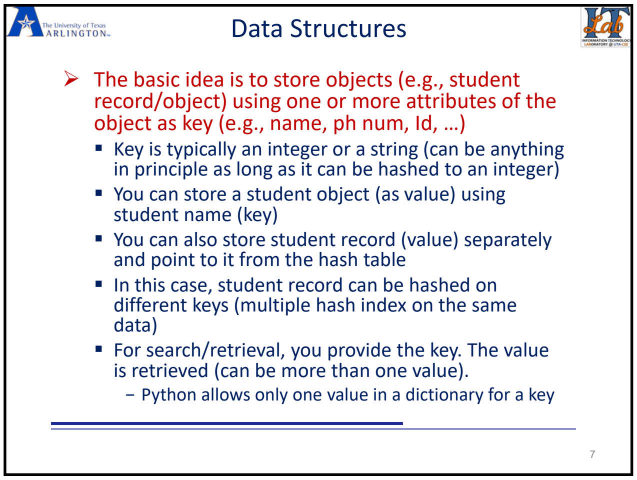
Data Structures

- Arrays and Matrices
  - Used in scientific data analysis
  - Programming that includes pre-processing of data
  - Versatile data structure
    - Can be Implemented using arrays
    - Can be Implemented using linked lists
    - Dynamic arrays (size is adjusted at runtime)
    - Ragged arrays (useful for saving memory)
- Graphs
  - A very useful and widely-used data structure that has a solid mathematical foundation (of over 100+ years)
  - Has a large number of (efficient) algorithms (although most are main memory algorithms)
- Trees (special case of graphs)
  - Will be discussed later



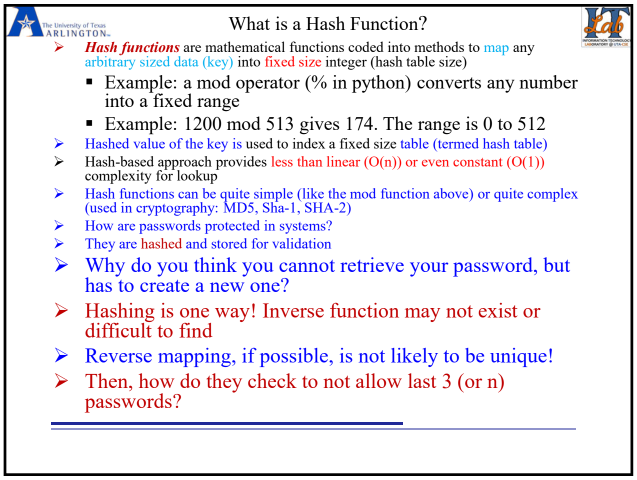
Data structures

- Hash Maps/Tables (aka Dictionary in Python)
  - Developed for efficient search/lookup
  - Used in DBMSs for search based on values ("associative search")
    - Find average GPA of all "female" students in the "CSE" dept.
    - You are looking for "females" in the record (key)
    - Also looking for the value "CSE" (key)
  - Hash-based approach facilitates the above search/queries with very efficient lookup
  - However, **only** applicable to **equality** searches (why?)
  - Uses a hash function, key, and value
  - Python supports this concept as a "dictionary" data structure natively!



Data Structures

- The basic idea is to store objects (e.g., student record/object) using one or more attributes of the object as key (e.g., name, ph num, Id, …)
  - Key is typically an integer or a string (can be anything in principle as long as it can be hashed to an integer)
  - You can store a student object (as value) using student name (key)
  - You can also store student record (value) separately and point to it from the hash table
  - In this case, student record can be hashed on different keys (multiple hash index on the same data)
  - For search/retrieval, you provide the key. The value is retrieved (can be more than one value).
    - Python allows only one value in a dictionary for a key

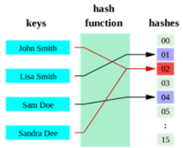


What is a Hash Function?

