

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()
```

The above was needed to run the `lab9/timeBSTandAVL.py` program which adds 5,000 items into initial empty BST and AVL trees in both sorted order and random order. My results were:

	5,000 items added in sorted order		5,000 items added in random order	
	BST	AVL Tree	BST	AVL Tree
Total time to perform adds	17.8 seconds	0.161 seconds	0.099 seconds	0.135 seconds
Height of resulting tree	4,999	12	29	14

a) 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?

b) What traversal (preorder, inorder, postorder) works 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 $\Theta(\quad)$

worst case $\Theta(\quad)$

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 $\Theta(\quad)$

worst case $\Theta(\quad)$

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.

<u>Set of Keys</u>	<u>Hash function</u>		<u>Hash Table Array</u>
John Doe	hash(John Doe) = 6	0	
		1	
Philip East	hash(Philip East) = 3	2	
		3	Philip East 3-2939
Mark Fienup	hash(Mark Fienup) = 5	4	
		5	Mark Fienup 3-5918
Ben Schafer	hash(Ben Schafer) = 8	6	John Doe 3-4567
		7	
		8	Ben Schafer 3-2187
		9	
		10	

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 $\Theta(\quad)$

worst case $\Theta(\quad)$

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 an array of linked lists (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 specific 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.

<u>Set of Keys</u>	<u>Hash function</u>		<u>Hash Table Array</u>
John Doe	hash(John Doe) = 6	0	
		1	
Philip East	hash(Philip East) = 3	2	
		3	Philip East 3-2939
Mark Fienup	hash(Mark Fienup) = 5	4	
		5	Mark Fienup 3-5918
Ben Schafer	hash(Ben Schafer) = 8	6	John Doe 3-4567
		7	
Paul Gray (3-5917)	hash(Paul Gray) = 3	8	Ben Schafer 3-2187
		9	
Kevin O'Kane (3-7322)	hash(Kevin O'Kane) = 4	10	

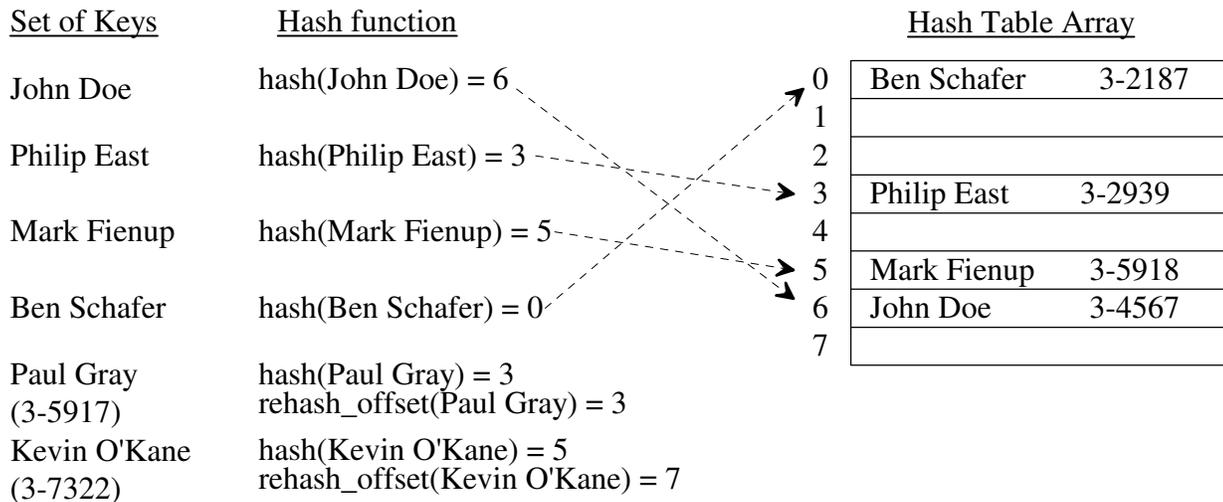
a) Assuming open-address with linear probing where would Paul Gray and Kevin O'Kane be placed?

