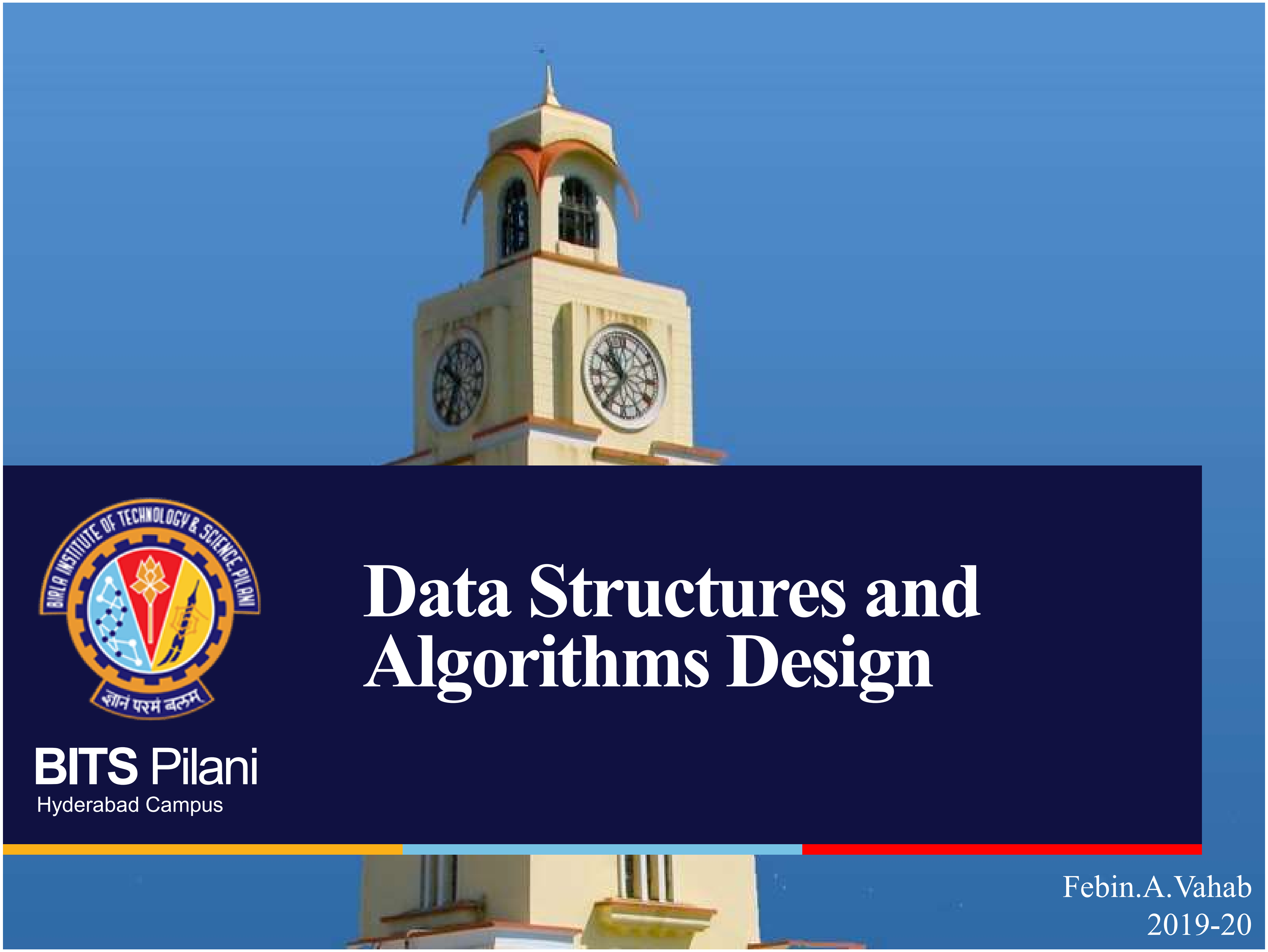


BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

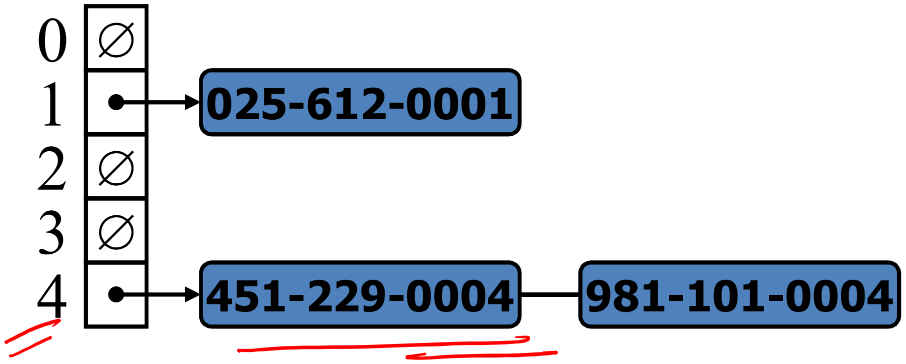
Febin.A.Vahab
2019-20

SESSION 7 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
7	Methods for Collision Handling: Separate Chaining, Notion of Load Factor, Rehashing, Open Addressing [Linear; Quadratic Probing, Double Hash]	T1: 2.5

