King Fahd University of Petroleum & Minerals

College of Computer Science & Engineering

Information & Computer Science Department

Unit 12

Hashing

Information and Computer Science

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
    - Chapter 10
        - Section 10.1: Hash Functions
        - Section 10.2: Collision Resolution (Except 10.2.3: Bucket Addressing)
        - Section 10.3: Deletion

Introduction to Hashing & Hashing Techniques

- Review of Searching Techniques
- Introduction to Hashing
- Hash Tables
  - Types of Hashing
- Hash Functions
  - Applications of Hash Tables
  - Problems for which Hash Tables are not suitable
- Collision Resolution Techniques
- Separate Chaining
- Introduction to Collision Resolution using Open Addressing
  - **Linear Probing**
  - **Quadratic Probing**
  - **Double Hashing**
  - **Rehashing**
- Algorithms for insertion, searching, and deletion in Open Addressing
- Separate Chaining versus Open-addressing

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 of data size would be better.

- Consider the following Java 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.
- Is there some way that we could get `O(1)` access without wasting a lot of space?
- The answer is hashing.

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 -> {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*.

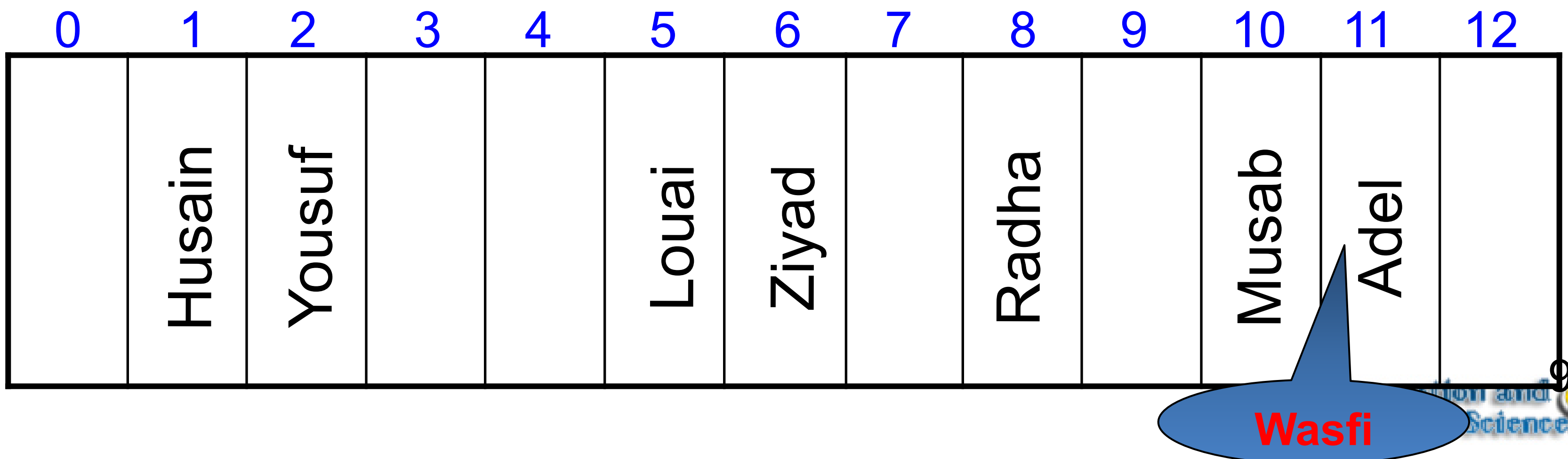
Example 1: Illustrating Hashing

- 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
Al-Khatib, Wasfi Ghassan	1.74	863523

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
Al-Khatib, Wasfi Ghassan	863523	11



Hash Tables

- There are two types of Hash Tables: **Open-addressed Hash Tables** and **Separate-Chained 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.***

Types of Hashing

- There are two types of hashing :

  1. ***Static hashing***: In static hashing, the hash function maps search-key values to a fixed set of *locations*.

  2. ***Dynamic hashing***: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

- 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.
  - What do you think will happen when the load factor becomes high?
  - With open addressing, the load factor cannot exceed 1. With chaining, the load factor often exceeds 1.

Desired Properties of 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.

Common Hashing Functions

1. **Division Remainder (using the table size as the divisor)**

- Computes hash value from key using the % operator.
- Table size that is a power of 2 like 32 and 1024 should be avoided, for it leads to more collisions.
- Also, powers of 10 are not good for table sizes when the keys rely on decimal integers.
- Prime numbers not close to powers of 2 are better table size values.

Common Hashing Functions (cont'd)

2. Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.

- To map the key 25936715 to a range between 0 and 9999, we can:
  - split the number into two as 2593 and 6715 and
  - add these two to obtain 9308 as the hash value.

