1. Python's default recursion limit is 1,000 which is too low for recursive traversals of a BST, but it can be changed by:

```
import sys

numberOfItems = 5000
print 'recursion limit:',sys.getrecursionlimit()
sys.setrecursionlimit(numberOfItems*2)
print 'numberOfItems:', numberOfItems,'recursion limit:',sys.getrecursionlimit()
```

After setting the recursion limits, I did some timings to compare a BST (binary search tree) to an AVL tree.

```
recursion limit: 1000
numberOfItems: 5000 recursion limit: 10000

TIMING BST items adding in ascending order
Time to add 5000 items: 17.3619138438 seconds.
Tree height: 4999
================================================

TIMING AVL items adding in ascending order
Time to add 5000 items: 0.155706679748 seconds.
Tree height: 12
================================================

TIMING BST items adding in random order
Time to add 5000 items: 0.0982299918095 seconds.
Tree height: 27
================================================

TIMING AVL items adding in random order
Time to add 5000 items: 0.136984438094 seconds.
Tree height: 14
================================================
```

a) Why is the time to add 5000 items in ascending order to a BST so much more than an AVL tree?

b) Why is the time to add 5000 items in random order to a BST LESS than an AVL tree?

c) How would you expect the average, successful search times to compare for the BST and the AVL tree after adding 5000 items in random order?

d) What code did I add to measure the height of the trees?

2. Hashing Motivation and Terminology:

a) Sequential search of an array or linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.

If *n* is the number of items being searched, what is the average and worst case theta notation for a search?

average case $\Theta($)

worst case $\Theta($)

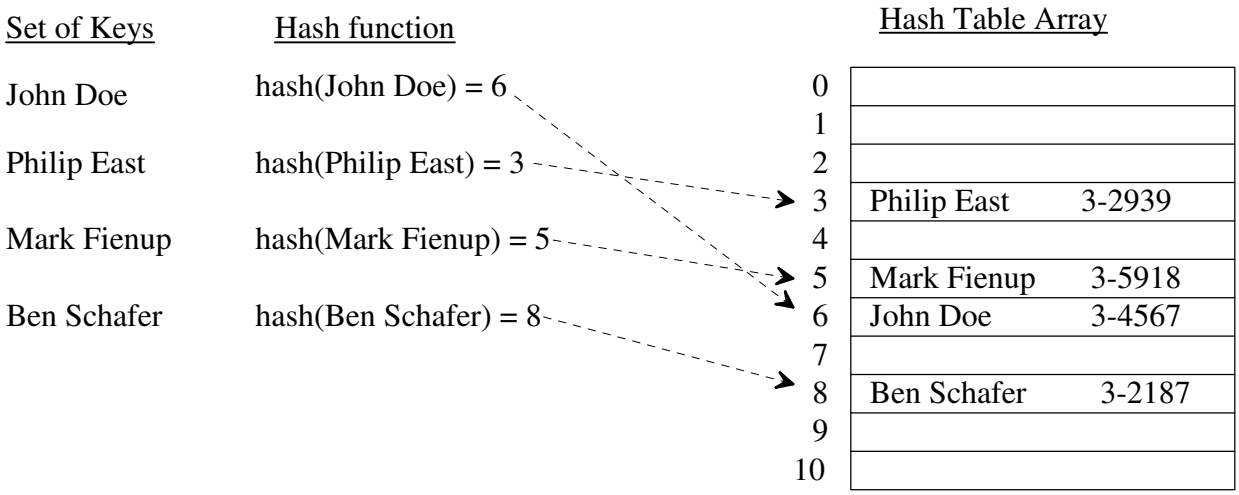
b) Similarly, binary search of a sorted array or AVL tree always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.

If *n* is the number of items being searched, what is the average and worst case theta notation for a search?

average case $\Theta($)

worst case $\Theta($)

Hashing tries to achieve average constant time (i.e., $O(1)$) searching by using the target's value to calculate where in the array (called the *hash table*) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target's *home address*) is the job of the *hash function*. A *perfect hash function* would take your set of target values and map each to a unique array index.



a) If *n* is the number of items being searched and we had a perfect hash function, what is the average and worst case theta notation for a search?