- ***Hash functions*** are mathematical functions coded into methods to map any arbitrary sized data (key) into fixed size integer (hash table size)
  - Example: a mod operator (% in python) converts any number into a fixed range
  - Example: 1200 mod 513 gives 174. The range is 0 to 512
- Hashed value of the key is used to index a fixed size table (termed hash table)
- Hash-based approach provides less than linear (O(n)) or even constant (O(1)) complexity for lookup
- Hash functions can be quite simple (like the mod function above) or quite complex (used in cryptography: MD5, Sha-1, SHA-2)
- How are passwords protected in systems?
- They are hashed and stored for validation
- Why do you think you cannot retrieve your password, but has to create a new one?
- Hashing is one way! Inverse function may not exist or difficult to find
- Reverse mapping, if possible, is not likely to be unique!
- Then, how do they check to not allow last 3 (or n) passwords?

Utility of a Hash Function?

- When a large range is squeezed into a small range, multiple key values will produce the same hash index/value for different keys
  - ❖ This is termed collision (of index/hash values)
- However, hash output should not be skewed! Preferably produce uniform distribution
- What happens otherwise? How is collision handled?
- There are several ways to handle collisions



This hash function shows "Lisa Smith" mapped to the value 01, "Sam Doe" to 04, but both "John Smith" and "Sandra Dee" map to the same value 02, which is a collision

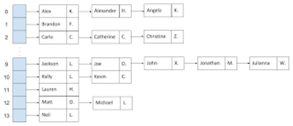
When different key values map to the same hash value those are collisions which need to be resolved

Handling Collisions

- When two (or more) keys hash to the same hash value, there is a collision
- This can not be completely eliminated and usually becomes more common when either the hash function calculation is not appropriate for the data or when the table size is too small for the number of items to be hashed, that results in a heavily loaded table
  - linear probing (simplest)
    - Keep searching from the hash index until you find an empty slot and put it there
  - Rehashing (or double hashing) and use it as an increment (generalizes linear hashing)
    - Take the hashed value and either go through the same hash function again or on the second round use a different hash function and use it as an increment
  - Quadratic hashing
    - After the first hash, use a quadratic function to determine the increment
  - Buckets and lists
    - Can accommodate duplicates easily
- Each has different properties with respect to clustering and computation tradeoff

Linear probing

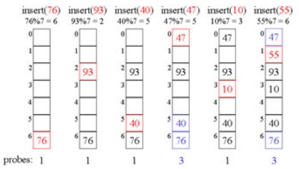
- Starts from the hash index and searches forward (step as 1) to find an empty slot



When inserting data and there is a collision, linear probes will check if the next place in the table is available, in a heavily loaded table (where most entries in the table are occupied) performance will suffer

Chained lists will significantly degrade lookup/search performance

Linear Probing Example



Some illustrations from: Wikipedia, Techopedia, Author Presentations, U North Texas

Rehashing or Double hashing

Create a second hash function h2(x) to be the increment (generalize linear probing).

Example: M=11, h(x) = x mod 11, h2(x) = x mod 7 + 1

Hash: 14, 17, 25, 37, 34, 16, 26

hash	Probes	
14 mod 11 = 3	1	
17 mod 11 = 6	1	
25 mod 11 = 3	2	25 mod 7 + 1 = 5
37 mod 11 = 4	1	
34 mod 11 = 1	1	
16 mod 11 = 5	1	
26 mod 11 = 4	2	26 mod 7 + 1 = 6
	----	
	9 probes	or 9/7 ~ 1 probe per key

0:
1: 34
2:
3: 14
4: 37
5: 16
6: 17
7:
8: 25
9:
10: 26

Do not use 0 as an increment
Make modulus of the second
Hash a prime number

Find 47:

	Probes
h(47) = 47 mod 11 = 3 not there	1
h2(47) = 47 mod 7 + 1 = 6 (cell is empty)	1

so 47 is not there

2 probes vs. 6 for linear probing

Quadratic hash

- Uses a quadratic polynomial to determine the increment
- Can use a simpler h(k, i) = (h(k) + i2) leads to the sequence h(k), h(k)+1, h(k)+4, h(k)+9 etc. i is the iteration number starting at 1
- HW: try this using the previous hash function