- Very useful if we have keys that are very large.

- Fast and simple especially with bit patterns.

- A great advantage is ability to transform non-integer keys into integer values.

Common Hashing Functions (cont'd)

3. 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 $3121^2 = 9740641$ 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.

Common Hashing Functions (cont'd)

4. Truncation or Digit/Character Extraction

- Works based on the distribution of digits or characters in the key.
- 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.
- Very fast but digits/characters distribution in keys may not be very even.

Common Hashing Functions (cont'd)

5. 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.

Common Hashing Functions (cont'd)

6. Use of a 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 implementation 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.

Exercises

1. What in your opinion is the single most important motivation for the development of hashing schemes while there already are other techniques that can be used to realize the same functionality provided by hashing methods?
2. How many storage cells will be wasted in an array implementation with O(1) access for records of 10,000 students each with a 7-digit ID number?
3. Must a hash table be implemented using an array? Will an alternative data structure achieve the same efficiency? If yes, why? If no, what condition must the data structure satisfy to ensure the same efficiency as provided by arrays?
4. Which of the techniques for creating hash functions is most general? Why?
5. Why do prime numbers generally make a good selection for hash table sizes?

Collision Resolution Techniques

- There are two broad ways of collision resolution:

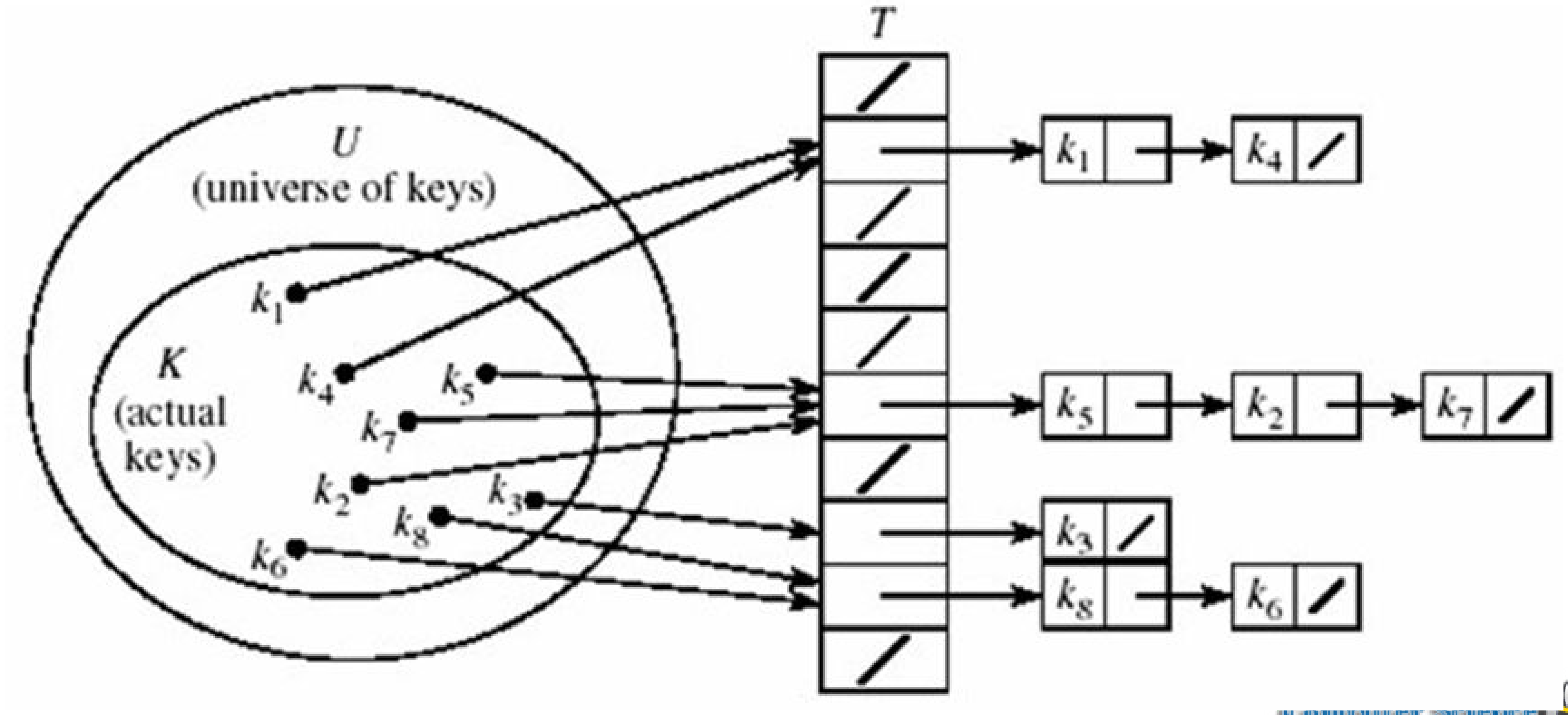
1. Separate Chaining:: An array of linked list implementation.