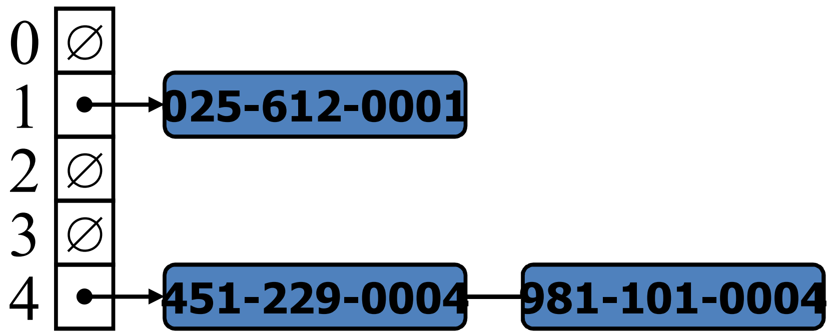
Collision- Handling Schemes

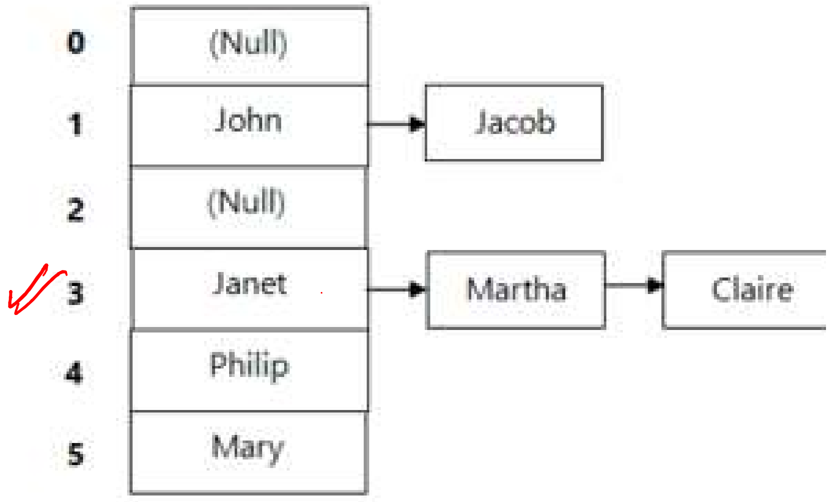
- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of elements that map there



Collision- Handling Schemes

- Chaining is simple, but requires additional memory outside the table

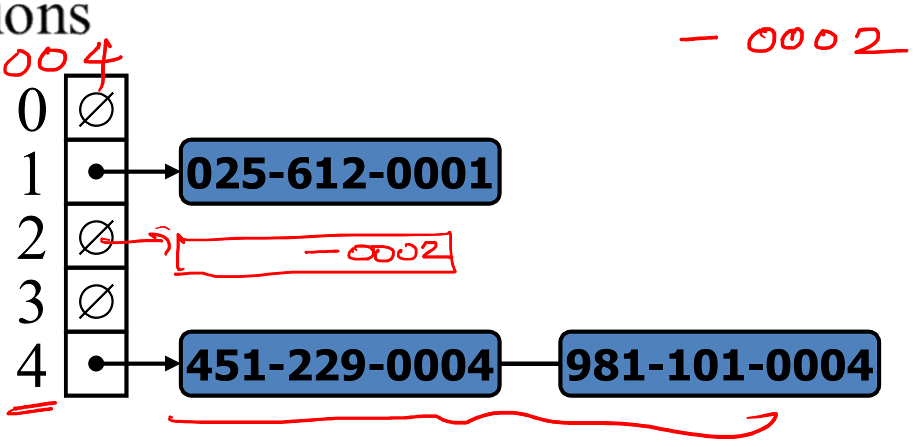




Separate Chaining

- A simple and efficient way for dealing with collisions is to have each bucket A [i] store a reference to a list that stores all the items that our hash function has mapped to the bucket A [i]
- Fundamental dictionary operations
- findElement (k) : 361-243-0004

```
B ← A[h (k)]
if B is empty then
return NO_SUCH_K EY
else
{ search for the key k in the sequence for this bucket}
return B.findElement(k)
```



