# Special Topics in Computer Science- CSC 4992

Search Algorithms

# Searching

- One of the most important concepts in computer science

- We have all this information, so how do we find and extract what we want from it?

- A big business: for example, Google, a multi-billion dollar company, was originally based on a search engine (see *The Search: How Google Rewrote the Rules of Business and Transformed Our Culture*, by John Battelle)

# Simple Searching in Python

- Determine whether or not a given element is in a collection of elements
- The **`in`** operator works with sequences and dictionaries
- Returns **`True`** if yes or **`False`** if no

# Examples

```
>>> 'e' in 'power elite'
True

>>> 'k' in 'quagmire'
False

>>> 33 in [99, 22, 44, 33]
True

>>> 'name' in {'age':39, 'name':'jack'}
True

>>> 'Ken/1001' in self._accounts     # a dictionary
True
```

**`in`** uses **`==`** to compare for equality

# How Does `in` Work?

- Loop through the collection and compare each element to the target element
- If the target equals the current element, return **`True`**
- After the loop is finished, return **`False`**

# Reinventing `in`

```
def reinventIn(target, collection):
    for element in collection:
        if element == target:
            return True
    return False
```

```
>>> reinventIn(33, [44, 22, 33, 111])
True
```

Uses **`==`** to compare for equality

The first return point quits as soon as a match is found

The second return point quits if no match is found

*But*: we can just use **`in`** with the collection instead!

# SavingsAccount Class

```
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount
```

# Defining a Bank Class

- Set up a data structure for the data

- Define methods to
  - Add a new account
  - Remove an account
  - Access an account for deposits and withdrawals
  - Compute the interest on all accounts

# Defining the `Bank` Class

```python
class Bank(object):
    """This class represents a bank."""

    def __init__(self):
        self._accounts = {}

    # Other methods go here
```

Use a dictionary for the accounts, keyed by the name + pin

# Defining the `Bank` Class

```python
class Bank(object):
    """This class represents a bank."""

    def __init__(self):
        self._accounts = {}

    def __str__(self):
        result = ""
        for account in self._accounts.values():
            result += str(account) + "\n"
        return result
```

Always define `__init__` and `__str__` first

# Defining the **Bank** Class

```python
class Bank(object):
    """This class represents a bank."""

    def __init__(self):
        self._accounts = {}

    def __str__(self):
        return "\n".join(map(str, self._accounts.values()))
```

Simplify, simplify!

# Adding an Account

```
class Bank(object):
    """This class represents a bank."""

    def __init__(self):
        self._accounts = {}

    def add(self, account):
        key = account._name + account._pin
        self._accounts[key] = account
```

An account is a value in the dictionary, keyed by its name and pin

The remaining methods are similar

# A Method for **`in`**

```
class Bank(object):

    def __init__(self):
        self.accounts = {}

    def __contains__(self, key):
        return key in self.accounts
```

```
>>> b = Bank()
>>> b.add(SavingsAccount('Ken', '1001', 1000.00))
>>> b.makeKey('Ken', '1001') in b
True
```

Python runs **`__contains__`** when it sees **`in`**

# Other Types of Searches

- Determine whether or not a customer named 'Ken' has an account in the bank
- Find *all* of the accounts that
  - Have the name 'Ken'
  - Have a balance greater than $100,000
  - Satisfy both of the above conditions
  - Satisfy any of the above conditions

# Satisfying a Property

```
def isNameInAnAccount(name, collection):
    for element in collection:
        if element.getName() == name:
            return element
    return None
```

```
isNameInAnAccount('Ken', self.accounts.values()))

# Returns <the first account object whose name is 'Ken'>
```

Works like **`in`**, but determines the presence or absence of an element with a given property in the collection

Usually iterates over a sequence of objects

Returns the object found or **`None`**

# Match Other Properties

```
def isBalanceInAnAccount(balance, collection):
    for element in collection:
        if element.getBalance() == balance:
            return element
    return None
```

```
isBalanceInAnAccount(10000, self.accounts.values()))

# Returns None      #self.accounts.values()=> SavingsAccount object
```

Varies the method called to access the property in the element

Otherwise, the code has *exactly* the same pattern

# Match Other Properties

```
def isBalanceInAnAccount(balance, collection):
    for element in collection:
        if element.getBalance() >= balance:
            return element
    return None
```

```
isBalanceInAnAccount(10000, self.accounts.values())

# Returns <the first account object whose balance >= 10000>
```

Might also vary the comparison used to detect the property

# Generalize as a **detect** Operation

```
def detect(test, collection):
    for element in collection:
        if test(element):
            return element
    return None

def has10000(account):
    return account.getBalance() == 10000
```

```
detect(has10000, self.accounts.values())

# Returns <the first account object whose balance == 10000>
```

The search function should not know the details of the test function

The test function is composed before **detect** is called

The test function is passed as an argument to **detect**

# Generalize as a **`detect`** Operation

```
def detect(test, collection):
    for element in collection:
        if test(element):
            return element
    return None

def hasKen(account):
    return account.getName() == 'Ken'
```

```
detect(hasKen, self.accounts.values())

# Returns <Ken's account object>
```

Change the test function to redirect the search

# Simplify with **`lambda`**

```python
def detect(test, collection):
    for element in collection:
        if test(element):
            return element
    return None
```

```python
detect(lambda account: account.getName() == 'Ken',
           self.accounts.values())

# Returns <Ken's account object>
```

**`lambda`** creates an *anonymous function* on the fly, to be used just where it’ s needed

The **`lambda`** expression expects one argument and must return a Boolean value

# Find All the Elements that Satisfy a Given Criterion

- Simple detection finds and returns the first element that satisfies the search criterion

- Alternatively, we could find *all* the matching elements and build a list of them as we go

- Return this list

- This is just Python's **`filter`** function!

# Efficiency of Search

- Searches thus far have been *sequential*: each element in a collection must be visited, from beginning to end, before the target is found or we run off the end

- On the average, a simple sequential search stops halfway through the collection, but in the worst case, it must visit every element

# Improving Efficiency: Binary Search

- If we assume that the elements are *sorted*, we can visit the relevant ones much more quickly

- Start at the collection’s midpoint (must be a sequence)

- If the target is less than the midpoint element, continue the search only to the left of the midpoint

- If the target is greater than the midpoint element, continue the search only to the right of the midpoint

- Will eventually hit the target or run out of elements

# Binary Search

```
def binaryIn(target, collection):
    left = 0
    right = len(collection) - 1
    while left <= right:
        midpoint = (left + right) // 2
        if target == collection[midpoint]:
            return True
        elif target < collection[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return False
```

Assumes that the element in a collection supports the **==** and **<** operators