Hash: 14, 17, 25, 37, 34, 16, 26

hash	Probes	
14 mod 11 = 3	1	
17 mod 11 = 6	1	
25 mod 11 = 3	2	??
37 mod 11 = 4	1	
34 mod 11 = 1	1	
16 mod 11 = 5	1	
26 mod 11 = 4	2	??
	----	

9 probes or 9/7 ~ 1 probe per key

0:
1:
2:
3:
4:
5:
6:
7:
8:
9:

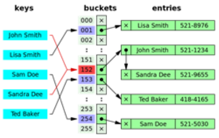
Find 47:

?? probes vs. 6 for linear probing

	Probes
h(47) = 47 mod 11 = 3 not there	??
h2(47) = 47 mod 7 + 1 = 6 (cell is empty)	??

Handling Collisions (buckets and lists)

- A common approach to handling collisions is to use a linked list for storing all objects whose keys (different) hash to the same index



The keys are hashed to an index of a bucket which then points to the data entries. Lisa Smith hashes to the value 001 which is an index to a bucket that points to a data entry of Lisa's name and phone number. John Smith and Sandra Dee both hash to the value 152, which is a collision. In this case, the collision is resolved by chaining (pointing) to a next entry.

- Ideally, the work to create a hash value should be very fast (why?), collisions should be minimized (why?), and the hash values should be uniformly distributed through out the hash table (Why?)
- Note that the hash value range should be the same size as the number of entries in the table, for a table of size N entries the hash values should be in the range of 0 to N-1

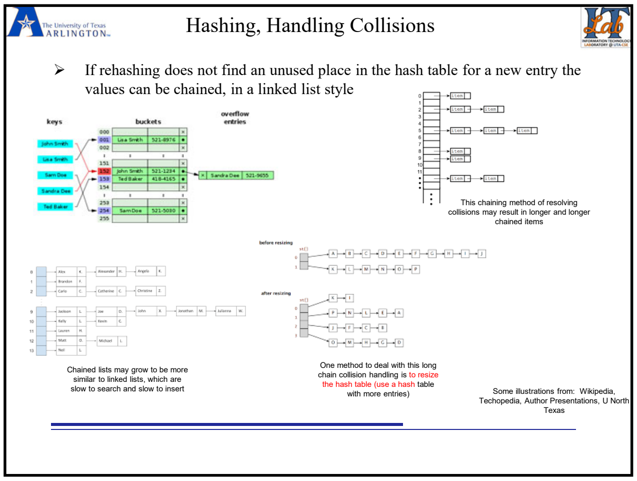
Some illustrations from: Wikipedia, Techopedia, Author Presentations

Handling Collisions (using buckets and Lists)

- Advantages
  - Hash the key only ones
  - Perform sequential search for the key to find the object associated with the key (remember multiple keys hash to the same bucket)
    - List can be kept sorted; tradeoff between sorting and lookup
  - Multiple objects with the same key (duplicates) can be handled easily
- Disadvantages
  - Sequential search of the list is O(n); on average, half the buckets need to be searched
  - Some buckets may be very long, increasing search cost
  - Non-uniform (skewed) hash values give rise to different bucket lengths
- Can compute average bucket lengths, etc.

Handling Collisions (using buckets and Lists)

- What if the bucket list length becomes too much from the response time perspective?
  - Increase the hash table size to distribute the keys across a larger range
  - Decreases the length of linked lists (assuming uniform distribution)
  - But, need to resize the hash table
    - rehash the entire set of keys and create a larger hash table (why?)
    - Can be expensive based on the number of keys
    - Does the hash function change?
- This representation does not need multiple hashing of the record on different keys (why?)
  - Can handle duplicate records (records with the same key value)
- One could use data pointers from buckets; typically <key, data pts>



