Abstract Data Structures and Objects Part 1

Week 2: Computer Programming for Data Scientists (7CCSMCMP)

7 October 2016

JSON File Structure

JSON (pronounced *JAY-SON*) - **J**ava**S**cript **O**bject **N**otation

hierarchical data - like XML, but the syntax is closer to Python

data interchange format comprised of a dictionary, containing any of the following structures:

- primitive data types: *int*, *float*, *boolean*, *string*
- *lists* (arrays)
- *dictionaries*

Definition available: http://json.org

```
{"specials": [{
                    "name": "Plain",
                    "inches": 12,
                    "toppings": ["Extra Cheese"],
                    "price": 3.99,
                    "vegetarian": true
                  },{
                   "name": "Supreme",
                   "inches": 22,
                   "toppings": ["Mushrooms","Anchoives",
                                "Avocados","Extra Cheese","Green Peppers",
                                "Sausage","Olives","Red Onions"],
                   "price": 19.99
                  },{
                   "name": "Garden",
                   "inches": 16,
                   "toppings": ["Extra Cheese","Olives","Spinach","Red Onions",
                                "Mushrooms","Avocados"],
                   "price": 10.99,
                   "vegetarian": true
                  }],
"display_until": "31-10-2016"
}
```

JSON - using Python `json` module

Docs: https://docs.python.org/2/library/json.html

`json.load(f)` - read and parse data from a .json file

- `f` is the file descriptor returned from the `open()` function

```
import json

# open pizza-specials.json file and parse into a variable
try:
    with open("data/pizza-specials.json", "r") as fd:
        pizza_json = json.load(fd)
except IOError as ioe:
    print("I/O Error in openning file: %s" % ioe)
```

```
pizza_json
```

```
{u'display_until': u'31-10-2016',
 u'specials': [{u'inches': 12,
   u'name': u'Plain',
   u'price': 3.99,
   u'toppings': [u'Extra Cheese'],
   u'vegetarian': True},
  {u'inches': 22,
   u'name': u'Supreme',
   u'price': 19.99,
   u'toppings': [u'Mushrooms',
    u'Anchoives',
    u'Avocados',
```

JSON - using Python `json` module

`json.loads(s)` - read and parse data from a string, `s`, that is in json format

```
supreme_string = """{
  "name": "Supreme",
  "inches": 22,
  "toppings": ["Mushrooms","Anchoives",
               "Avocados","Extra Cheese","Green Peppers",
               "Sausage","Olives","Red Onions"],
  "price": 19.99
}
"""
```

```
supreme_json = json.loads(supreme_string)
supreme_json
```

```
{u'inches': 22,
 u'name': u'Supreme',
 u'price': 19.99,
 u'toppings': [u'Mushrooms',
  u'Anchoives',
  u'Avocados',
  u'Extra Cheese',
  u'Green Peppers',
  u'Sausage',
  u'Olives',
  u'Red Onions']}
```

JSON - parsing the dictionary

Parsing the read-in JSON `dict` is the same as parsing Python *dictionaries* and *lists*.

Know your format - parsing `dict` depends on the structure of the data!

```
# can use dict.keys() to help
for k in supreme_json.keys():
    if k == "toppings":
        print("toppings: ")
        for t in supreme_json["toppings"]:
            print("* %s" % t)
    else: # just print the other key-value pairs
        print("%s = %s" %(k, supreme_json[k]))
```

```
price = 19.99
toppings:
* Mushrooms
* Anchoives
* Avocados
* Extra Cheese
* Green Peppers
* Sausage
* Olives
* Red Onions
name = Supreme
inches = 22
```

JSON - using Python `json` module

`json.dump(d, f)` - writes the Python dictionary `d` to a json file, `f`

- `f` is the file descriptor returned from the `open()` function

```
pizza_orders = [{"size": "M", "toppings": 2, "price": 5.99},
                {"size": "S", "toppings": 1, "price": 2.99},
                {"size": "L", "toppings": 1, "price": 7.99},
                {"size": "XL", "toppings": 5, "price": 12.99}]
```

```
pizza_dict = {"orders": pizza_orders}
```

```
try:
    with open("data/pizza-orders.json","w") as pizza_fd:
        json.dump(pizza_dict, pizza_fd)
except IOError as ioe:
    print("Uh Oh! I/O Error: %s" % ioe)
```

`pizza-orders.json`

```
1  {"orders": [{"price": 5.99, "toppings": 2, "size": "M"}, {"price": 2.99, "toppings":
       1, "size": "S"}, {"price": 7.99, "toppings": 1, "size": "L"}, {"price": 12.99,
     "toppings": 5, "size": "XL"}]}
```

Use `indent` option in json.dump to output a prettier format.