Separate Chaining

- Fundamental dictionary operations
- insertltem(k, e) :

```
if A [h (k)] is empty then
    Create a new initially empty, sequence-based dictionary B
    A [h (k)] ←B
else
    B ← A[h (k)]
B . insertltem(k, e)
```

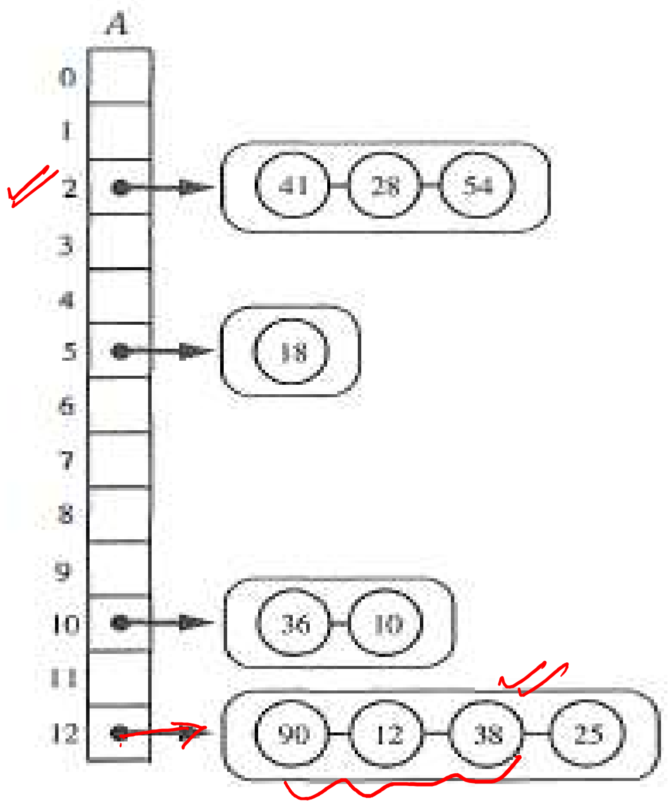
Separate Chaining

- Fundamental dictionary operations
- removeElement (k) :

```
B ←A[h (k)]
if B is empty then
return NO_SUCH _K EY
else
return B. removeElement(k)
```

Separate Chaining

$N=13$



$20 \% 13 = 7$

$38 \% 13 = 12$

Example of a hash table of size 13, storing 10 integer keys, with collisions resolved by the chaining method. The compression map in this case is h (k) = k mod 1 3 .

Separate chaining

- **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

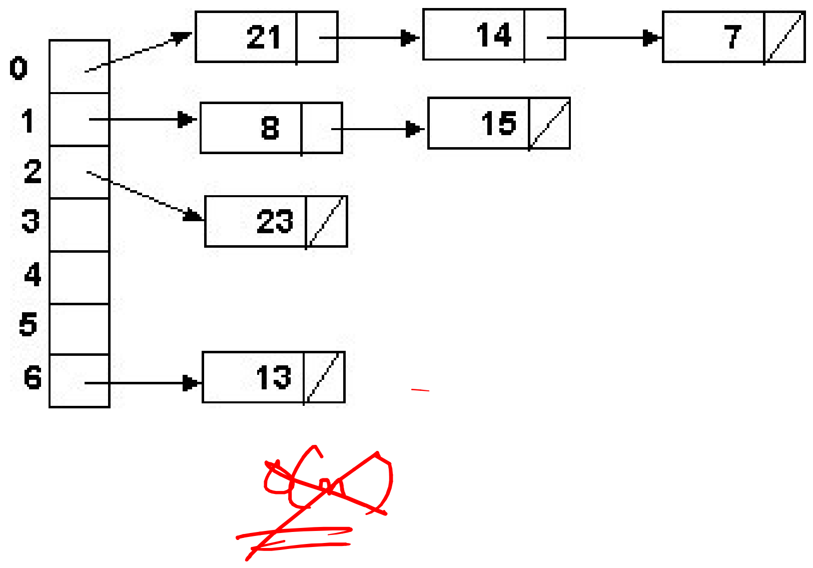
  h(21) = 21 % 7 = 0

  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

- A good hash function will try to minimize collisions as much as possible, which will imply that most of our buckets are either empty or store just a single entry.
- Assume we use a good hash function to index the *n* entries of our map in a bucket array of capacity *N*, we expect each bucket to be of size *n*/*N(average)*
- This value is called the **load factor** of the hash table
- Should be bounded by a small constant, preferably below 1

Separate Chaining

- For a good hash function,the expected running time of operations findElement, insertItem, removeElement in a dictionary implemented using hash table which uses separate chaining to resolve collisions is O(n/N).
- Thus we can expect the standard dictionary operations to run in O(1)expected time provided we know that n is O(N)