2. Open Addressing: Array-based implementation.

   (i) Linear probing (linear search)
   (ii) Quadratic probing (nonlinear search)
   (iii) Double hashing (uses two hash functions)

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 (cont'd)

- 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

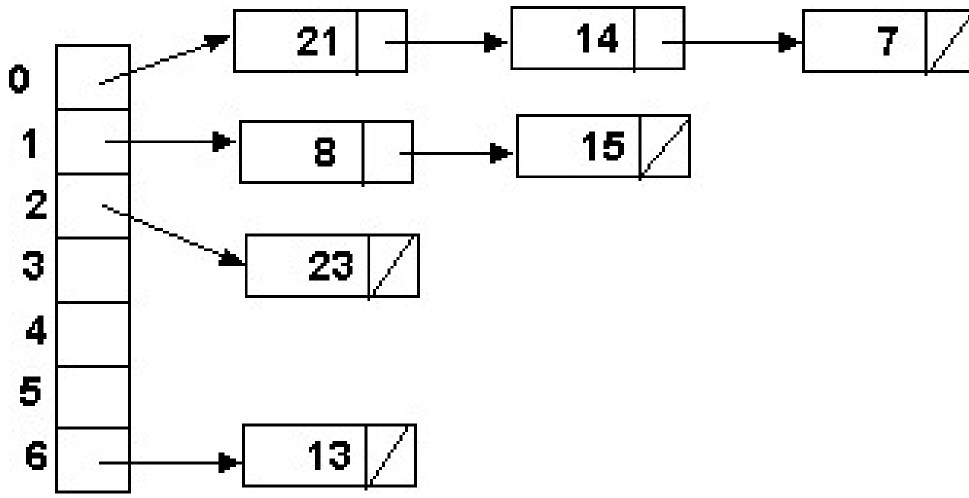
h(21) = 21 % 7 = 0

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:

```
public static int hash(String key, int tableSize){
   int hashValue = 0;
   for (int i = 0; i < key.length(); i++){
hashValue += key.charAt(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 (cont'd)

- 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

- Solution:

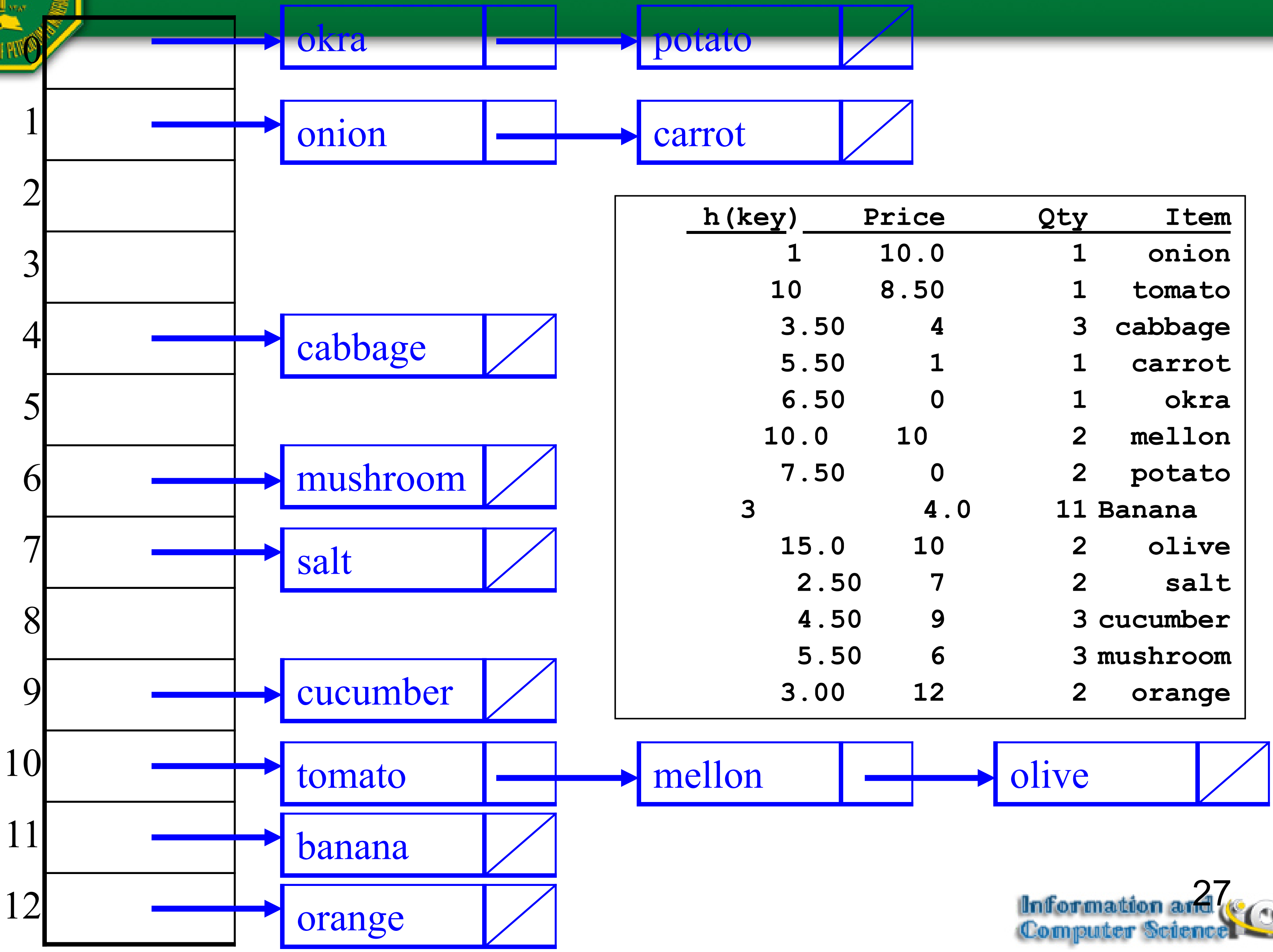
character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

$$\text{hash(onion)} = (111 + 110 + 105 + 111 + 110) \% 13 = 547 \% 13 = 1$$

$$\text{hash(salt)} = (115 + 97 + 108 + 116) \% 13 = 436 \% 13 = 7$$

$$\text{hash(orange)} = (111 + 114 + 97 + 110 + 103 + 101)\%13 = 636 \%13 = 12$$

Separate Chaining with String Keys (cont'd)



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

Separate Chaining with String Keys (cont'd)

- 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
- hash = $(c_0 + c_{n-1} + s.length())$ % tableSize
- hash = $$[\sum_{k=0}^{s.length()-1} 26 * k + s.charAt(k) - '\,'] \% tableSize$$

Introduction to Open Addressing

- All items are stored in the hash table itself.
- In addition to the cell data (if any), each cell keeps one of the three states: EMPTY, OCCUPIED, DELETED.
- 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.
- **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.
- The most common probe sequences are of the form:

$$h_i(key) = [h(key) + c(i)] \% \ n, \quad \text{for } i = 0, 1, \ldots, n-1.$$

where **h** is a hash function and **n** is the size of the hash table

