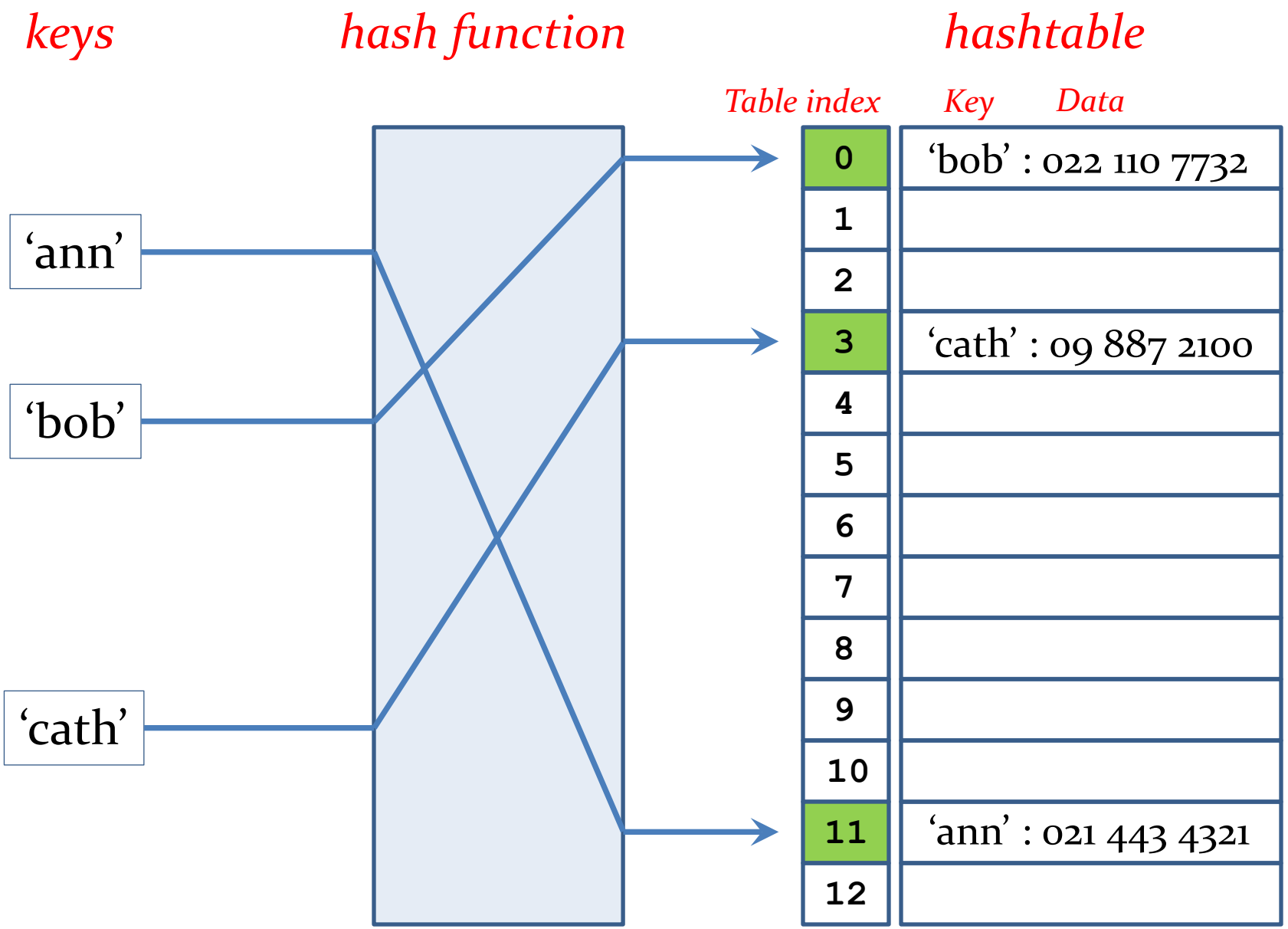
Hashtables

- A data structure for efficiently adding, removing and searching for items (all O(1))