Rehashing

- Mostly load factor ,0.75 is common
- Whenever we add elements we need to increase the size of our bucket array and change our compression map to match this new size, in order to keep the load factor below the specified constant.
- Moreover, we must then insert all the existing hash-table elements into the new bucket array using the new compression map. Such a size increase and hash table rebuild is called **rehashing**
- A good choice is to rehash into an array roughly double the size of the original array, choosing the size of the new array to be a prime number

Open Addressing

- **Open addressing**: the colliding item is placed in a different cell of the table
- This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions

Linear Probing

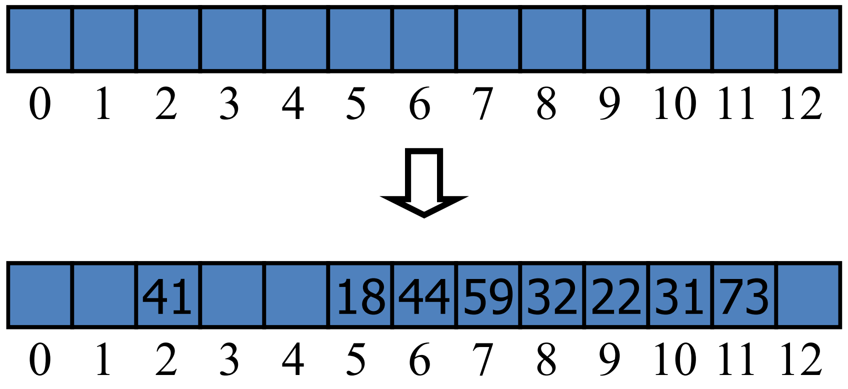
- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a “probe”
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Linear Probing

- In this method, if we try to insert an entry (k, v) into a bucket $A[i]$ that is already occupied, where $i = h(k)$, then we try next at $A[(i+1) \bmod N]$.
- This process will continue until we find an empty bucket that can accept the new entry.

Linear Probing

- **Example**: An insertion into a hash table using linear probing to resolve collisions.
  - h(x) = x mod 13
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Linear Probing

- **Example**
  - We want to add the following (phone, address) entries to an addressBook with size 101:
  - addressBook.add("869-1264", "8-128");
  - addressBook.add("869-8132", "9-101");
  - addressBook.add("869-4294", "8-156");
  - addressBook.add("869-2072", "9-101");

  The hash function is $h(k) = (k \% 10000) \% 101$

All of the above keys (phone numbers) map to index 52. By linear probing, all entries will be put to indices 52 - 55