Hashing, Handling Collisions

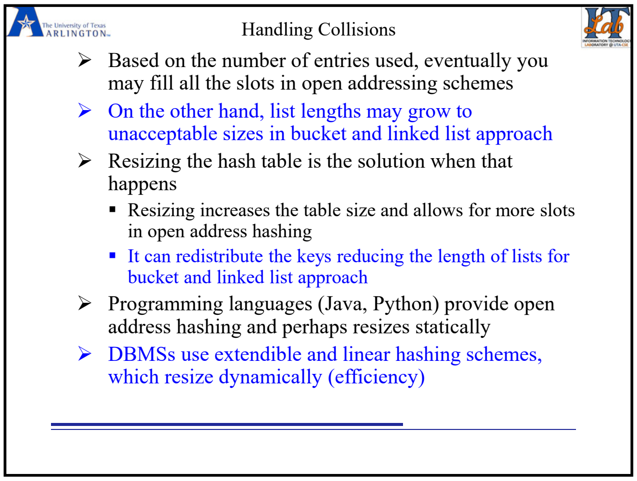
- ➢ If rehashing does not find an unused place in the hash table for a new entry the values can be chained, in a linked list style

This chaining method of resolving collisions may result in longer and longer chained items

Chained lists may grow to be more similar to linked lists, which are slow to search and slow to insert

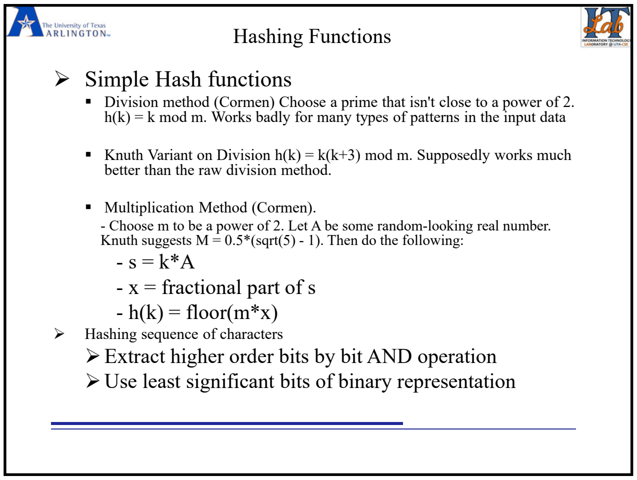
One method to deal with this long chain collision handling is to resize the hash table (use a hash table with more entries)

Some illustrations from: Wikipedia, Techopedia, Author Presentations, U North Texas



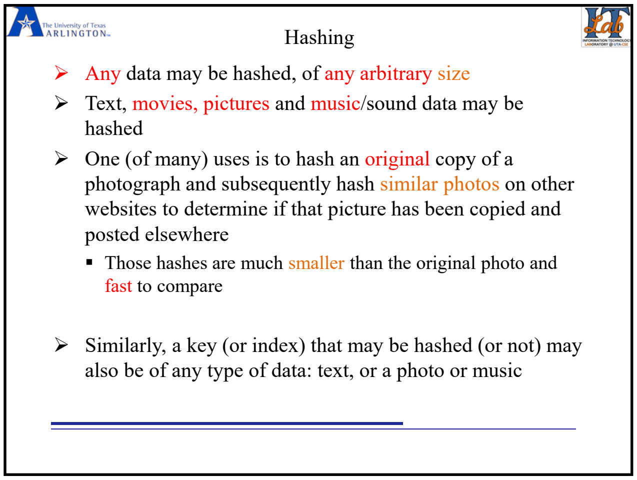
Handling Collisions

- ➢ Based on the number of entries used, eventually you may fill all the slots in open addressing schemes
- ➢ On the other hand, list lengths may grow to unacceptable sizes in bucket and linked list approach
- ➢ Resizing the hash table is the solution when that happens
  - Resizing increases the table size and allows for more slots in open address hashing
  - It can redistribute the keys reducing the length of lists for bucket and linked list approach
- ➢ Programming languages (Java, Python) provide open address hashing and perhaps resizes statically
- ➢ DBMSs use extendible and linear hashing schemes, which resize dynamically (efficiency)



