Hashing

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

Outlines

- Hashing Basics
- Hash Functions
- Separate Chaining
- Open Addressing
- Bucket Addressing
- Rehashing
- Cuckoo Hashing

Review of Searching Techniques

- Recall the efficiency of searching techniques covered earlier.
- The sequential search algorithm takes time proportional to the data size, i.e, O(n).
- Binary search improves on liner search reducing the search time to O(log n).
- With a BST, an O(log n) search efficiency can be obtained; but the worst-case complexity is O(n).
- To guarantee the O(log n) search time, BST height balancing is required (i.e., AVL trees).

Review of searching Techniques (cont'd)

- The efficiency of these search strategies depends on the number of items in the container being searched.
- Search methods with efficiency independent on data size would be better.
- Consider the following class that describes a student record:

```
class StudentRecord {
   string name;     // Student name
   double height;   // Student height
   long id;         // Unique id
};
```

- The id field in this class can be used as a search key for records in the container.

Introduction to Hashing

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
  - A linked list implementation would take O(n) time.
  - A height balanced tree would give O(log n) access time.
  - Using an array of size 100,000 would give O(1) access time but will lead to a lot of space wastage. (direct access table)

- Is there some way that we could get O(1) access without wasting a lot of space?

- The answer is hashing.

Hashing

- Hashing is a technique to search efficiently by using hash functions.
- The main idea is to use key value pairs to store them in a table (hash table).
- The hash table contains values and each value is accessed via its hash address.
- The user of the hash table provides key and value.
- The key is passed through a hash function whose purpose is to provide an index (address) in the table for the key.
- Searching calculates the position of the key in the table based on the value of the key.

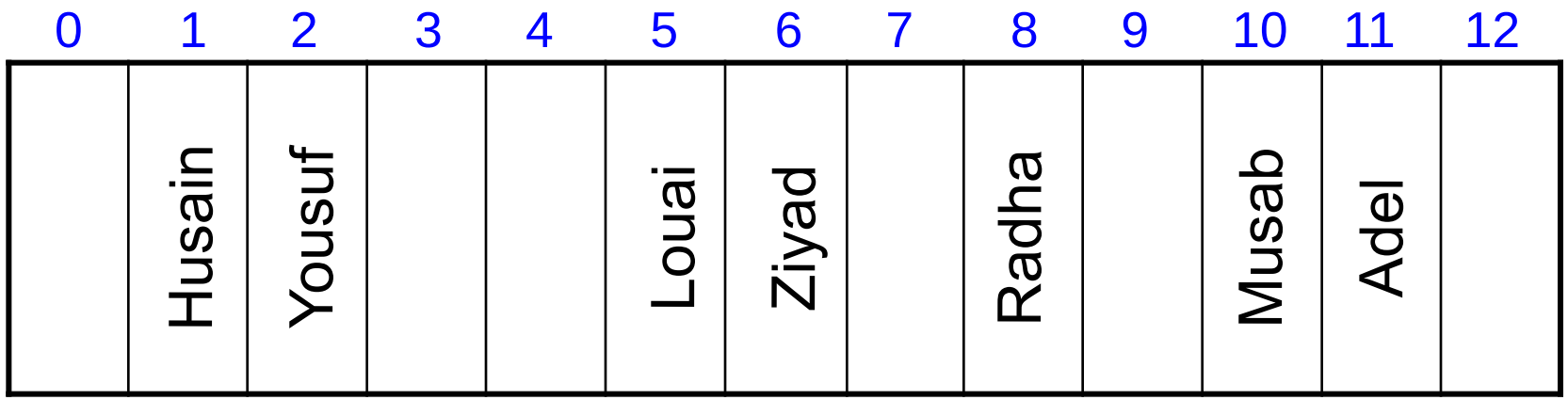
Example 1: Illustrating Hashing

- **Use the function `f(r) = r.id % 13` to load the following records into an array of size 13.**

Al-Otaibi, Ziyad	1.73	985926
Al-Turki, Musab Ahmad Bakeer	1.60	970876
Al-Saegh, Radha Mahdi	1.58	980962
Al-Shahrani, Adel Saad	1.80	986074
Al-Awami, Louai Adnan Muhammad	1.73	970728
Al-Amer, Yousuf Jauwad	1.66	994593
Al-Helal, Husain Ali AbdulMohsen	1.70	996321

Example 1: Introduction to Hashing (cont'd)

Name	ID	h(r) = id % 13
Al-Otaibi, Ziyad	985926	6
Al-Turki, Musab Ahmad Bakeer	970876	10
Al-Saegh, Radha Mahdi	980962	8
Al-Shahrani, Adel Saad	986074	11
Al-Awami, Louai Adnan Muhammad	970728	5
Al-Amer, Yousuf Jauwad	994593	2
Al-Helal, Husain Ali AbdulMohsen	996321	1



Hash Tables

There are two types of Hash Tables:

- An Open-addressed Hash Table is a one-dimensional array indexed by integer values that are computed by an index function called a hash function.
- A Separate-Chained Hash Table is a one-dimensional array of linked lists indexed by integer values that are computed by an index function called a hash function.
- Hash tables are sometimes referred to as scatter tables.
- Typical hash table operations are:
  - Initialization
  - Insertion
  - Searching
  - Deletion

Hash Table

- The load factor of a hash table is the ratio of the number of keys in the table to the size of the hash table.

- With open addressing, the load factor cannot exceed 1. With chaining, the load factor often exceeds 1.

- Note: The higher the load factor, the slower the retrieval.