Search with Linear Probing

- Consider a hash table A that uses linear probing
- **findElement(k)**
  - We start at cell h(k)
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Search with Linear Probing

```
Algorithm $findElement(k)$
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return $NO\_SUCH\_KEY$
        else if $c.key() = k$
            return $c.element()$
        else
            $i \leftarrow (i + 1) \bmod N$
        $p \leftarrow p + 1$
    until $p = N$
    return $NO\_SUCH\_KEY$
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce special object, called AVAILABLE, which replaces deleted elements

removeElement(k)

  - We search for an item with key k
  - If such an item (k, e) is found, we replace it with the special item AVAILABLE and we return element e
  - Else, we return NO_SUCH_KEY

Updates with Linear Probing

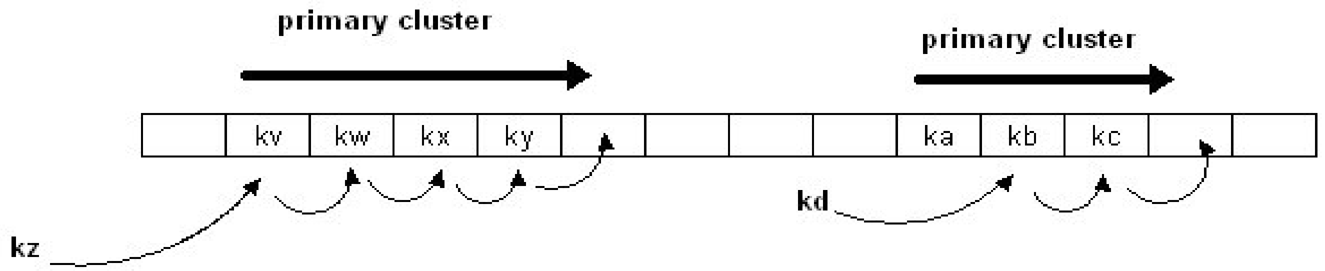
- **insertItem(*k*, *e*)**
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - *N* cells have been unsuccessfully probed
  - We store item (***k*, *e***) in cell ***i***

Problem with Linear Probing

- Primary Clustering

Problem with Linear Probing

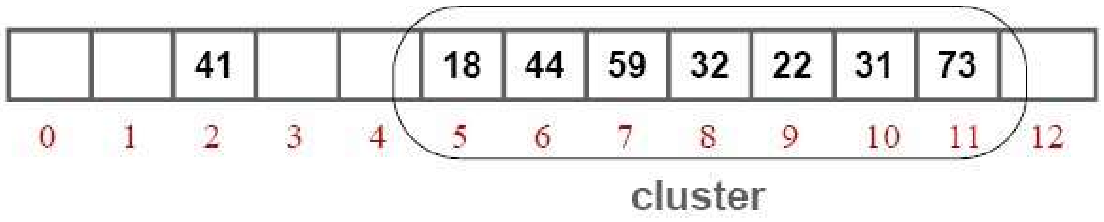
- Linear probing is subject to a primary clustering phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.



Problem with Linear Probing

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



Quadratic probing

- This open addressing strategy involves iteratively trying the buckets

  $\boldsymbol{A[(i + f(j)) \bmod N]}$,

  for $j = 1, 2, \ldots$, where $f(j) = j^2$, until finding an empty bucket
- This strategy may not find an empty slot even when the array is not full.(If N is not chosen as a prime)
- This strategy may not find an empty slot, if the bucket array is at least half full.

Example

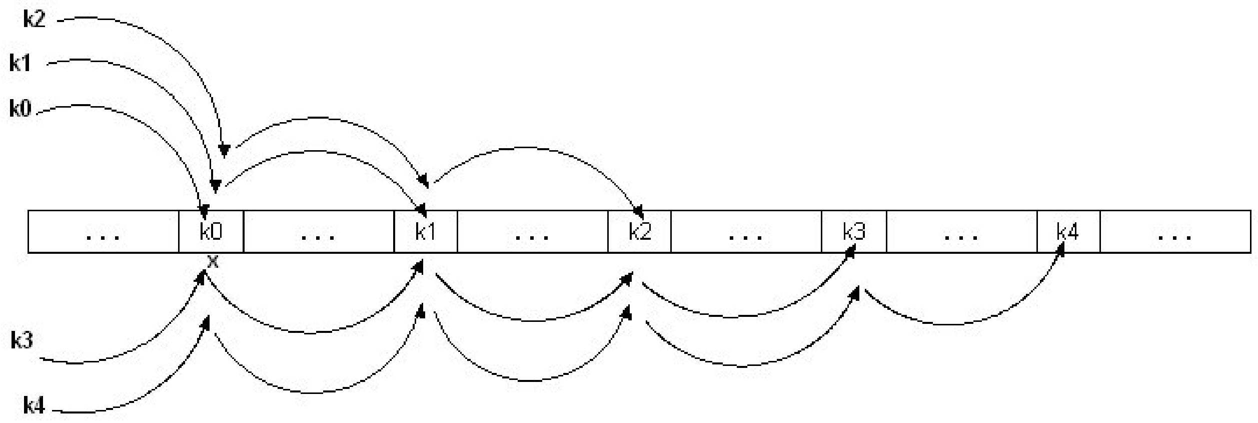
Insert the elements 76,40,48,5 and 55, N=7, h(k)=k mod N

- 76%7=6
- 40%7=5
- 48
  - h(k)=k mod N
  - =48%7=6 --collision
  - $h_1(k)=h(k) + i + i^2$