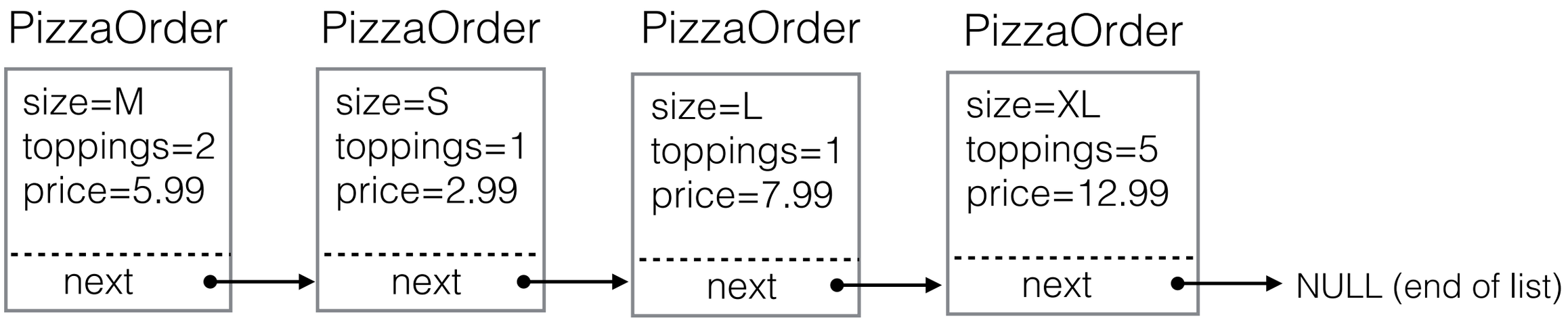
- **See**: https://docs.python.org/2/library/json.html#json.dump

Abstract Data Structures

- A *data structure* is an *abstract data type* (unlike a *real* data type like an `int`, or an `array`).
- It maps a *virtual* model of data to a **real** data type.
- Classic data structures in computer science:
  - *linked lists*
  - *queues*
  - *stacks*
  - *maps*
  - *trees*
- Each abstract data type has rules how to *add* and *remove* elements, and how to *iterate* through the elements of the data structure.

Abstract Data Structures - linked lists

- A *linked list* chains items together using a field called "*next*", which points to the next item in the chain:



- You can *iterate* through a linked list with a for loop
- You can customize how to *iterate* through a sequence of programmer-defied objects using an ***iterator*** (i.e. defining where the "next" pointer goes to)

Iterators - the iteration protocol

- in Python, there are special *iterator* types that support user-defined iterating over containers

- these define two types of functions:
  – `__iter__()` constructor
  which is defined in both the *container* object over which the iterating will happen and the *iterator* object which will do the iterating

  – `next()` function
  which goes to the "*next*" container item

  – the `next()` function must raise the `StopIteration` exception when it is time to stop iterating

- More information: http://anandology.com/python-practice-book/iterators.html#iterators

Iterators - Examples from `csv` and `xml` module

- We have seen two examples from previous lectures that used ***iterator*** objects.

- Reading lines from a CSV file with `csv.reader()` or `csv.DictReader()`

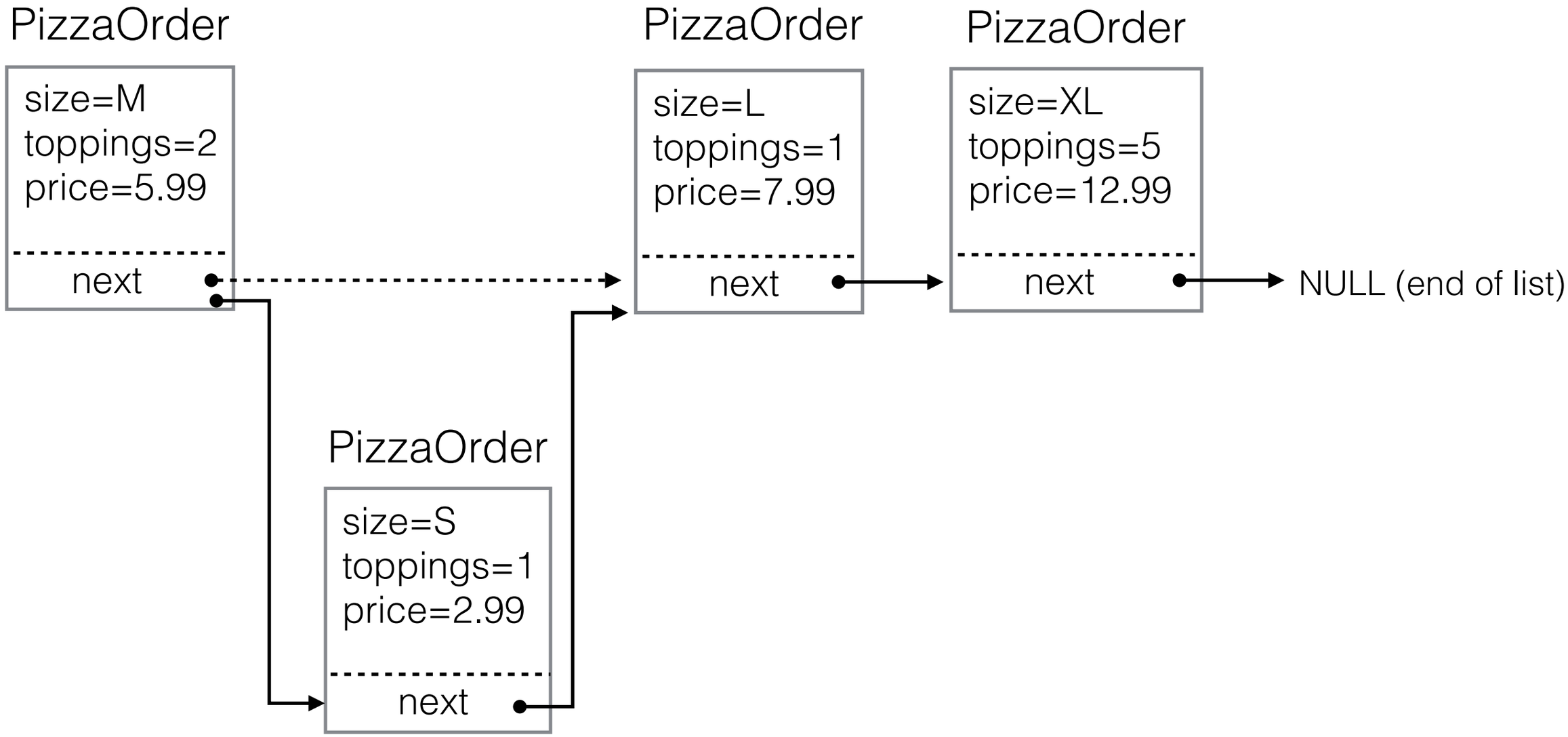
```
with open("movies.csv", "r") as csv_file:
csv_data = csv.DictReader(csv_file)
for row in csv_data:
    print(row) # prints each row as a dictionary
```