Hash Functions

- A hash function, h, is a function which transforms a key from a set, K, into an index in a table of size n:

$$h: K \rightarrow \{0, 1, ..., n-2, n-1\}$$

- A key can be a number, a string, a record etc.
- The size of the set of keys, $|K|$, to be relatively very large.
- It is possible for different keys to hash to the same array location.
- This situation is called collision and the colliding keys are called synonyms.

Hash Functions

A good hash function should:

- Minimize collisions.
- Be easy and quick to compute.
- Distribute key values evenly in the hash table.
- Use all the information provided in the key.

Hashing Functions - Division Remainder

- Computes hash value from key using the % operator (using the table size as the divisor).
- *TSize = sizeof(table)*, as in *h(K) = K % TSize*, if K is a number.
- It is best if *TSize* is a prime number.
- Or prime numbers not close to powers of 2 are better table size values.
- Otherwise, h(K) = (K % p) % TSize for some prime p > TSize can be used.

Hash Functions - Truncation or Digit/Character Extraction

- Only a part of the key is used to compute the address. For id 123-45-6789, this method might use the first four digits, 1,234; the last four, 6,789; the first two combined with the last two, 1,289, etc.
- Chose portion such that the unchosen portion is not very helpful in distinguishing the keys.
- Or More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision and hence those parts of keys are not used.
- Very fast but digits/characters distribution in keys may not be very even.

Hash Functions - Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.
- To map the key 123-45-6789 to a range between 0 and 9999, we can:
  - Shift folding: split the number into three parts as 123, 456, and 789 and add these to obtain 1368 as the hash value.
  - Boundary folding: every other part is put in the reverse order.
- Very useful if we have keys that are very large.
- Fast and simple, especially for bit patterns.
- A great advantage is ability to transform non-integer keys into integer values.

Hash Functions - Radix Conversion

- Transforms a key into another number base to obtain the hash value.
- Typically use number base other than base 10 and base 2 to calculate the hash addresses.
- To map the key 55354 in the range 0 to 9999 using base 11 we have:

$$55354_{10} = 38652_{11}$$

- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.
- This value can then be divided modulo *Tsize* and resulting number becomes address in table.
- Collisions can’t be avoided.

Hash Functions - Mid-Square

- The key is squared and the middle part of the result taken as the hash value.
- To map the key 3121 into a hash table of size 1000, we square it 31212 = 9740641 and extract 406 as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

Hash Functions - Random-Number Generator

- Given a seed as parameter, the method generates a random number.

The algorithm must ensure that:

- It always generates the same random value for a given key.
- It is unlikely for two keys to yield the same random value.

- The random number produced can be transformed to produce a valid hash value.

Some Applications of Hash Tables

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

- **Browser Cashes**: Hash tables are used to implement browser cashes.

Problems for Which Hash Tables are not Suitable

1. **Problems for which data ordering is required.**
   Because a hash table is an unordered data structure, certain operations are difficult and expensive. Range queries, proximity queries, selection, and sorted traversals are possible only if the keys are copied into a sorted data structure. There are hash table implementations that keep the keys in order, but they are far from efficient.

2. Problems having **multidimensional data**.

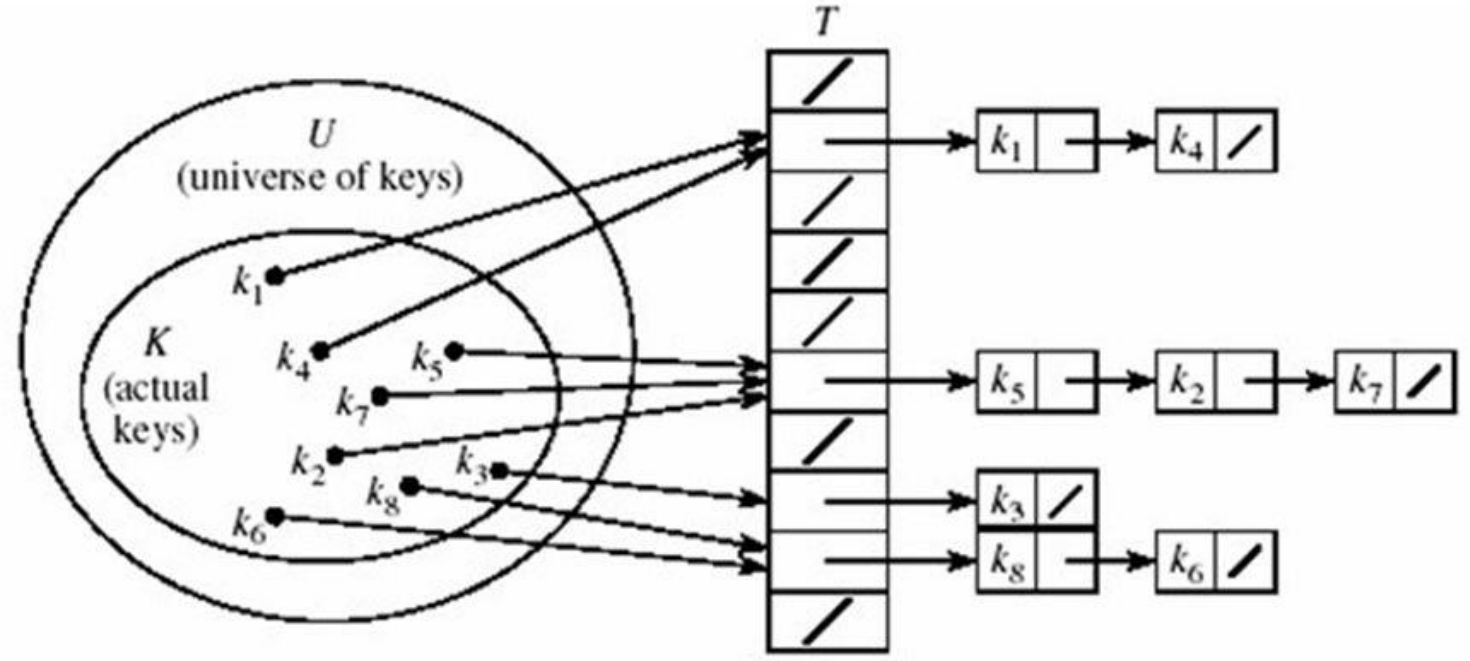
3. **Prefix searching** especially if the keys are long and of variable-lengths.