  - = (6+1+1)mod 7 =1

0	1	2	3	4	5	6
	48			5	40	76

Quadratic probing -Secondary Clusters

- Quadratic probing is better than linear probing because it eliminates primary clustering.
- However, it may result in **secondary clustering**: if **h(k1) = h(k2)** the probing sequences for **k1** and **k2** are exactly the same. This sequence of locations is called a secondary cluster.
- Secondary clustering is less harmful than primary clustering because secondary clusters do not combine to form large clusters.
-



Primary Vs Secondary Clustering

Linear Vs Quadratic probing

- An advantage of linear probing is that it can reach every location in the hash table.
- This property is important since it guarantees the success of the *insertItem* operation when the hash table is not full.
- Quadratic probing can only guarantee a successful *insertItem* operation when the hash table is at most half full.

Double Hashing

- Double hashing uses a secondary hash function h',
- If h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the bucket
  - **$A[(i + f(j)) \bmod N]$ next,**
  - **for $j = 1, 2, 3, \ldots,$ where $f(j) = j*h'(k)$**
- The secondary hash function cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Choose a secondary hash function that will attempt to minimize clustering as much as possible

Double Hashing

- Common choice of compression map for the secondary hash function:

  $$h_2(k) = q - k \bmod q$$

  where

  – $q < N$

  – q is a prime

- The possible values for $h_2(k)$ are

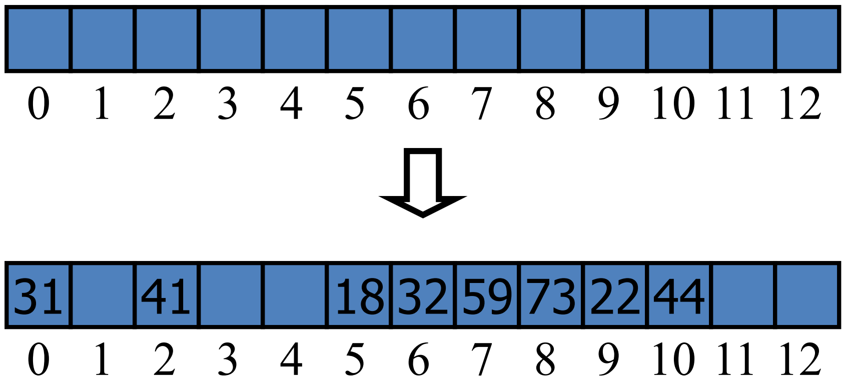
  $$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $h'(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

Example of Double Hashing

k	$h(k)$	$h'(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $\boldsymbol{O}(\boldsymbol{n})$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor = $\boldsymbol{n}/\boldsymbol{N}$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is