- Iterating over child Elements from a parent Element in an XML ElementTree

```
import xml.etree.ElementTree as et
tree = et.ElementTree(file="movies.xml")
root = tree.getroot()
for child in root:
  print("%s: %s" % (child.tag, child.text))
```

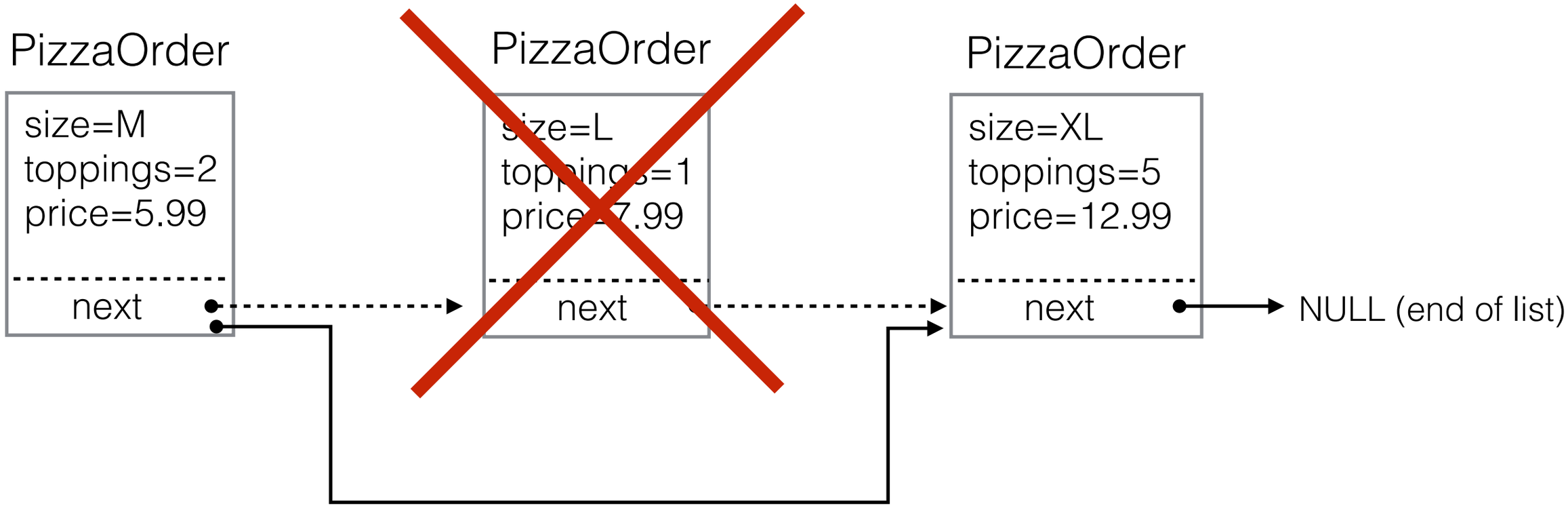
Abstract Data Structures - adding elements to linked lists

To **add** an item to a linked list, one instance of the item to be added is instantiated and then the “next” fields are set in the new item and the item in the linked list after which the new item is being inserted.



Abstract Data Structures - removing elements from a linked lists

• To remove an item to a linked list, the “next” field pointers are simply moved around it:



•Then the old item is *discarded* (so that the memory can be re-used)

Abstract Data Structures - queue

- a queue is like a check-out line at a store
- you can only add items (people) to the end of the line
- you can only remove items (people) from the front of the line
- hence, a queue is also called a FIFO (first in, first out)
- following are the typical names for queue routines:
  - **enqueue**: for adding items to a queue
  - **dequeue**: for removing items from a queue
- you can use a Python list as a queue with these list functions:

enqueue ···▶
dequeue ···▶

```
queue = ["A", "B", "C", "D", "E"]
queue.append("F")
print queue
queue.pop(0) # dequeue from the Front of the queue
print queue
```

```
['A', 'B', 'C', 'D', 'E', 'F']
['B', 'C', 'D', 'E', 'F']
```