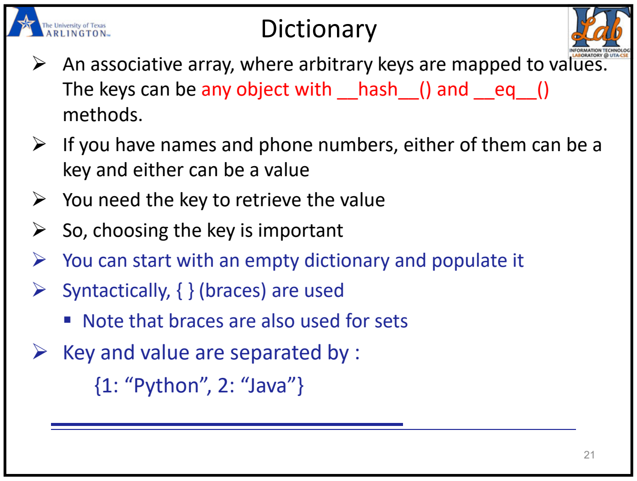
Hashing Functions

- ➢ Simple Hash functions
  - Division method (Cormen) Choose a prime that isn't close to a power of 2. h(k) = k mod m. Works badly for many types of patterns in the input data
  - Knuth Variant on Division h(k) = k(k+3) mod m. Supposedly works much better than the raw division method.
  - Multiplication Method (Cormen).
    - Choose m to be a power of 2. Let A be some random-looking real number. Knuth suggests M = 0.5*(sqrt(5) - 1). Then do the following:
      - s = k*A
      - x = fractional part of s
      - h(k) = floor(m*x)
- ➢ Hashing sequence of characters
  - ➢ Extract higher order bits by bit AND operation
  - ➢ Use least significant bits of binary representation



Hashing

- ➢ Any data may be hashed, of any arbitrary size
- ➢ Text, movies, pictures and music/sound data may be hashed
- ➢ One (of many) uses is to hash an original copy of a photograph and subsequently hash similar photos on other websites to determine if that picture has been copied and posted elsewhere
  - Those hashes are much smaller than the original photo and fast to compare
- ➢ Similarly, a key (or index) that may be hashed (or not) may also be of any type of data: text, or a photo or music

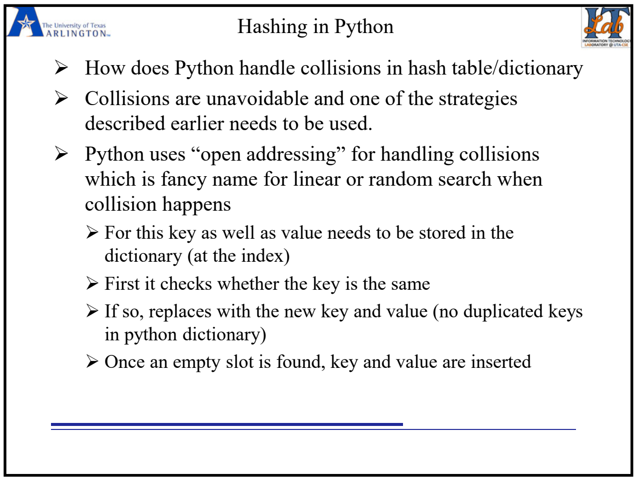


Dictionary

- An associative array, where arbitrary keys are mapped to values. The keys can be any object with __hash__() and __eq__() methods.
- If you have names and phone numbers, either of them can be a key and either can be a value
- You need the key to retrieve the value
- So, choosing the key is important
- You can start with an empty dictionary and populate it
- Syntactically, { } (braces) are used
  - Note that braces are also used for sets
- Key and value are separated by :