  1 / (1 – ***load factor***) [***R2:Section 11.4,Theorem 11.6,11.8***]

Performance of Hashing

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

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

- 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.

Separate chaining Vs 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.

Separate chaining Vs Open Addressing

- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

Separate chaining Vs Open Addressing

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.

Exercises-1

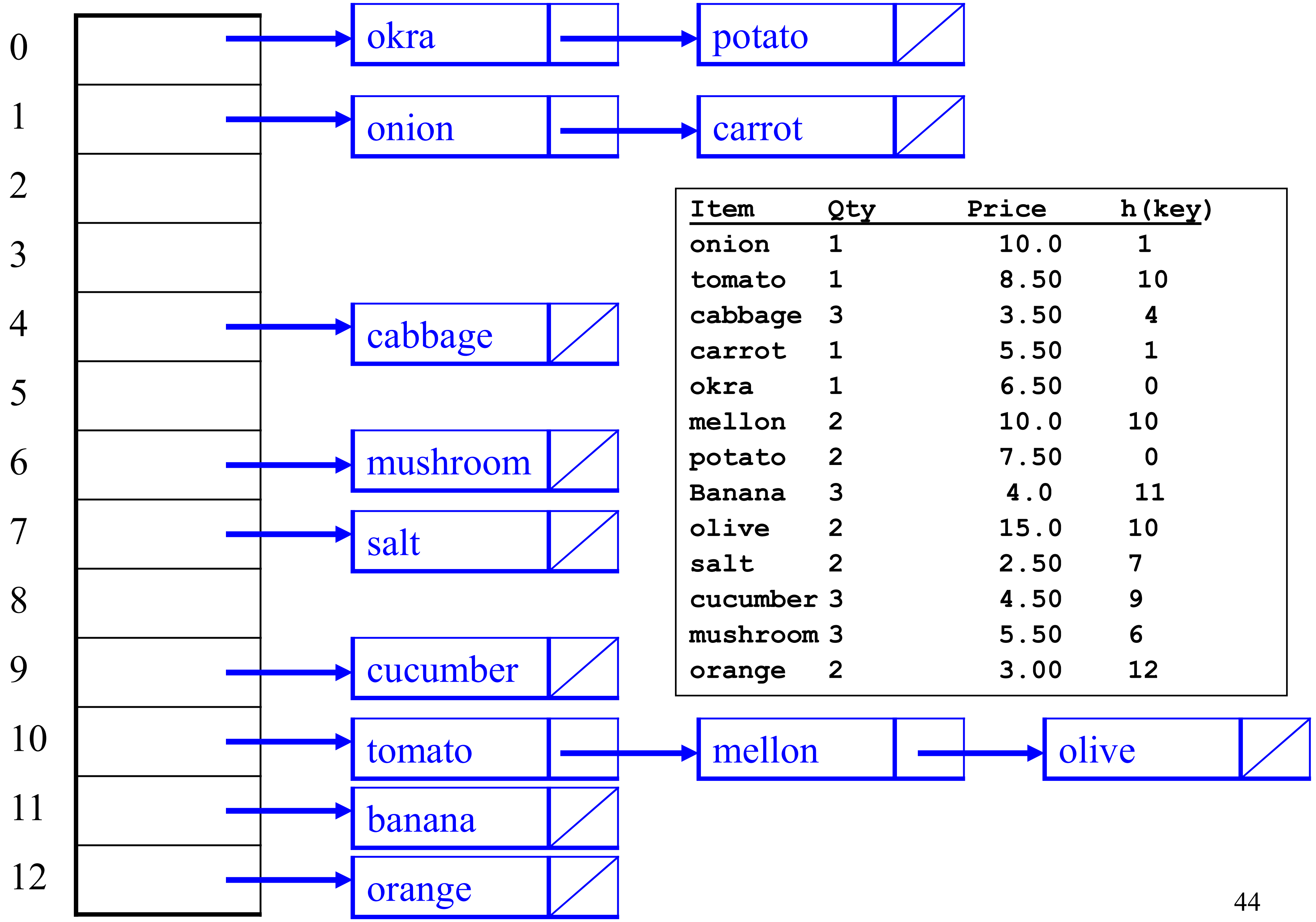
- Use the hash function **hash** to load the following commodity items into a hash table of size **13** using separate chaining:

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

Exercises-I Solution

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



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

Exercises-II

Example:

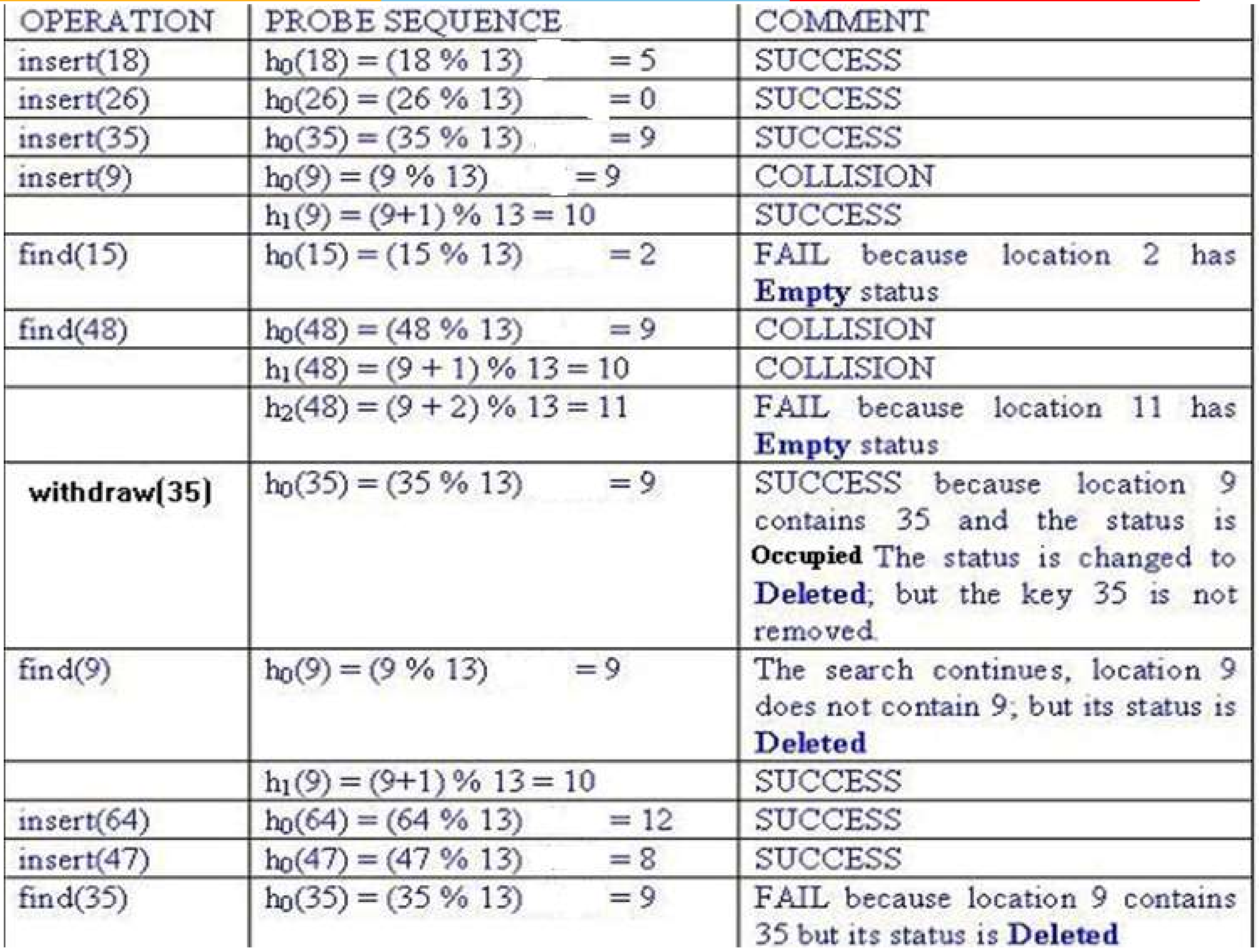
Perform the operations given below, in the given order, on an initially empty hash table of size **13** using linear probing with **c(i) = i** and the hash function: **h(key) = key % 13**:

insert(18), insert(26), insert(35), insert(9), find(15), find(48), delete(35), delete(40), find(9), insert(64), insert(47), find(35)

- The required probe sequences are given by:

$$\mathbf{h_i(key) = (h(key) + i) \% 13 \qquad i = 0, 1, 2, \ldots, 12}$$

Exercises-II



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

Exercises-III

- Suppose you are given an array A[1 : n] stored in read-only memory from which you want to sample ***k*** elements **uniformly** at random *without replacement* (so all of the sampled elements are distinct). Show how to do this, in O(n) expected time and O(k) space using an ADT covered in the contact sessions, and **do not** assume that the elements of ***A*** are *integer-valued*.

Exercises-IV

- Given a **hash table** of size **7** and hash function ***h(x) =x mod** 7*, show the final table after inserting the following elements in the table ***19, 26, 13, 48, 17*** for each of the cases