Abstract Data Structures - stack

- a stack is like a stack of plates
- you can only add items (plates) to the top of the stack
- you can only remove items (plates) from the top of the stack
- hence, a stack is also called a LIFO (last in, first out)
- following are the typical names for stack routines:
  - **push**: for adding items to a stack
  - **pop**: for removing items from a stack
- you can use a Python list as a stack with these list functions:

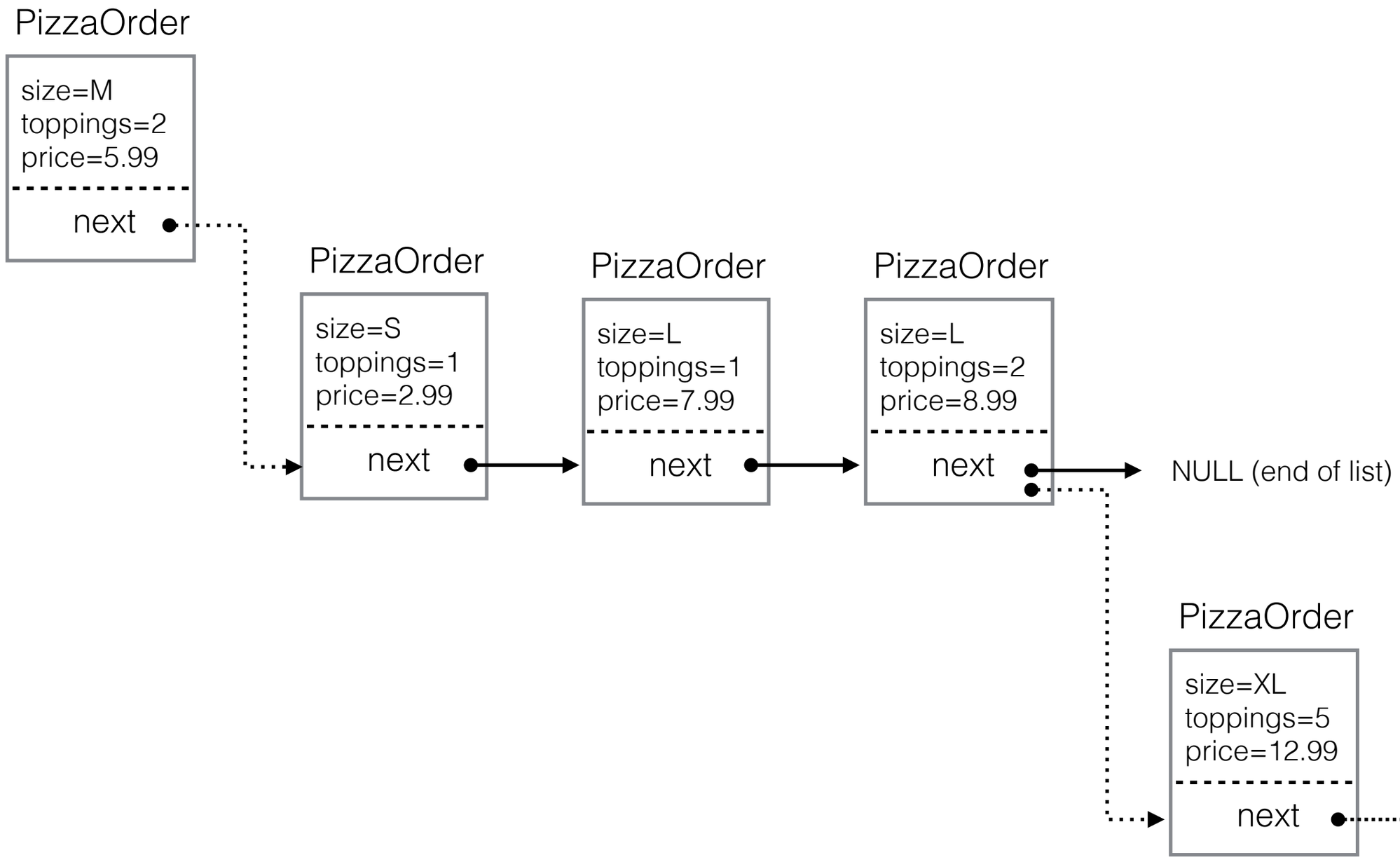
push ···▶
pop ···▶

```
stack = ["A", "B", "C", "D", "E"]
stack.insert(0, "F") # push to the top of the stack
print stack
stack.pop(0) # pop from the top of the stack
print stack
```

```
['F', 'A', 'B', 'C', 'D', 'E']
['A', 'B', 'C', 'D', 'E']
```