4. **Problems that have dynamic data**:
   Open-addressed hash tables are based on 1D-arrays, which are difficult to resize once they have been allocated. Unless you want to implement the table as a dynamic array and rehash all of the keys whenever the size changes. This is an incredibly expensive operation. An alternative is use a separate-chained hash tables or dynamic hashing.

5. **Problems in which the data does not have unique keys.**
   Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is use separate-chained hash tables.

Separate Chaining

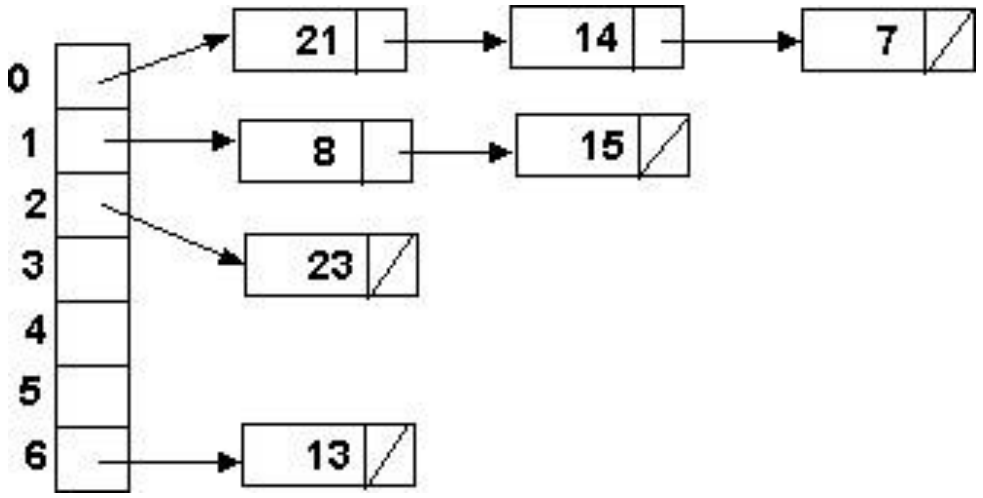
- The hash table is implemented as an array of linked lists.
- Inserting an item, r, that hashes at index i is simply insertion into the linked list at position i.
- Synonyms are chained in the same linked list.



Separate Chaining

- Retrieval of an item, r, with hash address, i, is simply retrieval from the linked list at position i.
- Deletion of an item, r, with hash address, i, is simply deleting r from the linked list at position i.
- **Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2
h(13) = 13 % 7 = 6
h(21) = 21 % 7 = 0
h(14) = 14 % 7 = 0 collision
h(7) = 7 % 7 = 0 collision
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1 collision



Separate Chaining with String Keys

- Recall that search keys can be numbers, strings or some other object.
- A hash function for a string s = c0c1c2…cn-1 can be defined as:

$$hash = (c_0 + c_1 + c_2 + \ldots + c_{n-1}) \% tableSize$$

this can be implemented as:

```
int hash(string key, int tableSize){
    int hashValue = 0;
    for (int i = 0; i < key.length(); i++){
        hashValue += key[i];
    }
    return hashValue % tableSize;
}
```

Example:

The following class describes commodity items:

```
class CommodityItem {
    String name;     // commodity name
    int quantity;    // commodity quantity needed
    double price;    // commodity price
}
```