- The function **c(i)** is required to have the following two properties:

Property 1: c(0) = 0

Property 2: The set of values **{c(0) % n, c(1) % n, c(2) % n, . . . , c(n-1) % n}** must be a permutation of **{0, 1, 2,. . ., n – 1}**, that is, it must contain every integer between **0** and **n - 1** inclusive.

Introduction to Open Addressing (cont'd)

- The function **c(i)** is used to resolve collisions.
- To insert item r, we examine array location **$h_0(r) = h(r)$**. If there is a collision, array locations **$h_1(r), h_2(r), \ldots, h_{n-1}(r)$** are examined until an empty slot is found.
- Similarly, to find item **r**, we examine the same sequence of locations in the same order.
- **Note**: For a given hash function **h(key)**, the only difference in the open addressing collision resolution techniques (linear probing, quadratic probing and double hashing) is in the definition of the function **c(i)**.
- Common definitions of **c(i)** are:

Collision resolution technique	**c(i)**
Linear probing	**i**
Quadratic probing	**$\pm i^2$**
Double hashing	**$i * h_p(key)$**

where **h_p(key)** is another hash function.

Introduction to Open Addressing (cont'd)

- **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:

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

Open Addressing: Linear Probing

- **c(i)** is a linear function in **i** of the form **c(i) = a*i**.
- Usually **c(i)** is chosen as:

  c(i) = i **for i = 0, 1, . . . , tableSize – 1**

- The probe sequences are then given by:

  h_i(key) = [h(key) + i] % tableSize **for i = 0, 1, . . . , tableSize – 1**

- For **c(i) = a*i** to satisfy Property 2, **a** and **n** must be relatively prime.

Linear Probing (cont'd)

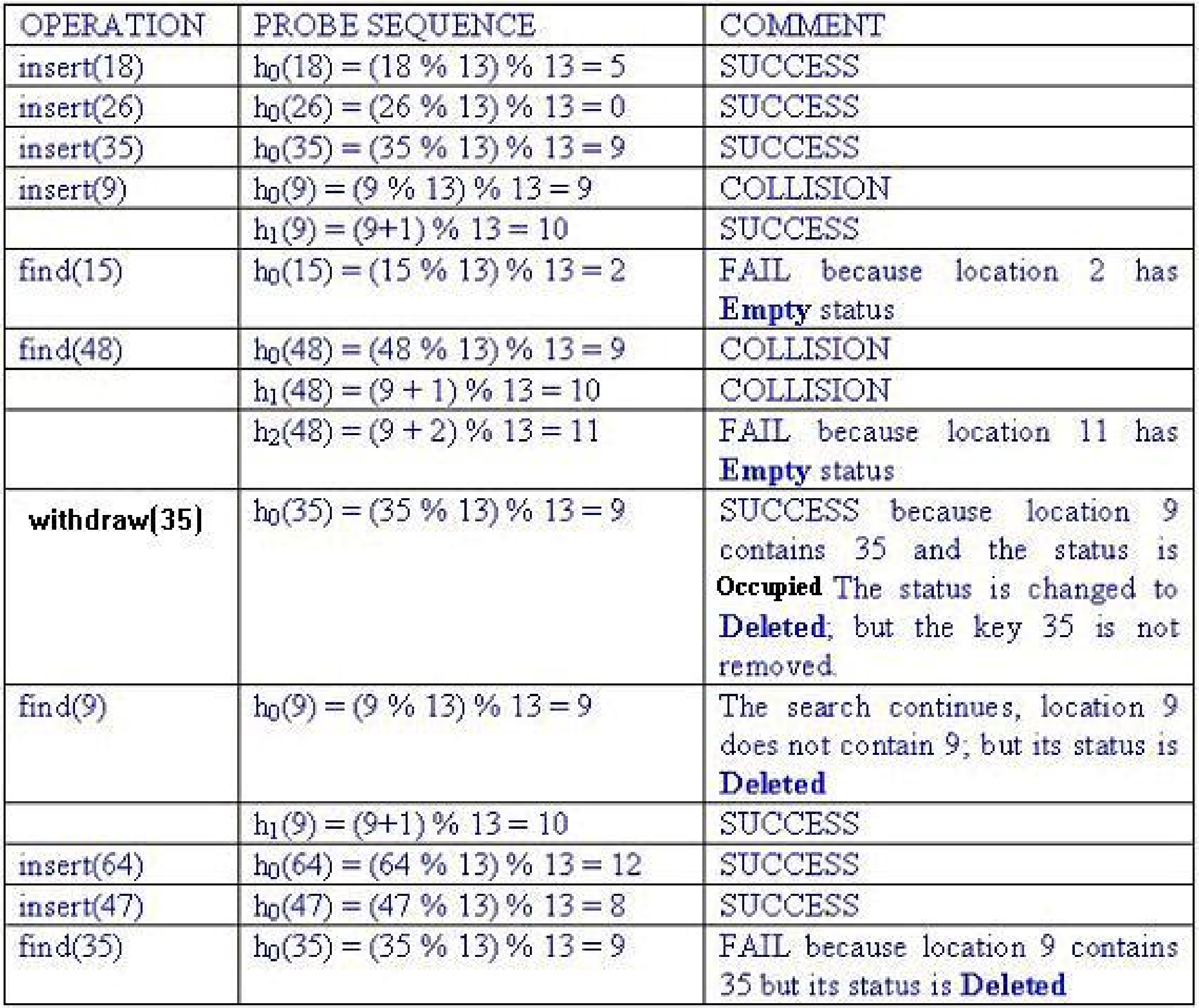
Example: Perform the operations given below, in the given order, on an initially empty hash table of size **13** using linear probing with **c(i) = i** and the hash function: **h(key) = key % 13**:

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:

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

Linear Probing (cont'd)

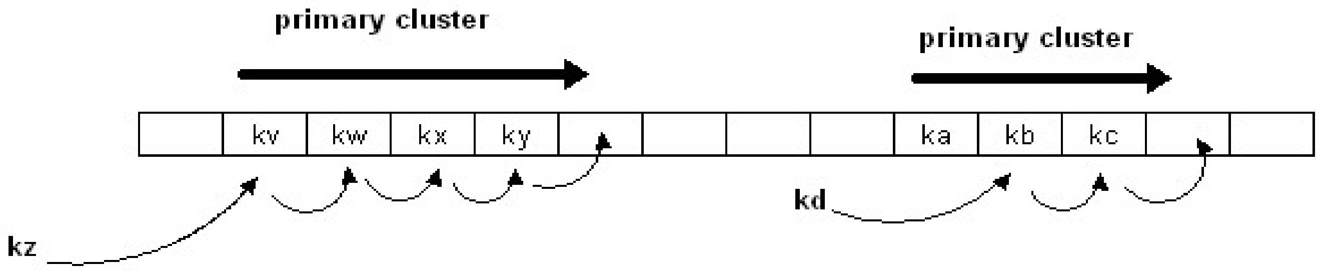


OPERATION	PROBE SEQUENCE	COMMENT
insert(18)	$h_0(18) = (18 \% 13) \% 13 = 5$	SUCCESS
insert(26)	$h_0(26) = (26 \% 13) \% 13 = 0$	SUCCESS
insert(35)	$h_0(35) = (35 \% 13) \% 13 = 9$	SUCCESS
insert(9)	$h_0(9) = (9 \% 13) \% 13 = 9$	COLLISION
	$h_1(9) = (9+1) \% 13 = 10$	SUCCESS
find(15)	$h_0(15) = (15 \% 13) \% 13 = 2$	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
withdraw(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

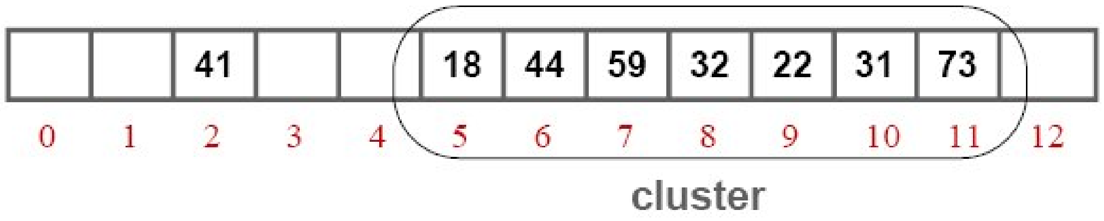
Disadvantage of Linear Probing: Primary Clustering

- 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+1a



Open Addressing: Quadratic Probing

- Quadratic probing eliminates primary clusters.
- **c(i)** is a quadratic function in **i** of the form **$c(i) = a*i^2 + b*i$**. Usually **c(i)** is chosen as:

  $c(i) = i^2$ **for i = 0, 1, . . . , tableSize – 1**

  or

  $c(i) = \pm i^2$ **for i = 0, 1, . . . , (tableSize – 1) / 2**

- The probe sequences are then given by:

  $h_i(key) = [h(key) + i^2]$ % tableSize **for i = 0, 1, . . . , tableSize – 1**

  or

  $h_i(key) = [h(key) \pm i^2]$ % tableSize **for i = 0, 1, . . . , (tableSize – 1) / 2**

- Note for Quadratic Probing:
  - Hashtable size should not be an even number; otherwise Property 2 will not be satisfied.
  - Ideally, table size should be a prime of the form 4j+3, where j is an integer. This choice of table size guarantees Property 2.

Quadratic Probing (cont'd)

- 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 with **$c(i) = \pm i^2$** and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Quadratic Probing (cont'd)

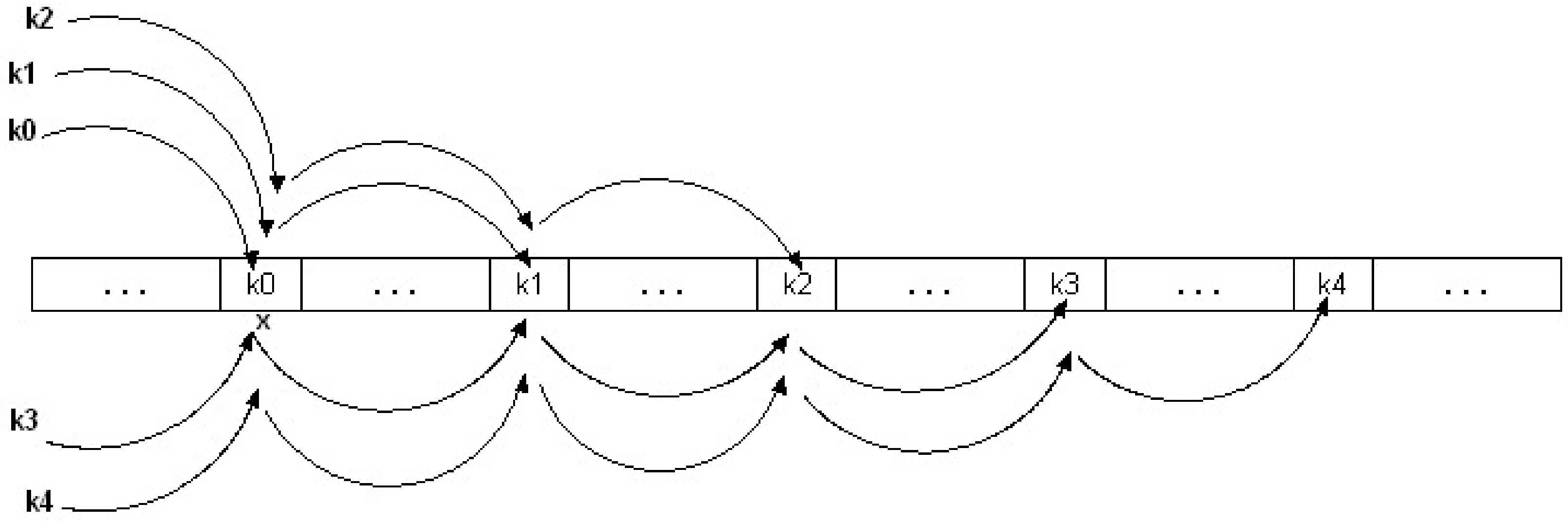