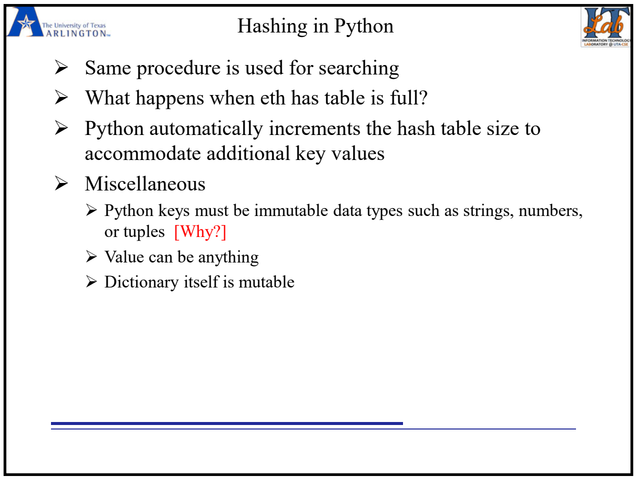
  {1: “Python”, 2: “Java”}

21



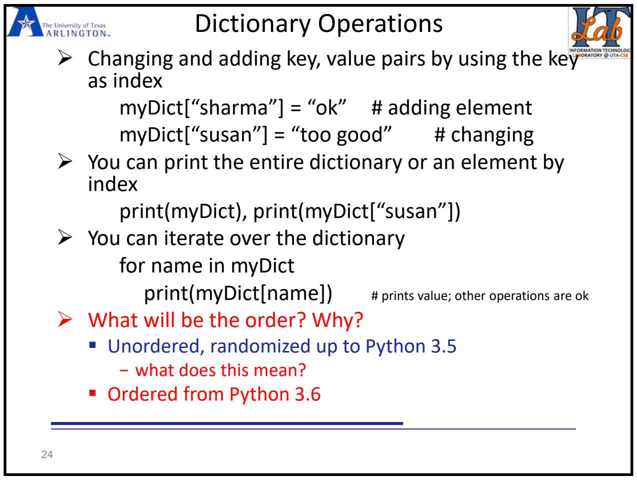
Hashing in Python

- How does Python handle collisions in hash table/dictionary
- Collisions are unavoidable and one of the strategies described earlier needs to be used.
- Python uses “open addressing” for handling collisions which is fancy name for linear or random search when collision happens
  - For this key as well as value needs to be stored in the dictionary (at the index)
  - First it checks whether the key is the same
  - If so, replaces with the new key and value (no duplicated keys in python dictionary)
  - Once an empty slot is found, key and value are inserted



Hashing in Python

- Same procedure is used for searching
- What happens when eth has table is full?
- Python automatically increments the hash table size to accommodate additional key values
- Miscellaneous
  - Python keys must be immutable data types such as strings, numbers, or tuples [Why?]
  - Value can be anything
  - Dictionary itself is mutable



Dictionary Operations

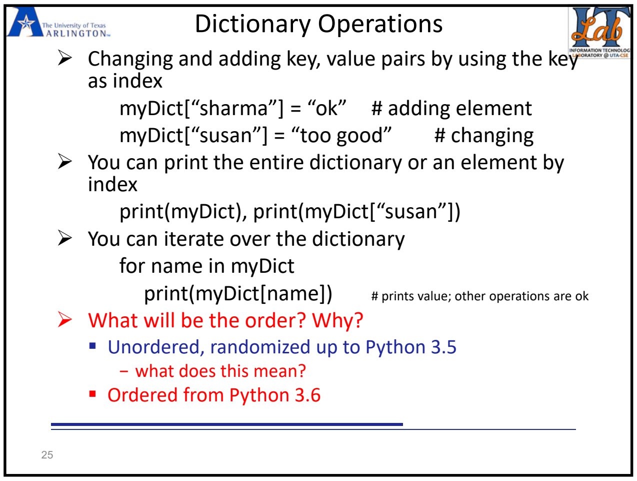
- Changing and adding key, value pairs by using the key as index

  myDict[“sharma”] = “ok” # adding element
  myDict[“susan”] = “too good” # changing
- You can print the entire dictionary or an element by index

  print(myDict), print(myDict[“susan”])
- You can iterate over the dictionary

  for name in myDict
      print(myDict[name]) # prints value; other operations are ok
- What will be the order? Why?
  - Unordered, randomized up to Python 3.5
    - what does this mean?
  - Ordered from Python 3.6