Abstract Data Structures - stack vs queue: adding items

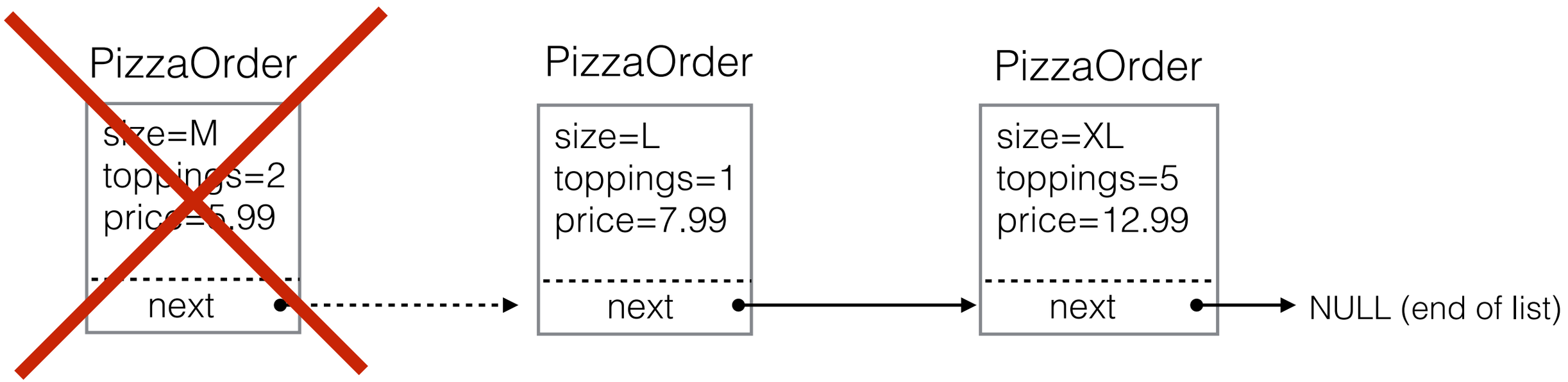
new stack item gets added to the front (top) of the list



new queue item gets added to the end of the list

Abstract Data Structures - stack vs queue: removing items

stack items get removed from the front (top) of the list



queue items get removed from the front of the list

Abstract Data Structures - hashing and mapping

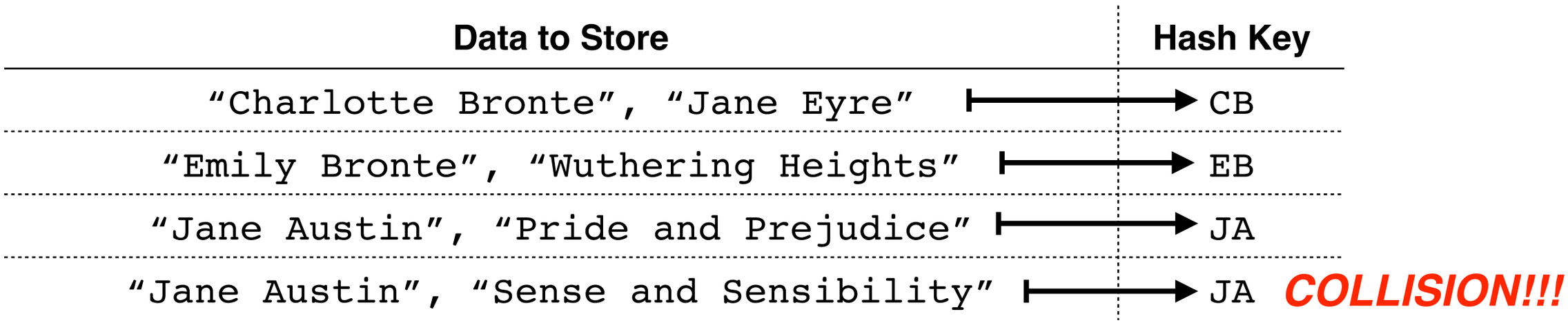
A *map* is an abstract data structure which uses a *key* as an index (into the table) to look up the associated *value*; a *hash table* is an example of a mapping data structure

- ***Quick*** to look up a value in a map or a hash table, versus a list (where you have to iterate through every item when search the list)

- In Python, a `dict` (dictionary) implements mapping

- A *collision* is when two keys "hash" to the same location the table
  - a *hash key* can be constructed out of an abbreviated form of the data you are trying to store (and index)