$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(23) = (23 \% 7) \% 7 = 2$
$h_0(13) = (13 \% 7) \% 7 = 6$
$h_0(21) = (21 \% 7) \% 7 = 0$
$h_0(14) = (14 \% 7) \% 7 = 0$ 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
$h_{-1}(7) = (0 - 1^2) \% 7 = -1$
NORMALIZE: $(-1 + 7) \% 7 = 6$ 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
$h_{-1}(8) = (1 - 1^2) \% 7 = 0$ 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
$h_{-1}(15) = (1 - 1^2) \% 7 = 0$ collision
$h_2(15) = (1 + 2^2) \% 7 = 5$ collision
$h_{-2}(15) = (1 - 2^2) \% 7 = -3$
NORMALIZE: $(-3 + 7) \% 7 = 4$ collision
$h_3(15) = (1 + 3^2)\%7 = 3$

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

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.
- **Example of Secondary Clustering:** Suppose keys **k0, k1, k2, k3, and k4** are inserted in the given order in an originally empty hash table using **quadratic probing** with **c(i) = i^2**. Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size:



Double Hashing

- To eliminate secondary clustering, synonyms must have different probe sequences.

- Double hashing achieves this by having two hash functions that both depend on the hash key.

- $c(i) = i * h_p(key)$ **for i = 0, 1, . . . , tableSize – 1**

  where h_p (or h_2) is another hash function.

- The probing sequence is:

  $h_i(key) = [h(key) + i*h_p(key)]\%$ tableSize **for i = 0, 1, . . . , tableSize – 1**

- The function $c(i) = i*h_p(r)$ satisfies Property 2 provided $h_p(r)$ and tableSize are relatively prime.

- To guarantee Property 2, tableSize must be a prime number.

- Common definitions for $\mathbf{h_p}$ are :
  - $h_p(key) = 1 + key \% (tableSize - 1)$
  - $h_p(key) = q - (key \% q)$ where **q** is a prime less than **tableSize**
  - $h_p(key) = q*(key \% q)$ where **q** is a prime less than **tableSize**

Double Hashing (cont'd)

Performance of Double hashing:

- Much better than linear or quadratic probing because it eliminates both primary and secondary clustering.
- BUT requires a computation of a second hash function $\mathbf{h_p}$.

Example: 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: $\mathbf{h_p(key) = 1 + key \% 12}$

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

Show all computations.

Double Hashing (cont'd)

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

$h_i(key) = [h(key) + i*h_p(key)]\% \ 13$

$h(key) = key \ \% \ 13$

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

$h_0(18) = (18\%13)\%13 = 5$
$h_0(26) = (26\%13)\%13 = 0$
$h_0(35) = (35\%13)\%13 = 9$
$h_0(9) = (9\%13)\%13 = 9$ 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
 $h_p(70) = 1 + 70\%12 = 11$
 $h_1(70) = (5 + 1*11)\%13 = 3$

Double Hashing (cont'd)

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

$h_0(18) = (18\%13)\%13 = 5$
$h_0(26) = (26\%13)\%13 = 0$
$h_0(35) = (35\%13)\%13 = 9$
$h_0(9) = (9\%13)\%13 = 9$ collision
    $h_p(9) = 7 - 9\%7 = 5$
    $h_1(9) = (9 + 1*5)\%13 = 1$
$h_0(64) = (64\%13)\%13 = 12$
$h_0(47) = (47\%13)\%13 = 8$
$h_0(96) = (96\%13)\%13 = 5$ collision
    $h_p(96) = 7 - 96\%7 = 2$
    $h_1(96) = (5 + 1*2)\%13 = 7$
$h_0(36) = (36\%13)\%13 = 10$
$h_0(70) = (70\%13)\%13 = 5$ collision
    $h_p(70) = 7 - 70\%7 = 7$
    $h_1(70) = (5 + 1*7)\%13 = 12$ collision
    $h_2(70) = (5 + 2*7)\%13 = 6$

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

$h(key) = key \% 13$

$h_p(key) = 7 - key \% 7$

Rehashing

- As noted before, with open addressing, if the hash tables become too full, performance can suffer a lot.
- So, what can we do?
- We can double the hash table size, modify the hash function, and re-insert the data.
  - More specifically, the new size of the table will be the first prime that is more than twice as large as the old table size.

Implementation of Open Addressing

