CISC 235: Topic 5

Dictionaries and Hash Tables

Outline

- Dictionaries
  - Dictionaries as Partial Functions
- Unordered Dictionaries
  - Implemented as Hash Tables
- Collision Resolution Schemes
  - Separate Chaining
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- Design of Hash Functions

Caller ID Problem Scenario

Consider a large phone company that wants to provide Caller ID to its customers:

- Given a phone number, return the caller's name

Key	Element
phone number	caller's name

Assumption: Phone numbers are unique and are in the range $0..10^7 - 1$. However, not all those numbers are current phone numbers.

How shall we store and look up our (phone number, name) pairs?

Caller ID Solutions

Let u = number of possible key values: 10^7

Let k = number of phone/name pairs

1. Use a linked list

   Time Analysis (search, insert, delete):

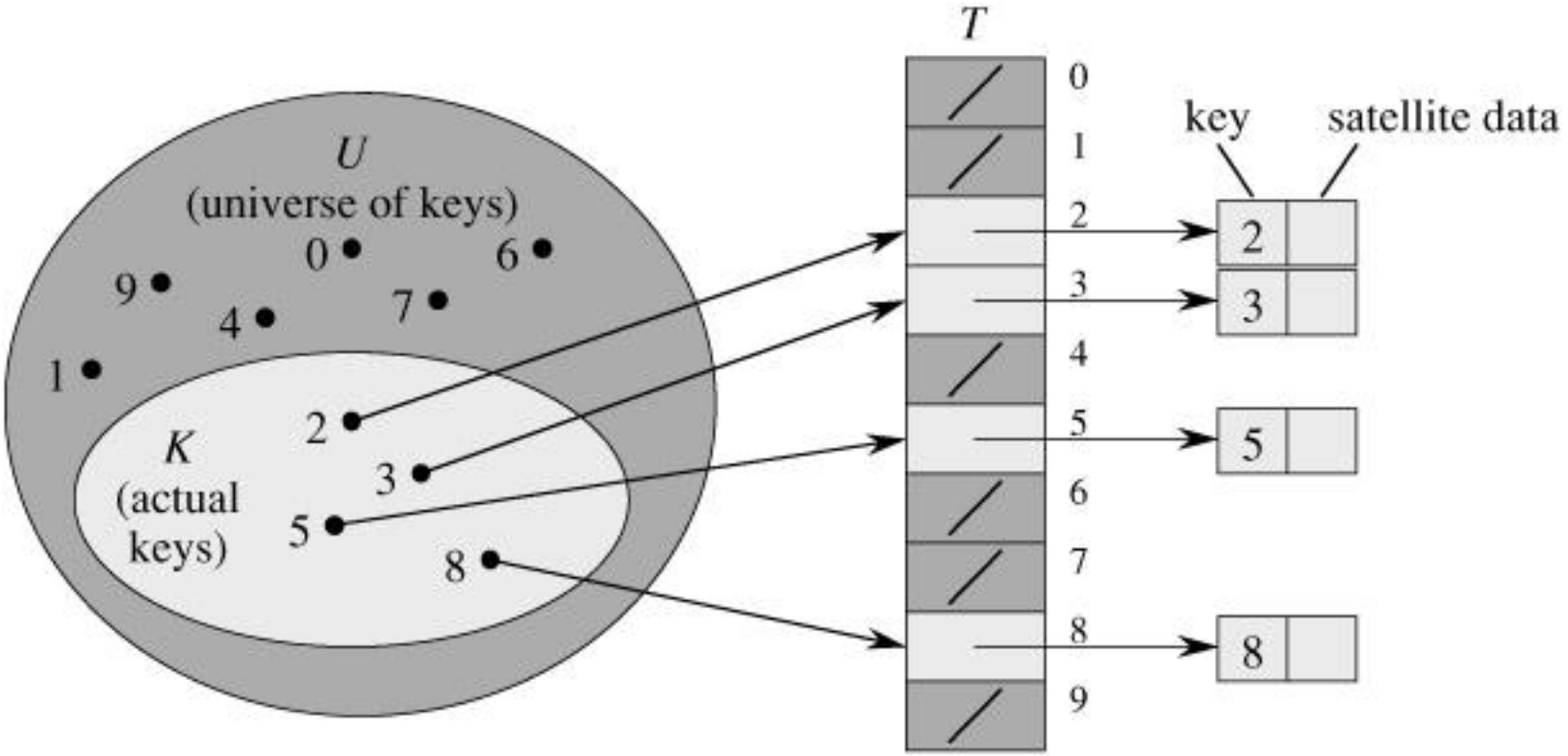
   Space Analysis:

2. Use a balanced binary search tree

   Time Analysis (search, insert, delete):

   Space Analysis:

Direct-Address Table



Direct-address Tables

Direct-Address-Search(T, k)

    return T[k]

Direct-Address-Insert(T, x)

    T[key[x]] ← x

Direct-Address-Delete(T, x)

    T[key[x]] ← NIL

We could use a direct-address table to implement caller-id, with the phone numbers as keys.

Time Analysis:

Space Analysis:

Dictionaries

A dictionary consists of key/element pairs in which the key is used to look up the element.