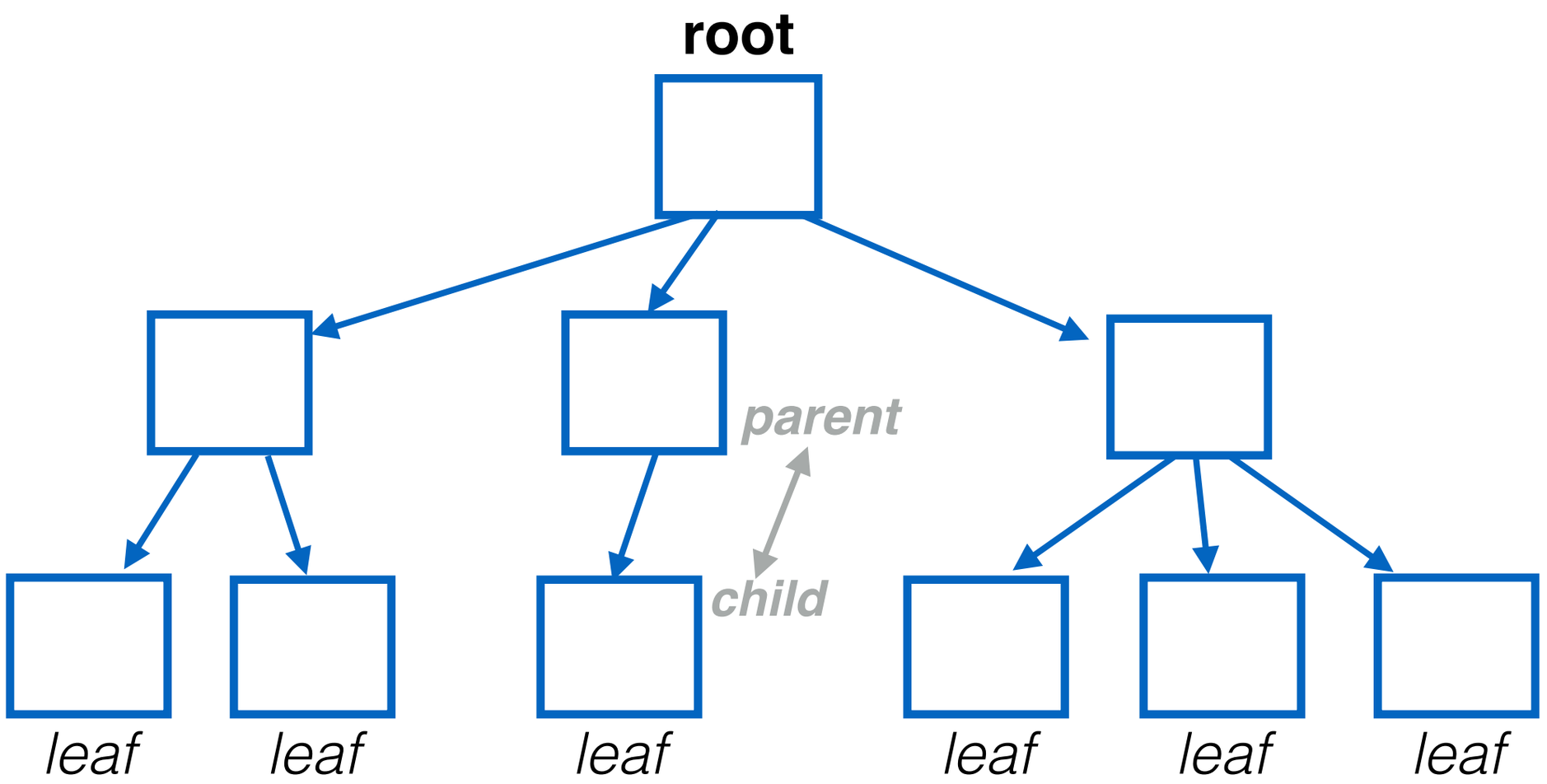
Example of *hashing* (and a *collision*):



- Collisions are not possible with a `dict` because defining a value for a key that already exists in the dictionary, then the new value overwrites the old value