- i. When ***linear probing*** is used
- ii. When ***double hashing*** is used with a second function **g(x) = 5 – (x mod 5)**

Exercise-V

- Example: Load the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(\text{key}) = (h(\text{key}) + i^2) \% 7 \quad i = 0, 1, 2, 3$$

Exercise-V-Solution

$h_0(23) = (23 \% 7) \% 7 = 2$
$h_0(13) = (13 \% 7) \% 7 = 6$
$h_0(21) = (21 \% 7) \% 7 = 0$
$h_0(14) = (14 \% 7) \% 7 = 0$ collision
$h_1(14) = (0 + 1^2) \% 7 = 1$
$h_0(7) = (7 \% 7) \% 7 = 0$ collision
$h_1(7) = (0 + 1^2) \% 7 = 1$ collision
$h_2(7) = (0 + 2^2) \% 7 = 4$
$h_0(8) = (8 \% 7)\%7 = 1$ collision
$h_1(8) = (1 + 1^2) \% 7 = 2$ collision
$h_2(8) = (1 + 2^2) \% 7 = 5$
$h_0(15) = (15 \% 7)\%7 = 1$ collision
$h_1(15) = (1 + 1^2) \% 7 = 2$ collision
$h_2(15) = (1 + 2^2) \% 7 = 5$ collision
$h_3(15) = (1 + 3^2)\%7 = 3$

0	21
1	14
2	23
3	15
4	7
5	8
6	13

Exercise-VI

Load the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in this order,in an empty hash table of size **13**

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 1 + key % 12**

(b) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 7 - key % 7**

Show all computations.

Exercise-VI-Solution

$h_0(18) = (18\%13)\%13 = 5$

$h_0(26) = (26\%13)\%13 = 0$

$h_0(35) = (35\%13)\%13 = 9$

$h_0(9) = (9\%13)\%13 = 9$ collision

$h_p(9) = 1 + 9\%12 = 10$

$h_1(9) = (9 + 1*10)\%13 = 6$

$h_0(64) = (64\%13)\%13 = 12$

$h_0(47) = (47\%13)\%13 = 8$

$h_0(96) = (96\%13)\%13 = 5$ collision

$h_p(96) = 1 + 96\%12 = 1$

$h_1(96) = (5 + 1*1)\%13 = 6$ collision

$h_2(96) = (5 + 2*1)\%13 = 7$

$h_0(36) = (36\%13)\%13 = 10$

$h_0(70) = (70\%13)\%13 = 5$ collision

$h_p(70) = 1 + 70\%12 = 11$

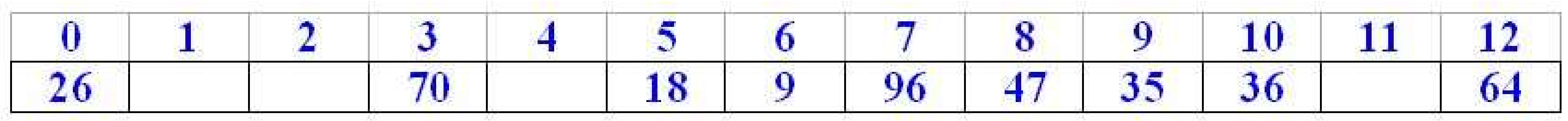
$h_1(70) = (5 + 1*11)\%13 = 3$

hi(key) = [h(key) + i*hp(key)]% 13

h(key) = key % 13

hp(key) = 1 + key % 12

Exercise-VI-Solution



0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64