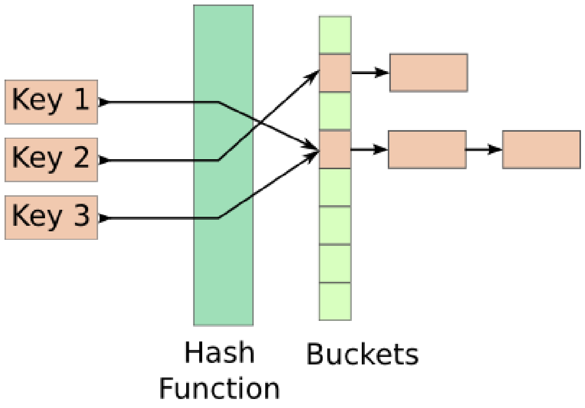
In this implementation of chain hash-table, the size of the table would be initialized at first, and it is assumed to be constant all the time. The table is an array of LinkedLists which are all initially empty.

For insertion of the values, the hashCode() method of the key is used to derive the index in the table where the value is stored. The following hash function is used to find the corresponding table index for a given key:

```
hash-value = Math.abs(key.hashCode()) % size;
```



The above figure shows an schematically view of a chain hash-table

If any other value in the table has the same key, it would be replaced and the old value would be returned by `put` method.

How to increase the efficiency?

- If the key is inserted for the fist time, the associated value would be stored in front of the corresponding LinkedList. This helps for more rapid retrieval when the most recent recent stored values are requested.
- For traversing the LinkedLists, each time I use an iterator in order to access to all the list values in constant time.

Get method, basically starts from the beginning of the corresponding LinkedList and penetrates through the list. It immediately return the found key and exit the method. Therefore, the most recent added values can be found more quickly since they in front of the list.

Efficiency Analysis

The execution time for put and get methods is a function of the loading factor of the hash-table. The loading factor is easily defined as the total number of inserted values divided by the size of the table. Therefore, increasing the loading factor results in more elements in the sub-linkelists and therefore inferences the number of comparison and list traversal.

To measure the efficiency of the the implemented hash-table, I have written a Simulation class, which uses the implemented hash-table. In this class, the size of the table is assumed to be 1,000,000 (one million). The table is filled with random integer numbers which are converted to strings. These strings are considered are treated as the keys. The integer numbers themselves are treated as the associated values.

First of all, the table is loaded based on the initial loading factor (# of inserted values/ table-size). Then, we put 20 random values into the table and get them again to measure the execution time. For loading factor L=10 and L=0.1, put and get process time is calculated in nano-second. The numerical results is reported in the following table. The first row is removed and the average and median of the execution time for 19 rows is calculated. As seen for the `put` method, by changing L from 0.1 to 10 the average execution time is changed from ~600 ns to ~2400 ns. For the `get` method, this is changed from ~550 ns to ~1950 ns. Typical execution time for the get method is slightly shorter than the put method, since for the get we don't have to necessarily check all the values in the linkedlist to find the desired key. But for the put method, all the values would be checked to see if there is any same key already stored in the list.

Chain probing hash-tables are more efficient compared to linear probing. The number of comparison for the put/get method is something like

```
c = 1+ L/2
```

So the execution time can be proportional to c in the above relation, plus the fact that we have to consider a constant time to define and initialize the variables each time. So we have

```
run-time = alpha * c + beta
```

where alpha and beta are constant and c is the same as the previous relation.

	L = 10			L = 0.1	
	put	get		put	get
	ns	ns		ns	ns
	14938	2705		1467	861
	3232	2413		726	559
	3015	11371		584	526
	1961	3094		644	8514
	2071	1992		722	568
	2625	1711		579	611
	2641	1165		635	554
	2815	1908		564	601
	2485	1240		1989	523
	2476	2287		590	575
	1932	1172		551	694
	4246	11363		617	556
	2371	2590		1624	519
	2389	1749		572	548
	1527	1293		688	562
	3371	1427		620	537
	1749	2702		623	552
	2323	10571		608	529
	1284	1942		667	559
	1856	866		610	543
average	**3065**	**3278**		**784**	**975**
median	**2433**	**1967**		**622**	**558**

Considering the median values for run-time of put/get method for L=10 and L=0.1, we get a constant value of beta=~300-400 nano-second, which sounds reasonable.

If beta = 0, the execution time should get almost 6 times larger when we change l from 0.1 to 10. So the execution time for L=10 must be something like ~6*600 = 3600 nano-sec which is about right but smaller. Considering the constant time for all other additional process (considering beta constant) can explain this discrepancy.

As a conclusion, increasing the loading factor (e.g. L>>1), increase the put/get execution time linearly (~ `O(n)`).