This program is an exercise in *hashing*. Continuing with the *Address Book* theme used in the first 3 assignments, you will modify the table, node, and addressBook classes as necessary to implement a new version utilizing **Open Hashing with Chaining**. As discussed in lecture, chaining utilizes a hash table implemented as an array of references. Each element within the hash table points to a singly linked list containing the collisions associated with the respective array index. In other words, the hash produces the same index for all items within a given linked list.

As with the previous exercises, your program will be a simple address book. It will store names (used as keys) and addresses that are associated with the names. The program will present the user with a menu listing the choice of operations: *add* a name (and address), *look up* a name (displaying the associated address), *update* the address for a name, *delete* an

entry, *display all* entries, *save to a file* and *quit*. In the case of *display*, the *average probe length* (discussed below) will also be displayed.

If the user chooses *add* the program prompts for a name. If the name already exists in the address book, a message is displayed to that effect and no change is made. Otherwise the program asks for an associated address and the new entry is entered and stored. If the user chooses *look up,* (s)he is asked for a name. The program will look up the entry for that name and display the associated address. If the user chooses *update,* (s)he is asked for a name. The name is looked up and the associated address is displayed. The user is then prompted for a new address which is accepted and stored in place of the old address. For *delete,* the user is asked for a name, and the entry for that name is removed from the address book. In any of these cases (other than insert) if the name given by the user is not found, an appropriate message is displayed. Again, if the user selects *display,* all of the names and addresses in the address book are displayed (the order is not important) along with the *average probe length*. The program continues displaying the menu and taking commands until the user chooses *quit*.

As in HW#3, the program will first ask the user "Do you want to open a file? (y/n)." If the user answers "y" the program will ask for a file name, open, and read the input file. Each pair of lines from the input file will be made an entry in the address book: a name line and an address line. If the user answers "n" the program will not open a file. In either case the program will continue. A "save" operation will also appear in the menu. When the user selects "save" the program will ask the user for the name of a file in which to write the address book. Your program will open the specified file and write the entire address book to the file. This operation will be very similar to the " Display all entries" operation.

If the user enters a selection which is not on the menu, the menu will be re-displayed and the user is prompted again for a command. The names and addresses can be any strings -- your program need not check that they make any sense.

**Internals:**

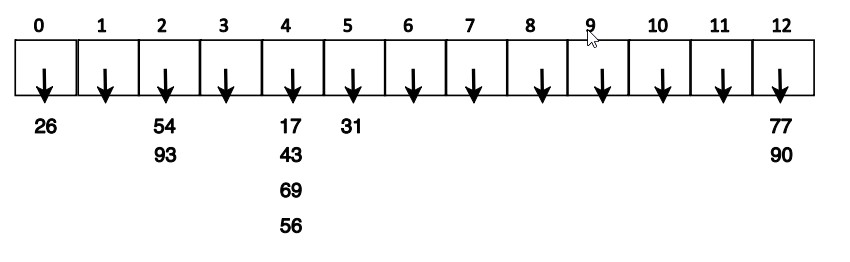
As discussed in class, **Separate Chaining** is ahashing technique in which each position within the hash table references a linked list (or other linked data structure) for handling collisions. Each position may be just a reference to the list, or may be an item and a link, essentially, the head of a list. In the latter, one item is in the table, and other colliding items are in the list. For this exercise, the table will consist of **references only**. A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hash to the same index are simply added to the linked list. In separate chaining it is normal to put N or more items into an N-cell array. Finding the initial cell takes O(1) time, however searching through a list takes time proportional to the number of items within the list.

For this assignment, use a Hash table of size: 10.

**Computing the average probe length:**

In lecture, we discussed the *average probe* metric with regard to *linear probing*.

The average probe length (and therefore the average access time) is dependent on the load factor, i.e. the ratio of items in the table to the size of the table. As the load factor increases, the average probe length increases. For *separate chaining*, we may compute the average probe length in the following manner:



(8 empty slots \* (1 probe/empty slot)) / 13 total slots

+

(2 slots with one element \* (2 probes for slot with one element)) /13 total slots

+

(2 slots with two elements \* (3 probes for slot with two elements)) /13 total slots

+

(1 slots with four elements \* (5 probes for slot with four elements)) /13 total slots

= 8/13 + 4/13 + 6/13 + 5/13

= 23/13

# = 1.77 Average Probe Length

Use the following Hash function:

public static int hash(String key, int tableSize)

{

int hashVal = 0;

for ( int i = 0; i < key.length(); i++ )

hashVal = 37 \* hashVal + key.charAt(i);

hashVal %= tableSize; if( hashVal < 0 ) //overflow case hashVal += tableSize;

return hashVal;

}

Rubric for HW4 (**10 points**):

* 1 point for overall program appearance/readability/comments.
* 1 point for correct implementation of the hash function
* 1 point for correct initial read-in of user specified data file.
* 1 point: Through experimentation, modify the Hash function to increase the

number of non-empty **Buckets** (i.e. linked lists) referenced from within the hash table). Discuss your results in a (short) separate document entitled

HW4 \_HashFunction\_Analysis

* 1 point (a piece) for the correct implementation of each of the following methods (total 6 points).

1. public boolean insert(String key, String value) lnserts a new entry to the table. If an entry already exists with the given key value make no insertion but return false.
2. public String lookUp(String key) L

Looks up the entry with the given key and returns the associated value. If no entry is found null is returned.

1. public boolean delete(String key)

Deletes the entry with the given key. If no entry is found returns false.

1. public boolean update(String key, String newValue)

Replaces the old value associated with the given key with the newValue string.

1. public void\_save()

Ask the user for the name of a file in which to write the address book. Open the file and write the entire address book to the file.

1. public int displayAll()

Displays Name/Address for each table entry, along with the average probe length. Returns total entry count.