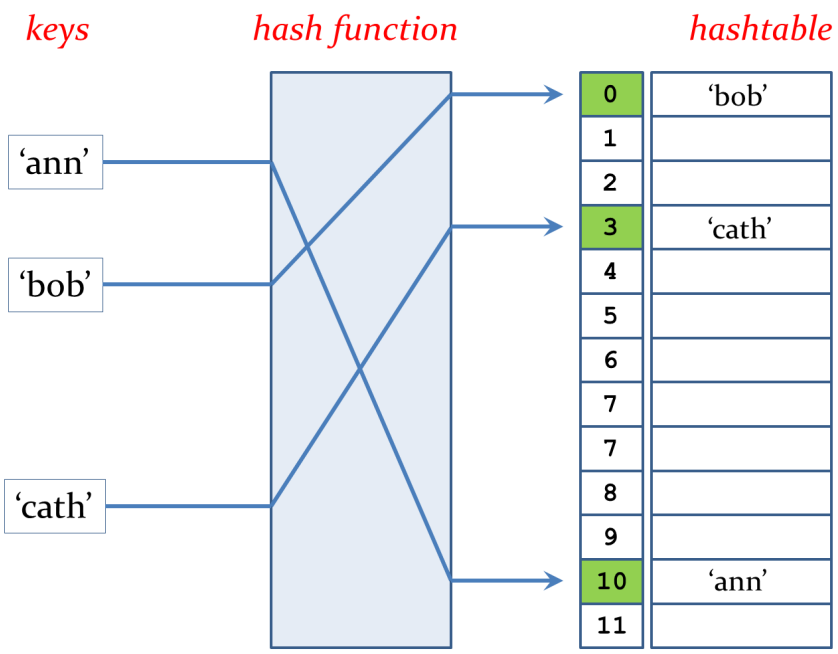
- But …
  - Requires a unique key for each item
  - Requires extra memory (if hash table becomes too full, operations become inefficient – in the worst case O(n) => Need to create a new, larger hash table and insert all items using hashing)
  - Requires a hash function for mapping keys to table indices
  - Requires collision handling if two keys map to the same table index

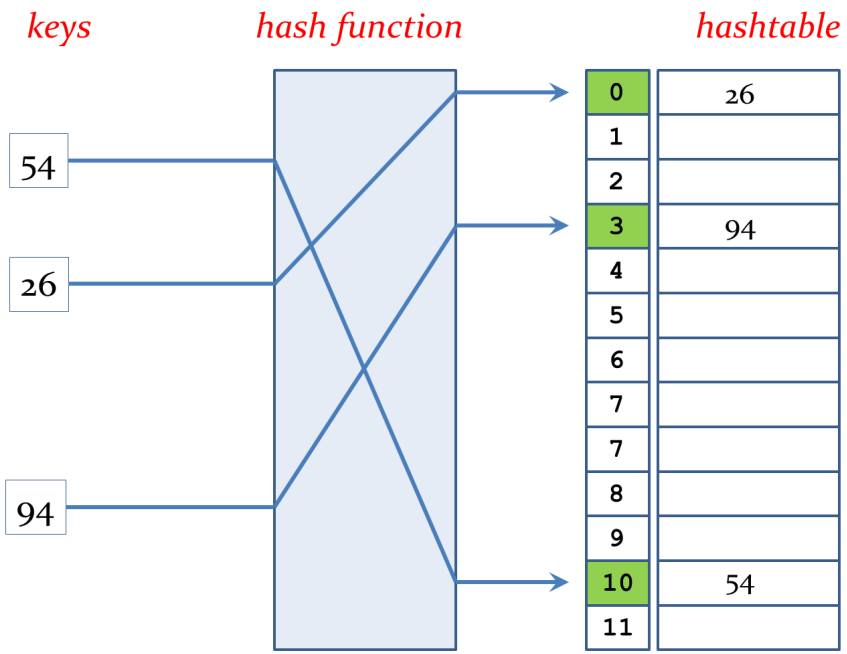
A conceptual view of a hashtable



Simplified Representation

- For our discussion of hashtables, we will disregard the *value* part of the *key/value* data, and just focus on the **keys**.
  - Keys will typically be either strings or integers:





Hash Functions

- The hash function maps the keys to index positions in the table
- A good hash function should be:
  - Effiently computable
  - Map the keys uniformly (i.e. when taking all possible keys, all table positions should be equally likely)

- Examples (if key is an integer)

  h1(*key*) = *key* % tableSize (where tableSize is a prime number)

  h2(*key*) = square the key, take all digits except the first one modulus the table size

Assumed table size is 13:
h1(27)=1
h2(27)=3
h3('cat')=0
(sum of ASCII codes is 99+97+116=312)