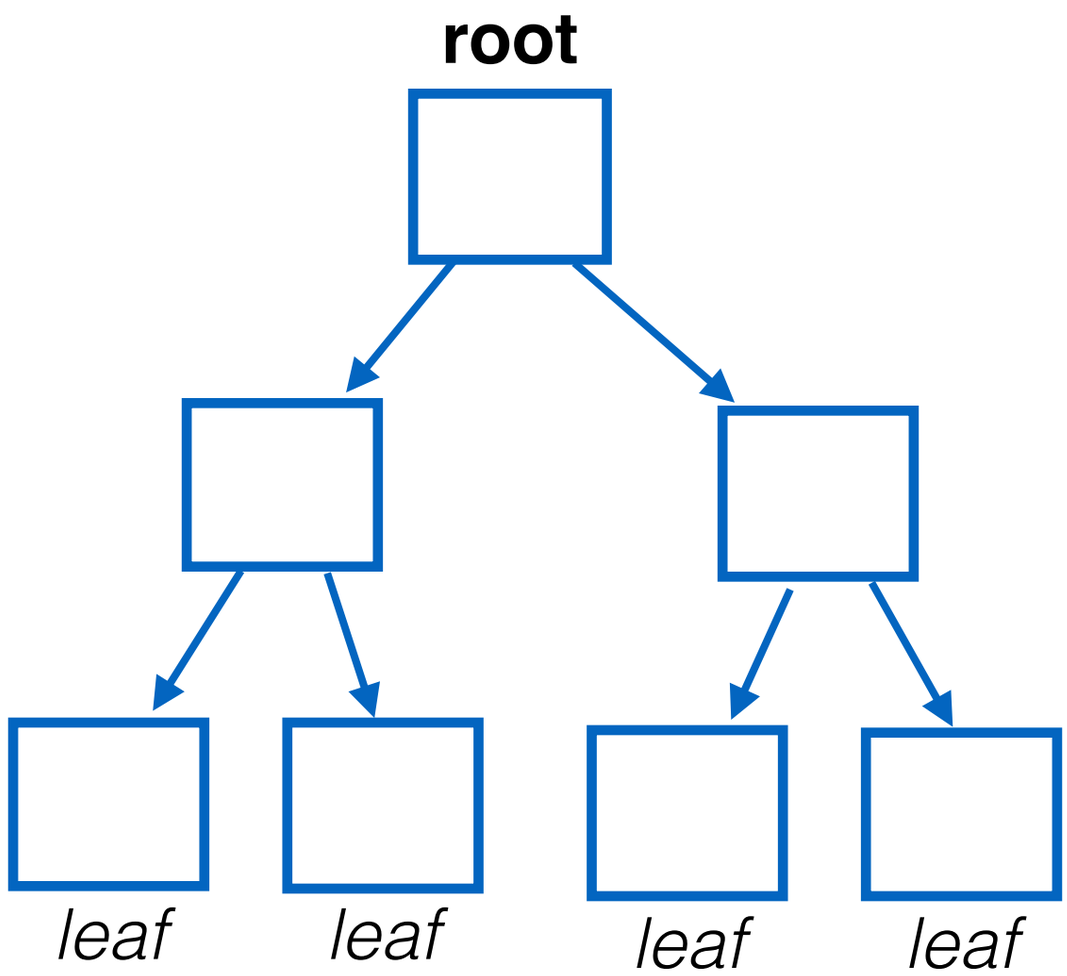
Abstract Data Structures - trees

- A *tree* is a hierarchical data structure
- Very convenient for representing data that is stored in XML and JSON formats
- A tree consists of *nodes* and *links*
- The *root* of the tree is the node at the top
- The *leaves* of the tree are the nodes at the bottom



Abstract Data Structures - binary trees

- In a *binary tree*, every non-leaf node has exactly two children



Pizza Topping Decoding

```
# defined codes for Pizza Toppings
toppings = {"o": "Onions",
            "g": "Green Peppers",
            "m": "Mushrooms",
            "p": "Pepperoni",
            "s": "Sausage",
            "x": "Extra cheese"}

# convert a list of toppings codes to Toppings
my_toppings = ["p", "x", "m"]

toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])

print toppings_list

# pretty print the list using join
print ", ".join(toppings_list)
```

```
['Pepperoni', 'Extra cheese', 'Mushrooms']
Pepperoni, Extra cheese, Mushrooms
```

List Comprehensions - Pizza Topping Decoding

```
# defined codes for Pizza Toppings
toppings = {"o": "Onions",
            "g": "Green Peppers",
            "m": "Mushrooms",
            "p": "Pepperoni",
            "s": "Sausage",
            "x": "Extra cheese"}

# convert a list of toppings codes to Toppings
my_toppings = ["p", "x", "m"]

toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])

print toppings_list

# pretty print the list using join
print ", ".join(toppings_list)
```

```
['Pepperoni', 'Extra cheese', 'Mushrooms']
Pepperoni, Extra cheese, Mushrooms
```

List Comprehensions - syntax

List comprehensions provide a concise way to create lists.

Equivalent Python

for-loop

```
toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])
```

list comprehension

```
toppings_list = [toppings[t] for t in my_toppings]
```

List Comprehensions - syntax

List comprehensions provide a concise way to create lists.

Uses the same square brackets that a **`list`** would use

for-loop

```
toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])
```

list comprehension

```
toppings_list = [toppings[t] for t in my_toppings]
```

List Comprehensions - syntax

List comprehensions provide a concise way to create lists.

For loop iterating over an iterator (the list **`my_toppings`**) with a loop variable (**`t`**)

for-loop

```
toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])
```

list comprehension

```
toppings_list = [toppings[t] for t in my_toppings]
```

List Comprehensions - syntax

List comprehensions provide a concise way to create lists.

Define what the item should be in the *new list*.

for-loop

```
toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])
```

list comprehension

```
toppings_list = [toppings[t] for t in my_toppings]
```

List Comprehensions - syntax

List comprehensions provide a concise way to create lists.

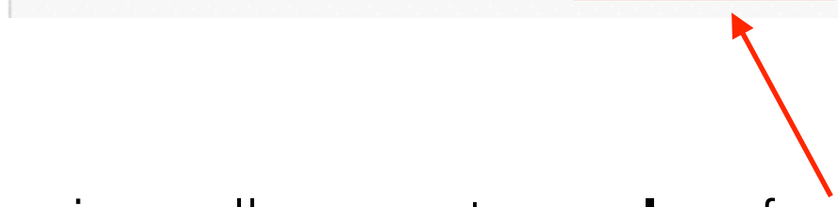
Define what the item should be in the *new list*.

for-loop

```
toppings_list = []
for t in my_toppings:
    toppings_list.append(toppings[t])
```

list comprehension

```
toppings_list = [toppings[t] for t in my_toppings]
```



List comprehensions allow you to **apply** a function on a list to create a new list.

In *functional programming* this is called **map** operation, applying function onto a collection of items. (Part of the Map-Reduce technique in Big Data processing)

List Comprehensions - Pizza Topping Decoding

Reduces the number of lines of code; also allows for embedding the list comprehension

```
# defined codes for Pizza Toppings
toppings = {"o": "Onions",
            "g": "Green Peppers",
            "m": "Mushrooms",
            "p": "Pepperoni",
            "s": "Sausage",
            "x": "Extra cheese"}

# convert a list of toppings codes to Toppings
my_toppings = ["p", "x", "m"]

toppings_list = [toppings[t] for t in my_toppings]

print toppings_list

# pretty print the list using join
print ", ".join(toppings_list)
```

```
['Pepperoni', 'Extra cheese', 'Mushrooms']
Pepperoni, Extra cheese, Mushrooms
```

List Comprehensions - Pizza Topping Decoding

Reduces the number of lines of code; also allows for embedding the list comprehension

```
# defined codes for Pizza Toppings
toppings = {"o": "Onions",
            "g": "Green Peppers",
            "m": "Mushrooms",
            "p": "Pepperoni",
            "s": "Sausage",
            "x": "Extra cheese"}

# convert a list of toppings codes to Toppings
my_toppings = ["p", "x", "m"]

# pretty print the list using join
print ", ".join([toppings[t] for t in my_toppings])
```

```
['Pepperoni', 'Extra cheese', 'Mushrooms']
Pepperoni, Extra cheese, Mushrooms
```