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

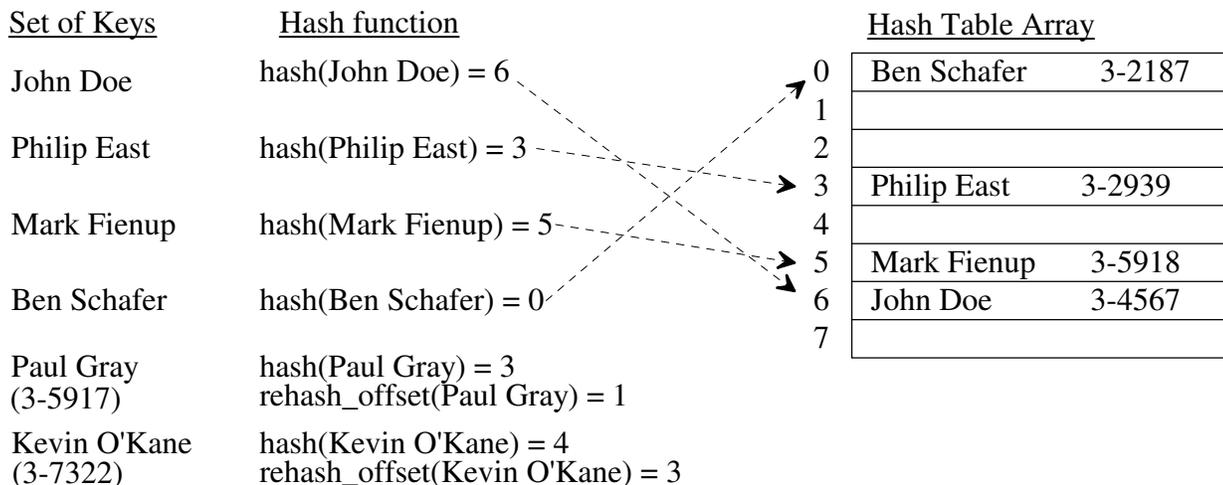
- *primary clustering* - keys mapped to a home address follow the same rehash pattern
- *secondary clustering* - rehash patterns from initially different home addresses merge together

Rehash Strategy	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.



Below from: <http://research.cs.vt.edu/AVresearch/hashing/index.php>

Quadratic Probing:

Another probe function that eliminates primary clustering is called **quadratic probing**. Here the probe function is some quadratic function $p(K, i) = c_1 i^2 + c_2 i + c_3$ for some choice of constants c_1 , c_2 , and c_3 .

The simplest variation is $p(K, i) = i^2$ (i.e., $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$). Then the i th value in the probe sequence would be $(h(K) + i^2) \bmod M$. Under quadratic probing, two keys with different home positions will have diverging probe sequences. For example, given a hash table of size $M = 101$, assume for keys k_1 and k_2 that $h(k_1) = 30$ and $h(k_2) = 29$. The probe sequence for k_1 is 30, then 31, then 34, then 39. The probe sequence for k_2 is 29, then 30, then 33, then 38. Thus, while k_2 will probe to k_1 's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Try inserting numbers for yourself, and demonstrate how the probe sequences for diverge by inserting these numbers into a table of size 16: 0, 16, 32, 15, 31.

Unfortunately, quadratic probing has the disadvantage that typically not all hash table slots will be on the probe sequence. Using $p(K, i) = i^2$ gives particularly inconsistent results. For many hash table sizes, this probe function will cycle through a relatively small number of slots. If all slots on that cycle happen to be full, this means that the record cannot be inserted at all! For example, if our hash table has three slots, then records that hash to slot 0 can probe only to slots 0 and 1 (that is, the probe sequence will never visit slot 2 in the table). Thus, if slots 0 and 1 are full, then the record cannot be inserted even though the table is not full! A more realistic example is a table with 105 slots. The probe sequence starting from any given slot will only visit 23 other slots in the table. If all 24 of these slots should happen to be full, even if other slots in the table are empty, then the record cannot be inserted because the probe sequence will continually hit only those same 24 slots.

Fortunately, it is possible to get good results from quadratic probing at low cost. The right combination of probe function and table size will visit many slots in the table. In particular, if the hash table size is a prime number and the probe function is $p(K, i) = i^2$, then at least half the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will be found. Alternatively, if the hash table size is a power of two and the probe function is $p(K, i) = (i^2 + i)/2$, then every slot in the table will be visited by the probe function.

Double Hashing

Both pseudo-random probing and quadratic probing eliminate primary clustering, which is the name given to the the situation when keys share substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will always follow the same probe sequence for every collision resolution method that we have seen so far. The probe sequences generated by pseudo-random and quadratic probing (for example) are entirely a function of the home position, not the original key value. This is because function p ignores its input parameter K for these collision resolution methods. If the hash function generates a cluster at a particular home position, then the cluster remains under pseudo-random and quadratic probing. This problem is called secondary clustering.

To avoid secondary clustering, we need to have the probe sequence make use of the original key value in its decision-making process. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have that constant be determined by a second hash function, h_2 . Thus, the probe sequence would be of the form $p(K, i) = i * h_2(K)$. This method is called double hashing.

A good implementation of double hashing should ensure that all of the probe sequence constants are relatively prime to the table size M . This can be achieved easily. One way is to select M to be a prime number, and have h_2 return a value in the range $1 \leq h_2(K) \leq M-1$. Another way is to set $M = 2^m$ for some value m and have h_2 return an odd value between 1 and 2^m .