24



Dictionary Operations

- Changing and adding key, value pairs by using the key as index

  myDict[“sharma”] = “ok”   # adding element

  myDict[“susan”] = “too good”   # changing

- You can print the entire dictionary or an element by index

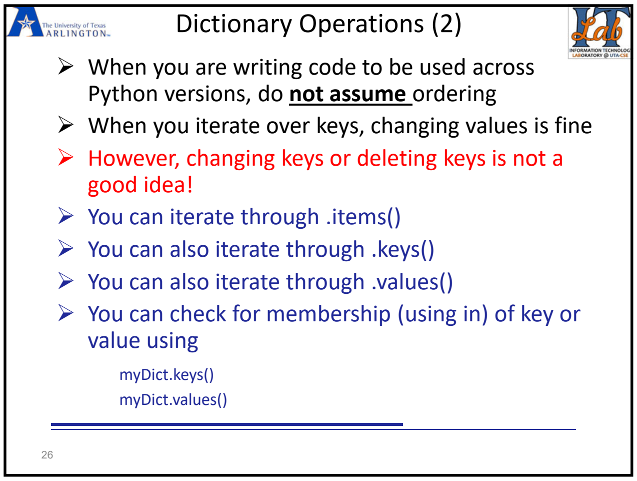
  print(myDict), print(myDict[“susan”])

- You can iterate over the dictionary

  for name in myDict

      print(myDict[name])   # prints value; other operations are ok

- What will be the order? Why?
  - Unordered, randomized up to Python 3.5
    - what does this mean?
  - Ordered from Python 3.6

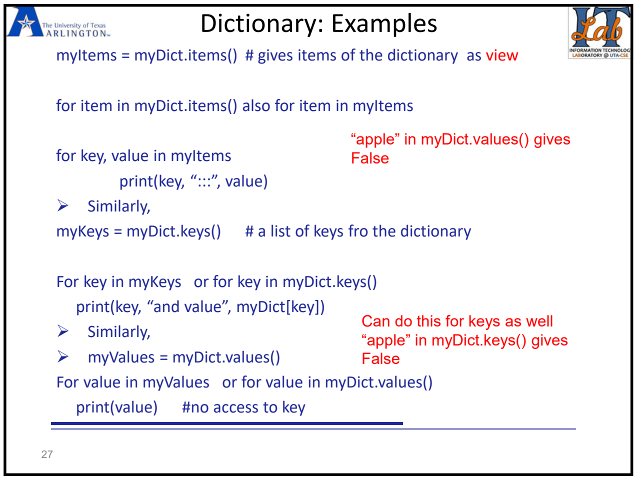


Dictionary Operations (2)

- When you are writing code to be used across Python versions, do **not assume** ordering
- When you iterate over keys, changing values is fine
- However, changing keys or deleting keys is not a good idea!
- You can iterate through .items()
- You can also iterate through .keys()
- You can also iterate through .values()
- You can check for membership (using in) of key or value using

  myDict.keys()

  myDict.values()



Dictionary: Examples

myItems = myDict.items() # gives items of the dictionary as view

for item in myDict.items() also for item in myItems

for key, value in myItems

    print(key, “:::”, value)

“apple” in myDict.values() gives False

- Similarly,

myKeys = myDict.keys()   # a list of keys fro the dictionary

For key in myKeys   or for key in myDict.keys()

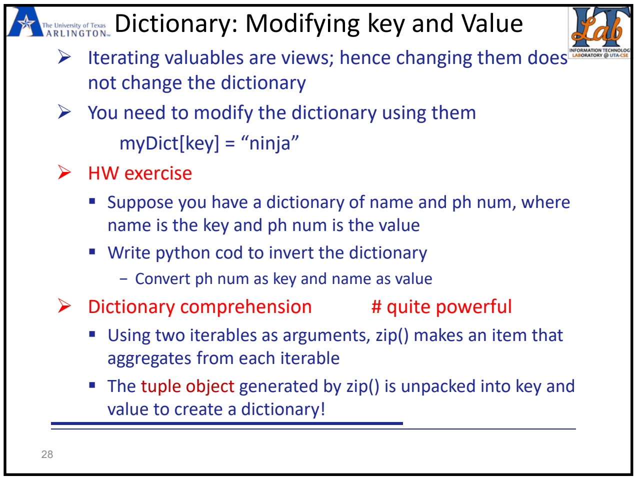
  print(key, “and value”, myDict[key])

- Similarly,
- myValues = myDict.values()