List Comprehensions - Hold the Mushrooms

Optionally can add an `if` statement to *filter* the original list items

```
# defined codes for Pizza Toppings
toppings = {"o": "Onions",
            "g": "Green Peppers",
            "m": "Mushrooms",
            "p": "Pepperoni",
            "s": "Sausage",
            "x": "Extra cheese"}

# convert a list of toppings codes to Toppings
my_toppings = ["p", "x", "m"]

toppings_list = [toppings[t] for t in my_toppings if t != "m"]

print toppings_list

# pretty print the list using join
print ", ".join(toppings_list)
```

```
['Pepperoni', 'Extra cheese']
Pepperoni, Extra cheese
```

List Comprehensions - Embed the List Comprehension

Optionally can add an `if` statement to *filter* the original list items

```
# defined codes for Pizza Toppings
toppings = {"o": "Onions",
            "g": "Green Peppers",
            "m": "Mushrooms",
            "p": "Pepperoni",
            "s": "Sausage",
            "x": "Extra cheese"}

# convert a list of toppings codes to Toppings
my_toppings = ["p", "x", "m"]

toppings_list = [toppings[t] for t in my_toppings if t != "m"]

print toppings_list

# pretty print the list using join
print ", ".join(toppings_list)
```

```
['Pepperoni', 'Extra cheese']
Pepperoni, Extra cheese
```

List Comprehensions - Exercises

```
[i for i in range(10)]
```

List Comprehensions - Exercises

```
[i for i in range(10)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[i for i in range(10) if i%2 == 0]
```

List Comprehensions - Exercises

```
[i for i in range(10)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[i for i in range(10) if i%2 == 0]
```

```
[0, 2, 4, 6, 8]
```

```
[i for i in range(10) if i%2 == 1]
```

List Comprehensions - Exercises

```
[i for i in range(10)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[i for i in range(10) if i%2 == 0]
```

```
[0, 2, 4, 6, 8]
```

```
[i for i in range(10) if i%2 == 1]
```

```
[1, 3, 5, 7, 9]
```

List Comprehensions - Exercises

```
[i for i in range(10)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[i for i in range(10) if i%2 == 0]
```

```
[0, 2, 4, 6, 8]
```

```
[i for i in range(10) if i%2 == 1]
```

```
[1, 3, 5, 7, 9]
```

```
[i**2 for i in range(10) if i%2 == 1]
```

Applying the function i^2

List Comprehensions - Exercises

```
[i for i in range(10)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[i for i in range(10) if i%2 == 0]
```

```
[0, 2, 4, 6, 8]
```

```
[i for i in range(10) if i%2 == 1]
```

```
[1, 3, 5, 7, 9]
```

```
[i**2 for i in range(10) if i%2 == 1]
```

```
[1, 9, 25, 49, 81]
```

List Comprehensions - Applying a Complicated Function

```
def score_word(my_word):
    if(my_word.startswith("chi")):
        return 9
    if(len(my_word) > 3 and len(my_word) < 7):
        return 1
    elif(len(my_word) > 7):
        return 3
    else:
        return 7

# apply the scoring function to a list of words
score_these_words = ["pizza", "zizzled", "chipp", "frog", "elephant"]
scores = [score_word(w) for w in score_these_words]

print scores
```

```
[1, 7, 9, 1, 3]
```

BONUS - *convert to a dictionary using* ***`zip`*** *and* ***`dict`***

```
# apply the scoring function to a list of words
score_these_words = ["pizza", "zizzled", "chipp", "frog", "elephant"]
scores = [score_word(w) for w in score_these_words]

print scores
```

```
[1, 7, 9, 1, 3]
```

```
# zip: handy way of joining to lists together!
scored_words = zip(score_these_words, scores)
print scored_words
```

```
[('pizza', 1), ('zizzled', 7), ('chipp', 9), ('frog', 1), ('elephant',
 3)]
```

BONUS - *convert to a dictionary using* ***`zip`*** *and* ***`dict`***

```
# apply the scoring function to a list of words
score_these_words = ["pizza", "zizzled", "chipp", "frog", "elephant"]
scores = [score_word(w) for w in score_these_words]

print scores
```

```
[1, 7, 9, 1, 3]
```

```
# zip: handy way of joining to lists together!
scored_words = zip(score_these_words, scores)
print scored_words
```

```
[('pizza', 1), ('zizzled', 7), ('chipp', 9), ('frog', 1), ('elephant',
 3)]
```

```
# combine the tuples into a dictionary!
scored_dict = dict(scored_words)
print scored_dict
```

```
{'zizzled': 7, 'chipp': 9, 'elephant': 3, 'frog': 1, 'pizza': 1}
```

Summary

JSON - commonly used data interchange format

Abstract Data Structures - concepts

- linked lists
- stacks vs. queues
- mapping and hashing
- trees

Iterators - Python interface for iterating over items in a collection

- examples from previous lectures

List Comprehensions - concise way of creating lists

- syntax:

```
[<item mapping> for <loop variable> in <original list>]
```

- filtering items using an `if` conditional