- Example (if key is a string)

  h3(*key*) = Add the ASCII values of each character in the key and take the sum modulus the table size

Collision Handling

- In **open addressing**, when a collision occurs, we continue to probe subsequent index positions in the table for a free slot
  - The sequence of locations that are examined is the *probe sequence*
- Linear probing
  - Search the hashtable sequentially, starting from the original location given by the hash function
    - Possible problem: primary clustering
- Quadratic probing
  - Search the hash table, trying increments of 1^2, 2^2, 3^2 and so on, always starting from the original location given by the hash function
    - Possible problem: secondary clustering
- Double hashing
  - Use two hash functions – if the main hash function generates a collision, then start from that location and consider every n^{th} location, where n is determined by a second hash function

Linear probing

- Properties
  - Simplest way to resolve a collision
  - Starting from the collision index, look sequentially until you find an empty slot
  - Wrap around to the start of the table when the end is reached
- Example: $h(key) = key \% 11$

Insert 45 => h(45)=1

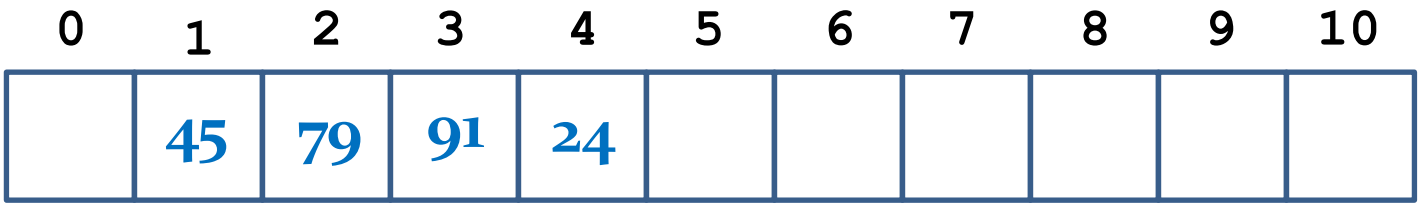
Insert 79 => h(79)=2

Insert 91 => h(91)=3

Insert 24 => h(24)=2 (collision)

(2+1) % 11 = 3 (collision)

(2+2) % 11 = 4



Quadratic probing

- Designed to avoid primary clustering
  - When a collision occurs, we advance the search for an empty position a considerable distance from the collision point
  - **Starting from the collision index**, we advance the probe distance by the square of the number of steps
- Example: $h(key) = key \% 11$

Insert 45 => h(45)=1

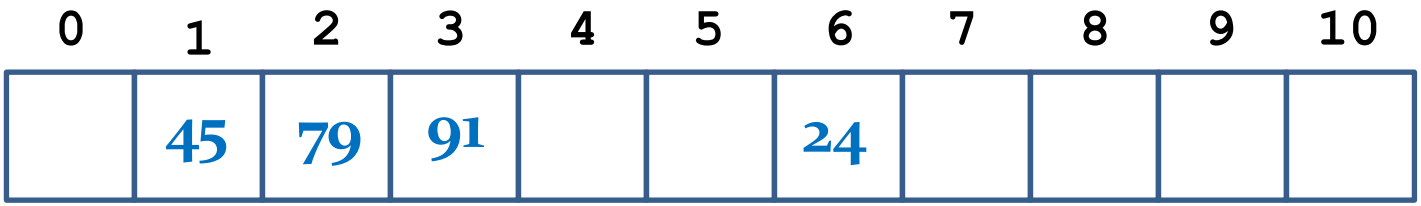
Insert 79 => h(79)=2

Insert 91 => h(91)=3

Insert 24 => h(24)=2 (collision)

(2+1) % 11 = 3 (collision)

(2+4) % 11 = 6



Double Hashing

- Double hashing aims to avoid both primary and secondary clustering
  - It does this by calculating a step value for the probing using a second hash function h'(key)
- Example: $h(key) = key \% 11$ $h'(key) = 7-(key \% 7)$