```
public class OpenScatterTable extends AbstractHashTable {
   protected Entry array[];
   protected static final int EMPTY = 0;
   protected static final int OCCUPIED = 1;
   protected static final int DELETED = 2;

   protected static final class Entry {
      public int state = EMPTY;
      public Comparable object;
      // …
   }

   public OpenScatterTable(int size) {
      array = new Entry[size];
      for(int i = 0; i < size; i++)
         array[i] = new Entry();
   }
   // …
}
```

Implementation of Open Addressing (Con't.)

```
/* finds the index of the first unoccupied slot
   in the probe sequence of obj */
protected int findIndexUnoccupied(Comparable obj){
   int hashValue = h(obj);
   int tableSize = getLength();
   int indexDeleted = -1;
   for(int i = 0; i < tableSize; i++){
      int index = (hashValue + c(i)) % tableSize;
      if(array[index].state == OCCUPIED
               && obj.equals(array[index].object))
         throw new IllegalArgumentException(
                        "Error: Duplicate key");

      else if(array[index].state == EMPTY ||
        (array[index].state == DELETED &&
            obj.equals(array[index].object)))
        return indexDeleted ==-1?index:indexDeleted;
      else if(array[index].state == DELETED &&
        indexDeleted == -1)
          indexDeleted = index;
   }
   if(indexDeleted != -1) return indexDeleted;

   throw new IllegalArgumentException(
               "Error: Hash table is full");
}
```

Implementation of Open Addressing (Con't.)

```
protected int findObjectIndex(Comparable obj){
   int hashValue = h(obj);
   int tableSize = getLength();

   for(int i = 0; i < tableSize; i++){
      int index = (hashValue + c(i)) % tableSize;
      if(array[index].state == EMPTY
            || (array[index].state == DELETED
            && obj.equals(array[index].object)))
         return -1;
      else if(array[index].state == OCCUPIED
               && obj.equals(array[index].object))
         return index;
   }
   return -1;
}

public Comparable find(Comparable obj){
   int index = findObjectIndex(obj);
   if(index >= 0)return array[index].object;
   else return null;
}
```

Implementation of Open Addressing (Con't.)

```
public void insert(Comparable obj){
    if(count == getLength()) throw new ContainerFullException();
    else {
        int index = findIndexUnoccupied(obj);
        // throws exception if an UNOCCUPIED slot is not found
        array[index].state = OCCUPIED;
        array[index].object = obj;
        count++;
    }
}

public void withdraw(Comparable obj){
    if(count == 0) throw new ContainerEmptyException();
    int index = findObjectIndex(obj);
    if(index < 0)
        throw new IllegalArgumentException("Object not found");
    else {
        array[index].state = DELETED;
        // lazy deletion: DO NOT SET THE LOCATION TO null
        count--;
    }
}
```

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.

Exercises

1. Given that,

   `c(i) = a*i,`

   for `c(i)` in linear probing, we discussed that this equation satisfies Property 2 only when `a` and `n` are relatively prime. Explain what the requirement of being relatively prime means in simple plain language.

2. Consider the general probe sequence,

   $h_i\ (r) = (h(r) + c(i))\%\ n.$

   Are we sure that if `c(i)` satisfies Property 2, then $h_i(r)$ will cover all `n` hash table locations, `0,1,...,n-1`? Explain.

3. Suppose you are given k records to be loaded into a hash table of size `n`, with `k < n` using linear probing. Does the order in which these records are loaded matter for retrieval and insertion? Explain.

4. A prime number is always the best choice of a hash table size. Is this statement true or false? Justify your answer either way.

Exercises

5. If a hash table is 25% full what is its load factor?

6. Given that,

$$c(i) = i^2,$$

for `c(i)` in quadratic probing, we discussed that this equation does not satisfy Property 2, in general. What cells are missed by this probing formula for a hash table of size 17? Characterize using a formula, if possible, the cells that are not examined by using this function for a hash table of size n.

7. It was mentioned in this session that secondary clusters are less harmful than primary clusters because the former cannot combine to form larger secondary clusters. Use an appropriate hash table of records to exemplify this situation.