6.006 Homework	**Problem set 2**	# 1 – Hash functions and load
Xola Ntumy		March 6, 2011
Collaborators: INSERT COLLABORATORS HERE		

(a) The algorithm will not have the same performance as the one in lecture. Provided the size of her hash table is prime number, the hash function will assign keys randomly. However, there will be the possibility of collisions because some phrases may be repeated and some words may have the same characters (letters). Also words containing the same characters but with different permutations would hash to the same value causing more collisions. The solution would be either to result to chaining or to use a more efficient hash function. For instance a hash function where each letter is multiplied by a base raised to a power depending on its position so that different permutations would result in different hash values.

(b) This algorithm will not have the same performance as the one in lecture. The reason for this is that the size of the table is a power of two, this makes h(k) the p lowest-order bits of k. Permuting the characters of k will not change the value of h(k). Also the items are in a contiguous blocks which means the hashes are not being assigned randomly. The solution would be to use a more appropriate size for his table, preferably a prime number.

(c) The problem with setHash is that it does not return the same hash value for the same set of integers that is ordered differently. Hence a solution would be to take the sum of the integers mod the size of the set.

(e) Starting out with a table of size 8 and a set of 4 values, the load factor is 0.5 As it turns out, any load factor can reliably be achieved since the load factor is simply the proportion of values to the size of the table. Experimenting with different sizes and sets of values is not necessary to show this.

(f) An example of such a situation is in the case of linear probing to resolve collisions. In such a case, we find that each value has equal probability of being placed in any slot (uniformity) but because of the clusters that are formed, the placement of the slots are not independent of each other.

(a) We must consider dynamically resizing our table because there will be an arbitrary sequence of inserts and deletes and there is the possibility that the table will become full. Therefore there must be a strategy to implement in the case where the table is full.

(b) On the other hand, resizing whenever we have a collision will be much too expensive since this will require rehashing all the slots in the table. We would like the time for an insert to be O(1) but it will be much worse.

(g) Chaining:

-Good way to resolve collisions, easy implementation

-However, worst case has runtime of O(n) and uses up a lot of space

Linear Probing:

-Uses up less space relatively since all slots are within the table

-The problem of clusters comes up and the placement of slots become less independent.

Quadratic Probing:

-Uses up less space relatively since all slots are within the table

-Slots are more independent since quadratic placement is more random than linear but still leads to a different form of clustering known as secondary clustering

Double Hashing:

-Much closer to the ideal scheme of uniform hashing

-More distinct probe sequences than linear or quadratic probing and hence one of the best methods available

(h) Dynamically resizing the table would involve rehashing all the values of the table now using n+m instead of m. The asymptotic time complexity would thus be O(n+m)

(i) The efficiency of such a solution would depend on the size of k. If k is a relatively small constant, it will not work; the hash table will be resized way too often. A large constant would do the opposite of course and the table would be resized too few times. A reasonable k would have to be near the size of the table, ie. m.

6.006 Homework	**Problem set 2**	# 3 – Python dictionaries
Xola Ntumy		March 6, 2011
Collaborators:		

(a) Examples of use cases are: -Passing keyword arguments

-Class method lookup

-Instance attribute lookup and global variables

-Builtins

-Uniquification

-Membership testing

-Dynamic mappings

(b) This use case will deploy one hash table perhaps of relatively large size, that contains the details of the given members. It will be created once and need not be altered significantly after its creation. Calls will often by made to ”has-key” or ”contains” to search out and test the validity of a supposed member.

(c) A multiplication hash function could be used. This might be suitable since we may not know the value of m (the size of the table) when we begin to assign values to slots and this is the advantage of this hash function; the value of m is not critical. Double hashing would be the best method of resolving collisions because it renders the most distinct probe sequences in comparison to linear or quadratic probing. The size of the hash table should preferably be a prime number, close to a power of two since we are using a multiplication hash function where the lowest order bits are significant.

(d) In the global variables use case, the requirements are a hash table of relatively small size where calls to insert, delete and search are made frequently. A division hash function may be used where collisions are resolved through chaining.

(e) PyDict-MINSIZE: This tunable concerns cache lines which come into play only practically but not theoretically. Maximum dictionary load in PyDict-SetItem: This tunable seems to be similar to the theoretical maximum load factor. Growth rate upon hitting maximum load: This is similar to the theoretical limit for dynamic resizing the table. Maximum sparseness: This tunable is more important practically than theoretically since we do not consider the need for freeing up memory per se in theory. Shrinkage rate upon exceeding maximum sparseness: This tunable is also more important practically than theoretically for similar reasons.

(f) PyDict-MINSIZE: A suitable m will be a power of 2 since a division hash will be used. Maximum dictionary load in PyDict-SetItem: Preferably between 0.5 and 1 so as to keep a decent proportion of used to unused slots. Growth rate upon hitting maximum load: This may be close to the size of the table. Maximum sparseness: This should be preferably more than half the size of the table. Shrinkage rate upon exceeding maximum sparseness: This could be any value.

6.006 Homework	**Problem set 2**	# 4 – Matching DNA sequences
Xola Ntumy		March 6, 2011
Collaborators: INSERT COLLABORATORS HERE		

(a) A rolling hash provides a more efficient way of hashing a sequence of nucleotides since the reading frame can slide which allows us to compare n-length strings in O(n) time.