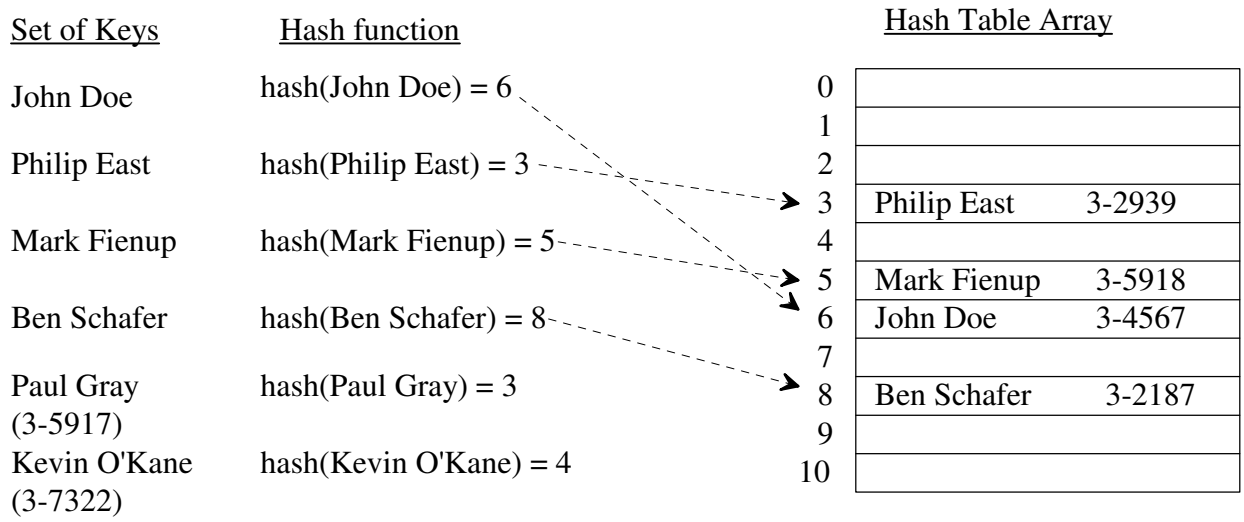
average case $\Theta($)

worst case $\Theta($)

3. Unfortunately, perfect hash functions are a rarity, so in general two or more target values might get mapped to the same hash-table index, called a *collision*.

Collisions are handled by two approaches:

- *chaining, closed-address*, or *external chaining*: all target values hashed to the same home address are stored in a data structure (called a *bucket*) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is a array of linked list (or whatever data structure is being used for the buckets)
- *open-address* with some *rehashing* strategy: Each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is linear probing where the hash table is scanned circularly from the home address until an empty hash table address is found.



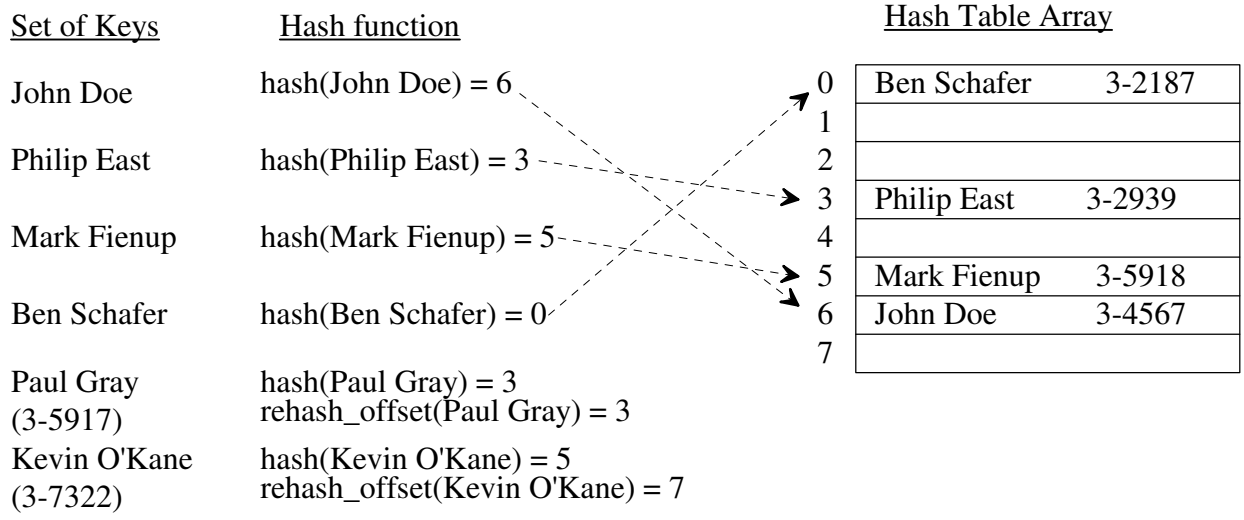
a) Assuming open-address with linear probing where would Paul Gray and Kevin O'Kane be placed?

b) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.

- *primary clustering* - keys mapped to a home address follow the same rehash pattern
- *secondary clustering* - rehash patterns from initially different home addresses merge together

Rehashing strategies			
Strategies	**Description**	**Suffers from:**	
		primary clustering	**secondary clustering**
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)		
quadratic probing	Check a square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2		
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size		

c) Assume quadratic probing, insert "Paul Gray" and "Kevin O'Kane" into the hash table.



d) Assume double hashing, insert “Paul Gray” and “Kevin O’Kane” into the hash table.

Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 0
Paul Gray (3-5917)	hash(Paul Gray) = 3 rehash_offset(Paul Gray) = 1
Kevin O'Kane (3-7322)	hash(Kevin O'Kane) = 4 rehash_offset(Kevin O'Kane) = 3

Hash Table Array