Insert 45 => h(45)=1

Insert 79 => h(79)=2

Insert 91 => h(91)=3

Insert 24 => h(24)=2 (collision)

h'(24)= 7 - (24%7) = 4

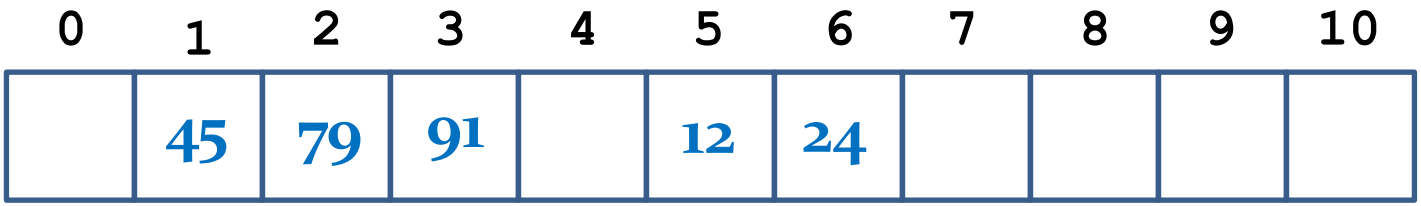
(2+ 1*4) % 11 = 6

Insert 12 => h(12)=1 (collision)

h'(12)= 7 - (12%7) = 2

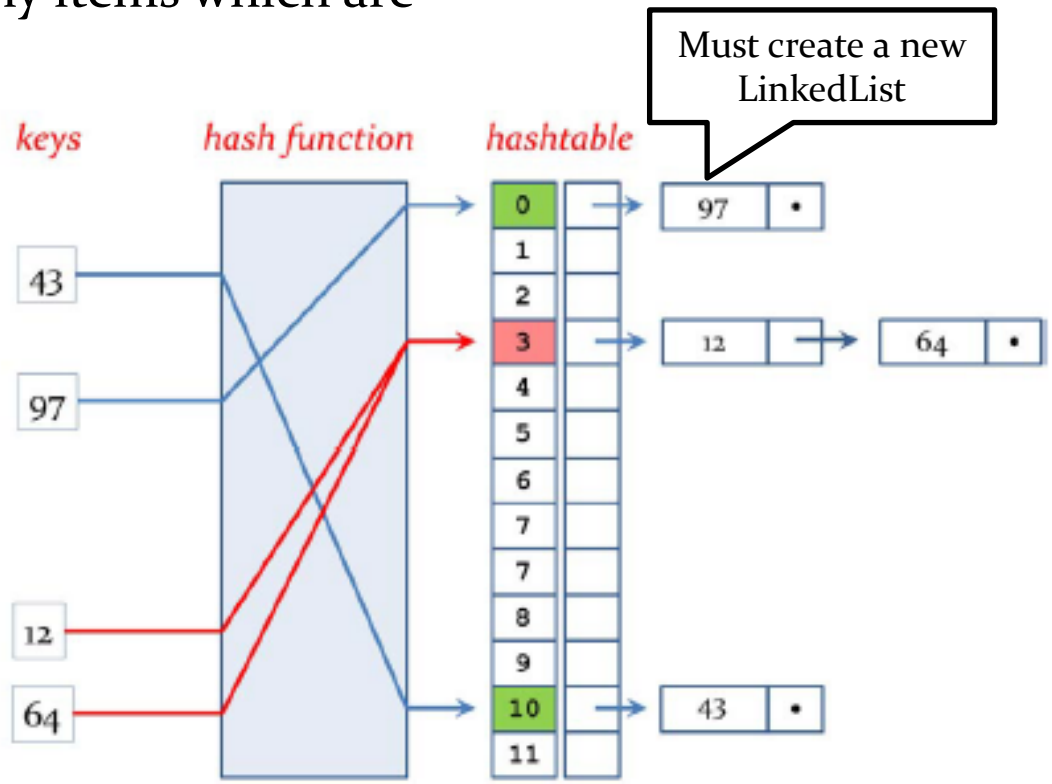
(1+ 1*2) % 11 = 3 collision

(1 +2*2) % 11 = 5



Separate Chaining

- In **separate chaining**, the hashtable consists of a list of linked lists
  - every element of the hashtable is a **linked list**
  - any items which are



Coderunner Tips

- Q6: Copy your SimpleHashTable class and rename it to QuadraticHashTable – you will need to modify your put method in this question
- Q7 – Same as with Q6
- Q8 – Sadly, you will need to make a class from scratch in this question
- Q8 – you do not need to paste your own implementation of the LinkedList class – this is provided for you
  BUT! It might be helpful to go over the LinkedList class you wrote in the last lab to remember what methods it has