Separate Chaining with String Keys

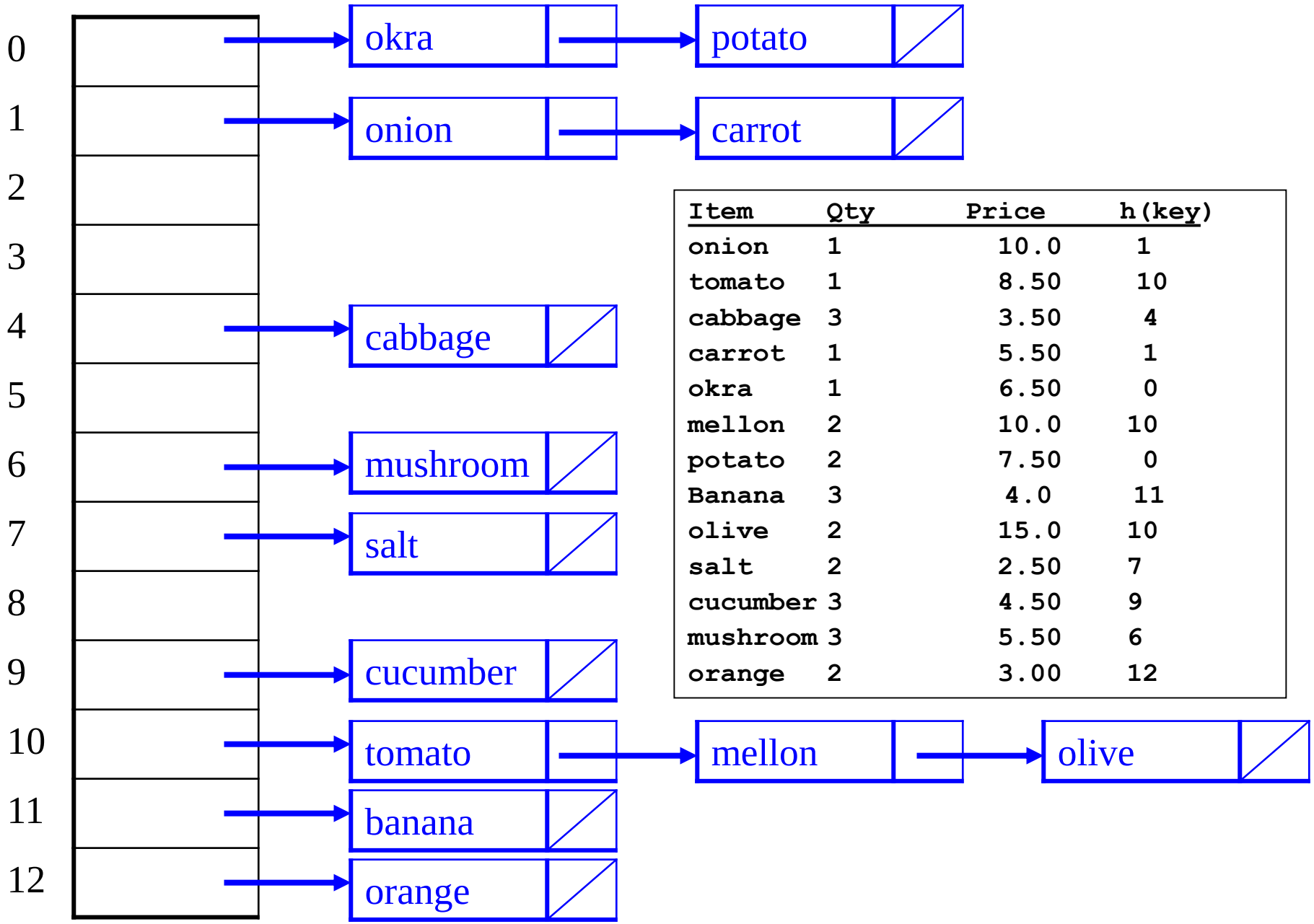
- Use the hash function **hash** to load the following commodity items into a hash table of size **13** using separate chaining:

```
onion        1        10.0
tomato       1        8.50
cabbage      3        3.50
carrot       1        5.50
okra         1        6.50
mellon       2        10.0
potato       2        7.50
Banana       3         4.00
olive        2        15.0
salt         2        2.50
cucumber     3        4.50
mushroom     3        5.50
orange       2        3.00
```

character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = 1
hash(salt) = (115 + 97 + 108 + 116) % 13 = 436 % 13 = 7
hash(orange) = (111 + 114 + 97 + 110 + 103 + 101)%13 = 636 %13 = 12

Separate Chaining with String Keys



Item	Qty	Price	h(key)
onion	1	10.0	1
tomato	1	8.50	10
cabbage	3	3.50	4
carrot	1	5.50	1
okra	1	6.50	0
mellon	2	10.0	10
potato	2	7.50	0
Banana	3	4.0	11
olive	2	15.0	10
salt	2	2.50	7
cucumber	3	4.50	9
mushroom	3	5.50	6
orange	2	3.00	12

Separate Chaining with String Keys

- Alternative hash functions for a string

  $s = c_0c_1c_2\ldots c_{n-1}$

exist, some are:

- hash = $(c_0 + 27 * c_1 + 729 * c_2)$ % tableSize
- It assumes at least three characters long key. The value 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2.

- hash = $(c_0 + c_{n-1} + s.length())$ % tableSize

- hash = $$[\sum_{k=0}^{s.length()-1} 26*k + s.charAt(k) - '']\%tableSize$$

Separate Chaining versus Open-addressing

Separate Chaining has several advantages over open addressing:

- Collision resolution is simple and efficient.
- The hash table can hold more elements without the large performance deterioration of open addressing (The load factor can be 1 or greater)
- The performance of chaining declines much more slowly than open addressing.
- Deletion is easy - no special flag values are necessary.
- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

Disadvantages of Separate Chaining:

- It requires the implementation of a separate data structure for chains, and code to manage it.
- The main cost of chaining is the extra space required for the linked lists.
- For some languages, creating new nodes (for linked lists) is expensive and slows down the system.

Open Addressing

- Open addressing hash tables store the records directly within the array.
- Initially all cells are Empty (NULL), when an element is inserted then the cell becomes Occupied (some valid value), when an element is removed it is marked as Deleted (set key = -1).
- While inserting, if a collision occurs, alternative cells are tried until an empty cell is found.
- Deletion: (lazy deletion): When a key is deleted the slot is marked as DELETED rather than EMPTY otherwise subsequent searches that hash at the deleted cell will fail.

Open Addressing

- Probe sequence: A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.
- A hash collision is resolved by probing, or searching through alternate locations in the array.

Types of probing:

- Linear probing
- Quadratic probing
- Double hashing

Open Addressing

- If collision occurs, alternative cells are tried.
- $h_0(key), h_1(key), h_2(key), \ldots$

$$h_i(key) = (Hash(key) + F(i)) \bmod TableSize$$

- Linear probing : $F(i) = i$
- Quadratic probing : $F(i) = i^2$
- Double hashing : $F(i) = i * Hash_2(key)$