Can do this for keys as well “apple” in myDict.keys() gives False

For value in myValues   or for value in myDict.values()

  print(value)   #no access to key

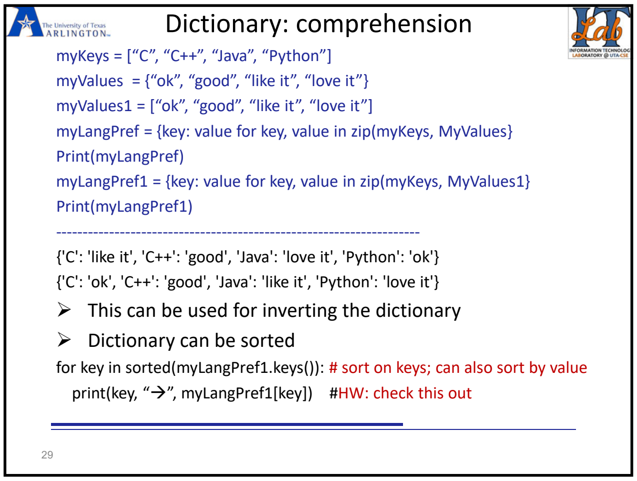


Dictionary: Modifying key and Value

- Iterating valuables are views; hence changing them does not change the dictionary
- You need to modify the dictionary using them

  myDict[key] = “ninja”

- HW exercise
  - Suppose you have a dictionary of name and ph num, where name is the key and ph num is the value
  - Write python cod to invert the dictionary
    - Convert ph num as key and name as value
- Dictionary comprehension   # quite powerful
  - Using two iterables as arguments, zip() makes an item that aggregates from each iterable
  - The tuple object generated by zip() is unpacked into key and value to create a dictionary!



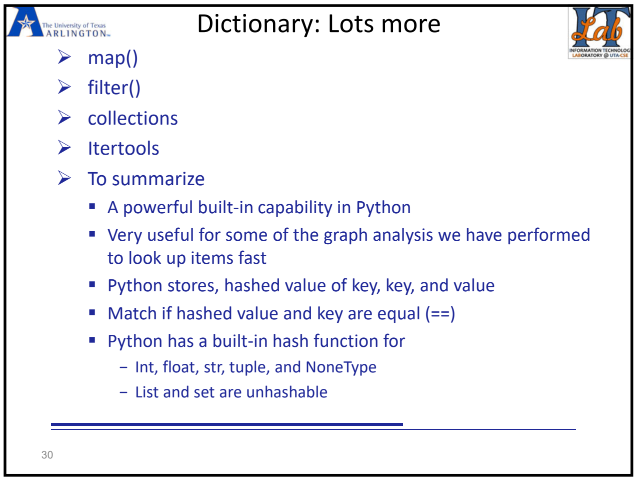
Dictionary: comprehension

```
myKeys = [“C”, “C++”, “Java”, “Python”]
myValues  = {“ok”, “good”, “like it”, “love it”}
myValues1 = [“ok”, “good”, “like it”, “love it”]
myLangPref = {key: value for key, value in zip(myKeys, MyValues}
Print(myLangPref)
myLangPref1 = {key: value for key, value in zip(myKeys, MyValues1}
Print(myLangPref1)
-----------------------------------------------------------------------
{'C': 'like it', 'C++': 'good', 'Java': 'love it', 'Python': 'ok'}
{'C': 'ok', 'C++': 'good', 'Java': 'like it', 'Python': 'love it'}
```

- This can be used for inverting the dictionary
- Dictionary can be sorted

```
for key in sorted(myLangPref1.keys()): # sort on keys; can also sort by value
   print(key, “→”, myLangPref1[key])    #HW: check this out
```

29



Dictionary: Lots more

- map()
- filter()
- collections
- Itertools
- To summarize
  - A powerful built-in capability in Python
  - Very useful for some of the graph analysis we have performed to look up items fast
  - Python stores, hashed value of key, key, and value
  - Match if hashed value and key are equal (==)
  - Python has a built-in hash function for
    - Int, float, str, tuple, and NoneType
    - List and set are unhashable

30

Questions/comments

For more information visit:
http://itlab.uta.edu

Spring 2019 CSE 6331