Open Addressing: Linear Probing

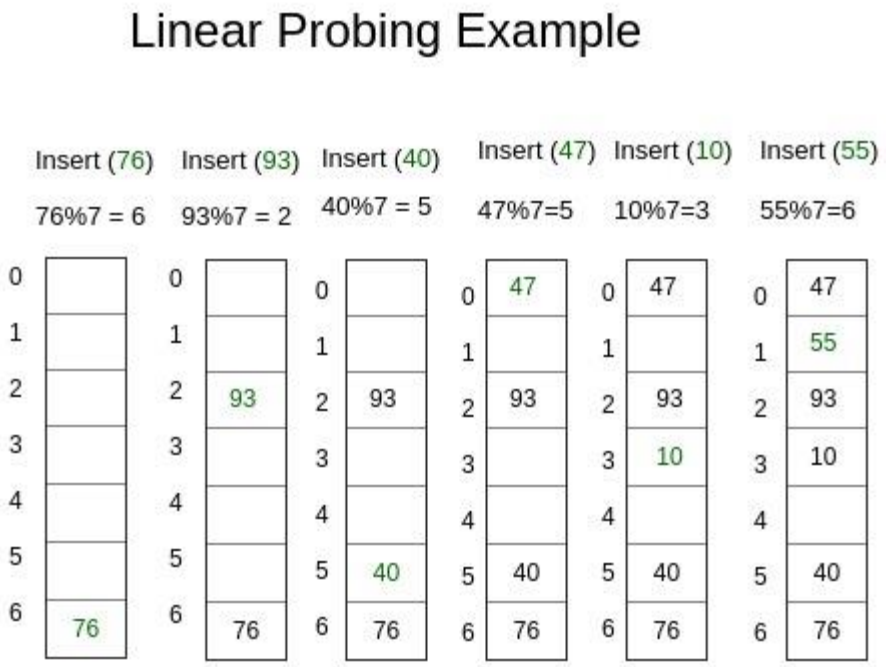
- Linearly probe slots by keeping the gap between each probe as 1.

Linear probing
F(i) = i
hi(X) = (Hash(X) + i) mod TableSize

h0(X) = (Hash(X) + 0) mod TableSize, // if slot is full, probe next
h1(X) = (Hash(X) + 1) mod TableSize, // if slot is full, probe next
h2(X) = (Hash(X) + 2) mod TableSize, ...

Example hash function with linear probing

```
int h(int i, int input)
{
  return (hash(input) + i)% HASHTABLESIZE;
}
```



Linear Probing Example Run

OPERATION	PROBE SEQUENCE	COMMENT
insert(18)	$h_0(18) = (18 \% 13) \% 13 = 5$	SUCCESS
insert(26)	$h_0(26) = (26 \% 13) \% 13 = 0$	SUCCESS
insert(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS
insert(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	COLLISION
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
find(15)	$h_0(15) = (15 \% 13) \% 13 = 2$	FAIL because location 2 has **Empty** status
find(48)	$h_0(48) = (48 \% 13) \% 13 = 9$	COLLISION
	$h_1(48) = (9 + 1) \% 13 = 10$	COLLISION
	$h_2(48) = (9 + 2) \% 13 = 11$	FAIL because location 11 has **Empty** status
remove(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS because location 9 contains 35 and the status is **Occupied** The status is changed to **Deleted**; but the key 35 is not removed.
find(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	The search continues, location 9 does not contain 9; but its status is **Deleted**
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
insert(64)	$h_0(64) = (64 \% 13) \% 13 = 12$	SUCCESS
insert(47)	$h_0(47) = (47 \% 13) \% 13 = 8$	SUCCESS
find(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	FAIL because location 9 contains 35 but its status is **Deleted**

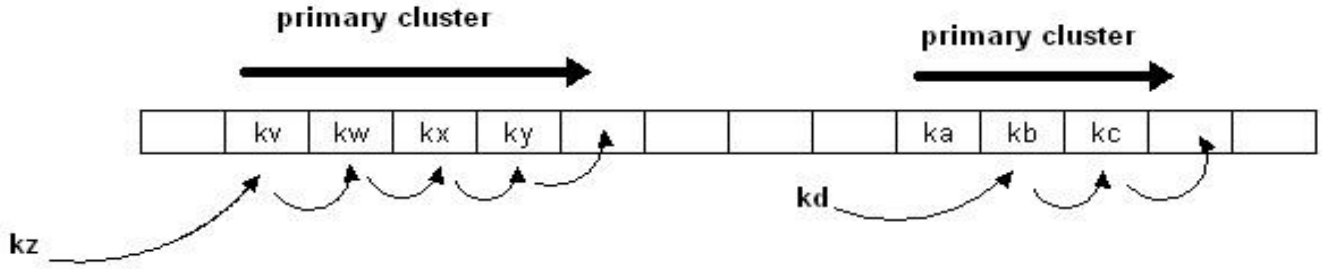
Index	Status	Value
0	O	26
1	E	
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	D	35
10	O	9
11	E	
12	O	64

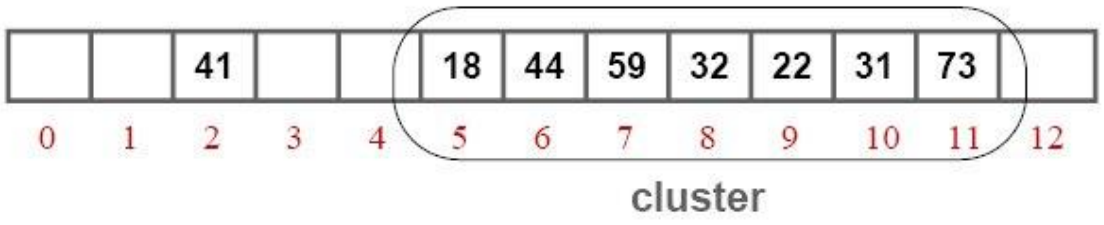
Disadvantage of Linear Probing: Primary Clustering

- Linear probing is subject to a primary clustering phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.

Example of a primary cluster: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **c(i) = i**:

h(18) = 5
h(41) = 2
h(22) = 9
h(44) = 5+1
h(59) = 7
h(32) = 6+1+1
h(31) = 5+1+1+1+1+1
h(73) = 8+1+1+1





Open Addressing: Quadratic Probing

- We look for i^2'th slot in i'th iteration.

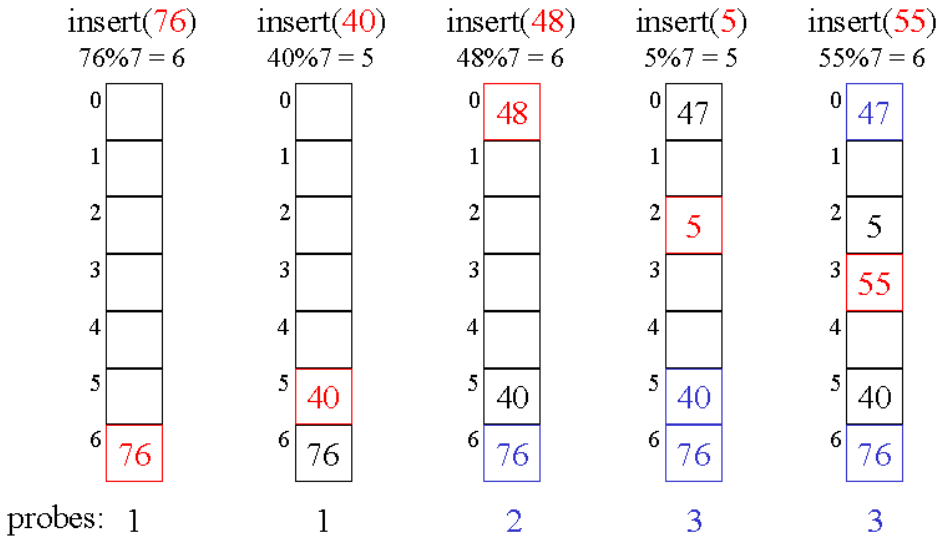
F(i) = i^2

h_i(X) = (Hash(X) + i^2) mod TableSize

h_0(X) = (Hash(X) + 0^2) mod TableSize, // if slot is full, probe next
h_1(X) = (Hash(X) + 1^2) mod TableSize, // if slot is full, probe next
h_2(X) = (Hash(X) + 2^2) mod TableSize, ...

Example hash function with quadratic probing

```
int h(int i, int input)
{
   return (hash(input) + i * i) % HASHTABLESIZE;
}
```



Open Addressing: Double Probing

- We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

F(i) = i * Hash_2(X)

h_i(X) = (Hash(X) + i * Hash_2(X)) mod TableSize

h_0(X) = (Hash(X) + 0 * Hash_2(X)) mod TableSize, // if slot is full, probe next
h_1(X) = (Hash(X) + 1 * Hash_2(X)) mod TableSize, // if slot is full, probe next
h_2(X) = (Hash(X) + 2 * Hash_2(X)) mod TableSize, ...

Example hash function with double probing

```
int h(int i, int input)
{
        return (hash(input) + i * hash_2(input) % HASHTABLESIZE;
}
```

Open Addressing: Double Probing

A Good Double Hash Function…

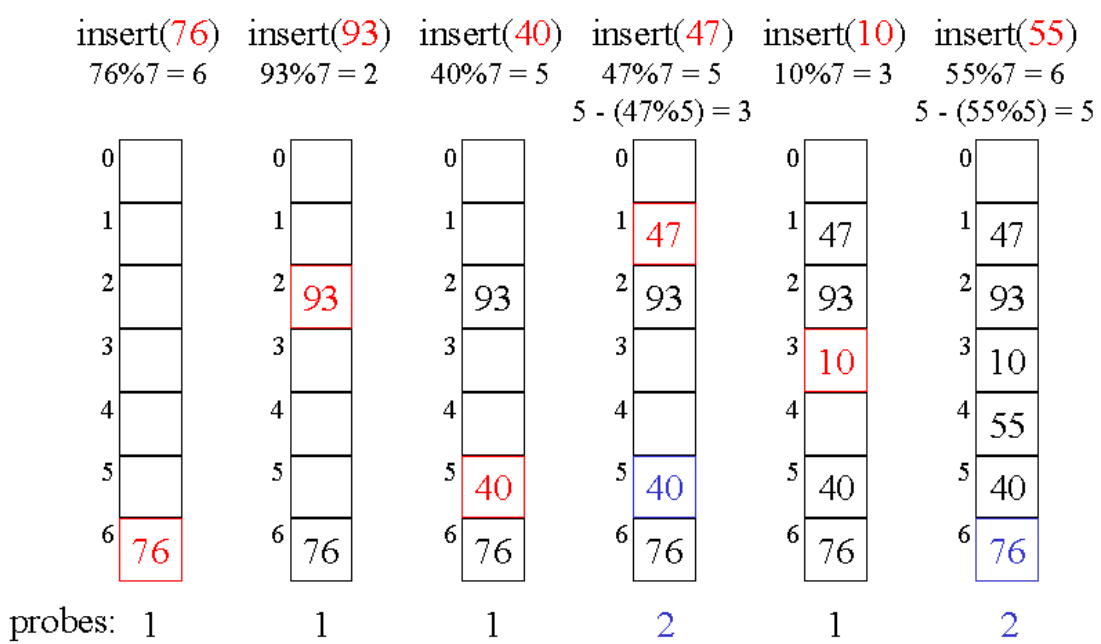
…is quick to evaluate.

…differs from the original hash function.

…never evaluates to 0 (mod size).

One good choice is to choose a prime R < size and:

$hash_2(x) = R - (x \bmod R)$



Open Addressing

- **Advantages of Open addressing:**
  - All items are stored in the hash table itself. There is no need for another data structure.
  - Open addressing is more efficient storage-wise.

- **Disadvantages of Open Addressing:**
  - The keys of the objects to be hashed must be distinct.
  - Dependent on choosing a proper table size.
  - Requires the use of a three-state (Occupied, Empty, or Deleted) flag in each cell.

Open Addressing Facts

- In general, primes give the best table sizes.
- With any open addressing method of collision resolution, as the table fills, there can be a severe degradation in the table performance.
- Load factors between 0.6 and 0.7 are common.
- Load factors > 0.7 are undesirable.
- The search time depends only on the load factor, not on the table size.
- We can use the desired load factor to determine appropriate table size:

$$\text{table size} = \text{smallest prime} \geqslant \frac{\text{number of items in table}}{\text{desired load factor}}$$

Bucket Addressing

- Another solution to the collision problem is to store colliding elements in the same position in the table.
- This can be achieved by associating a *bucket* with each address.
- A bucket is a block of space large enough to store multiple items.
- By using buckets, the problem of collisions is not totally avoided. If a bucket is already full, then an item hashed to it has to be stored somewhere else.
- By incorporating the open addressing approach, the colliding item can be stored in the next bucket if it has an available slot when using linear probing.

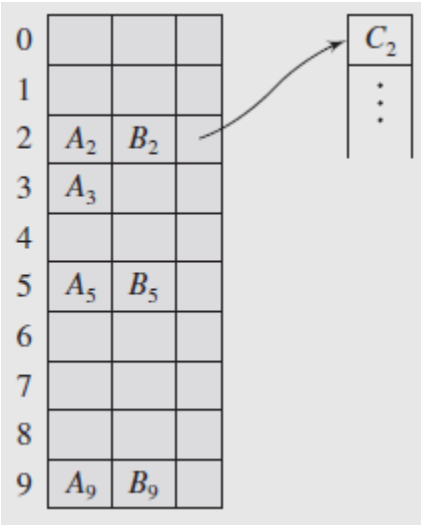
Bucket Addressing

Insert: $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$

0		
1		
2	A_2	B_2
3	A_3	C_2
4		
5	A_5	B_5
6		
7		
8		
9	A_9	B_9

Bucket Addressing

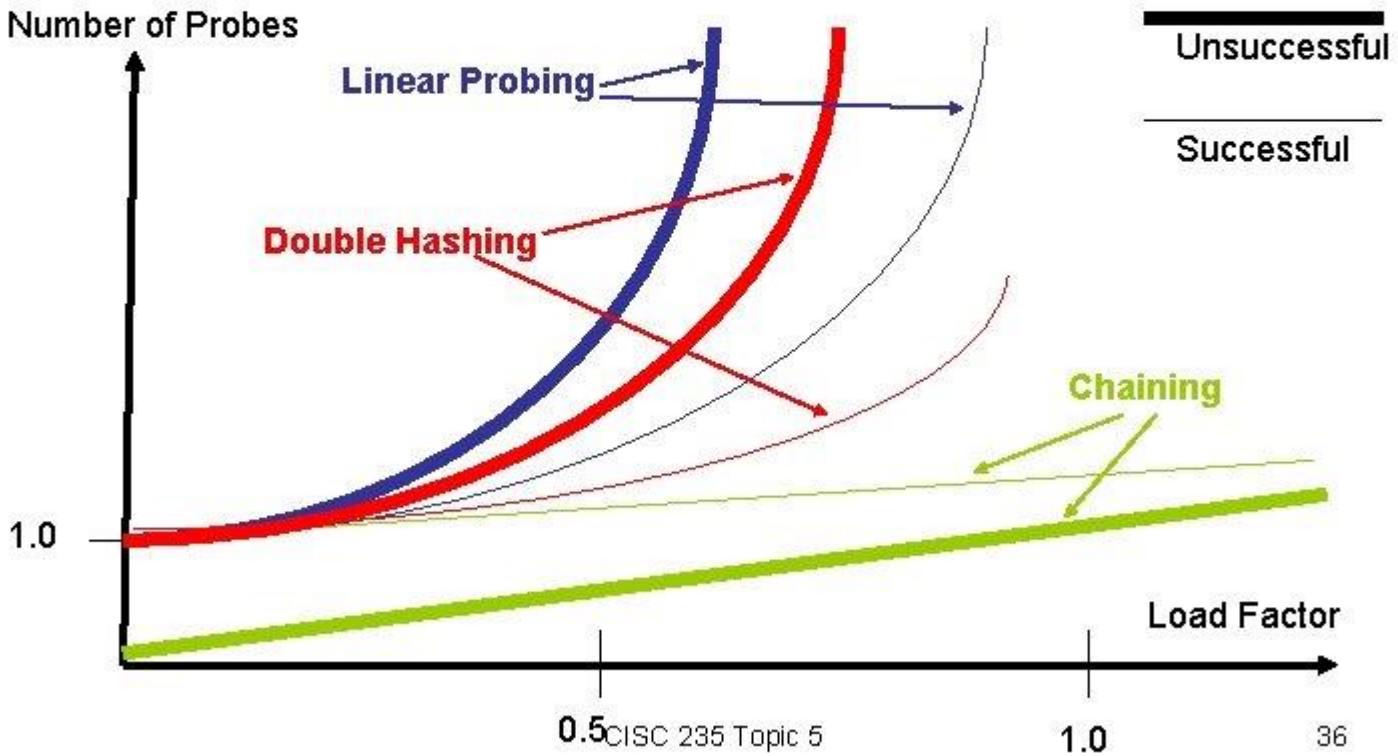
- The colliding items can also be stored in an overflow area.
- In this case, each bucket includes a field that indicates whether the search should be continued in this area or not.
- It can be simply a yes/no marker. In conjunction with chaining, this marker can be the number indicating the position in which the beginning of the linked list associated with this bucket can be found in the overflow area.



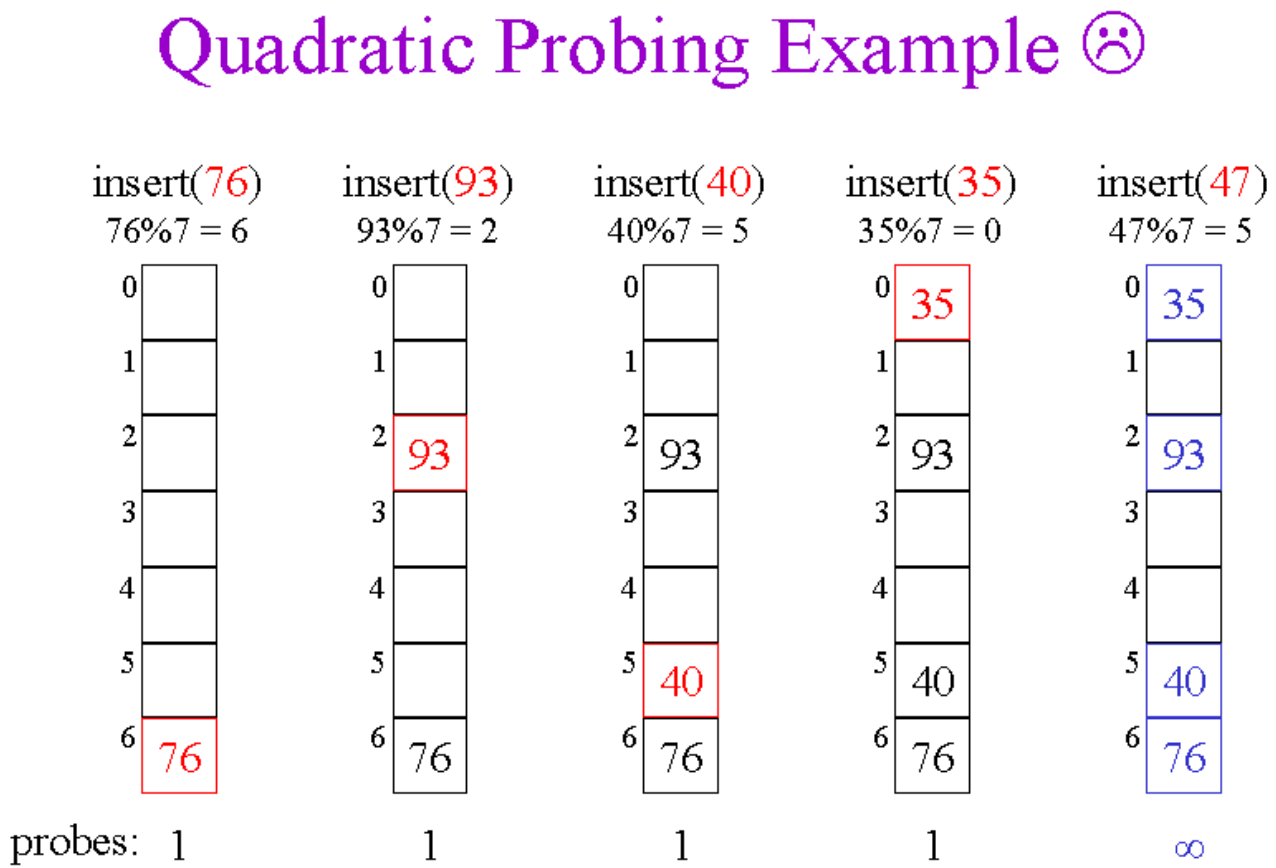
Rehashing

- When hash table reaches a certain saturation level, hashing becomes slow, requiring many tries to locate an item.
- Apply rehashing when the load factor goes above a predefined level (say, 70%).
- In rehashing, allocate a large table and copy the entries from the old table to new table using a new hash function.
- Note, we use table size in computing a hash function. Therefore, the new hash function should use the new table size.
- New size can be a prime number which is the first prime number greater than double the size of old table.
- This way, load factor will be decreased again.

Expected Number of Probes vs. Load Factor



Worst Situation (High load)



Rehashing

- It can be done is two simple steps,

1. If the load factor is above the threshold, make a new table (array).
2. Traverse through the old table, for each entry, call the insert function to insert it in the new table using new hash function.

Cuckoo Hashing

- A method for which rehashing is particularly important.
- It uses two tables, $T1$ and $T2$, and two hash functions, $h1$ and $h2$.
- To insert a key $K1$, table $T1$ is checked with function $h1$. If position $T1[h1(K1)]$ is free, the key is inserted there. If the position is occupied by a key $K2$, then $K2$ is removed to make room for $K1$ and then an attempt is made to place $K2$ with the second hash function in the second table in position $T2[h2(K2)]$. If this position is occupied by a key $K3$, then $K3$ is moved out to make room for $K2$ and then an attempt is made to place $K3$ in position $T1[h1(K3)]$.
- The arriving key (the original key or the key being pushed out to the opposite table) has a priority over a key occupying the former's position.

Cuckoo Hashing

- It can lead to an infinite loop if the very first position that was tried is tried again.
- Also, if the tables are full then tries will be unsuccessful.
- Solution: use rehashing. Set a limit on the number of tries. If limit is exceeded, rehash to create two new and larger tables, define two new hash functions and rehash keys from the old tables to the new ones.