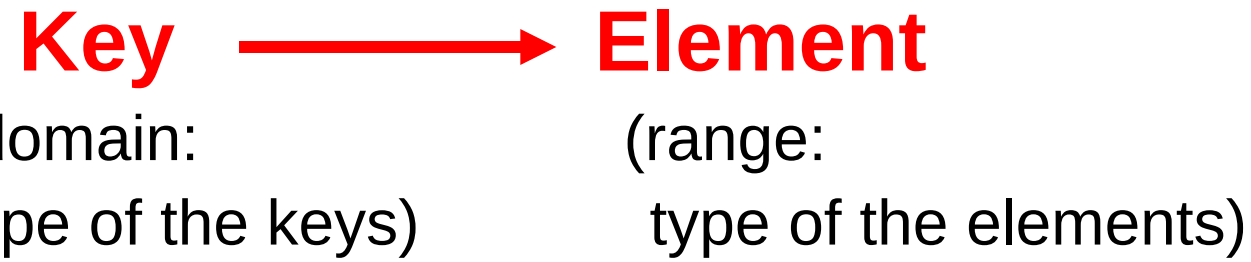
Ordered Dictionary: Elements stored in sorted order by key

Unordered Dictionary: Elements not stored in sorted order

Example	Key	Element
English Dictionary	Word	Definition
Student Records	Student Number	Rest of record: Name, …
Symbol Table in Compiler	Variable Name	Variable's Address in Memory
Lottery Tickets	Ticket Number	Name & Phone Number

Dictionary as a Function

Given a key, return an element



A dictionary is a partial function. Why?

Unordered Dictionary
Best Implementation: Hash Table

Space: O(n)

Time: O(1) average-case

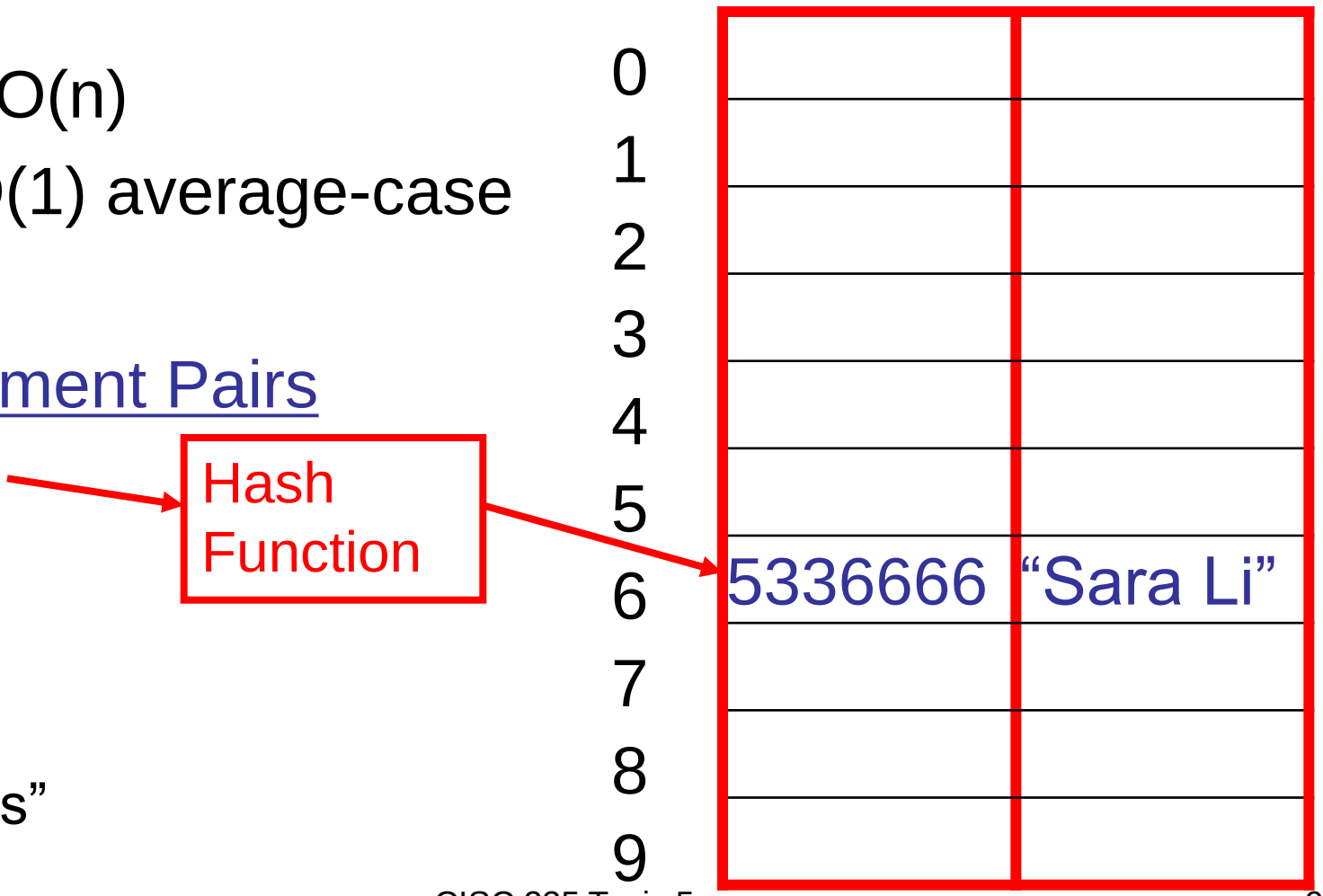
Key/Element Pairs

5336666

“Sara Li”

5661111

“Lea Ross”

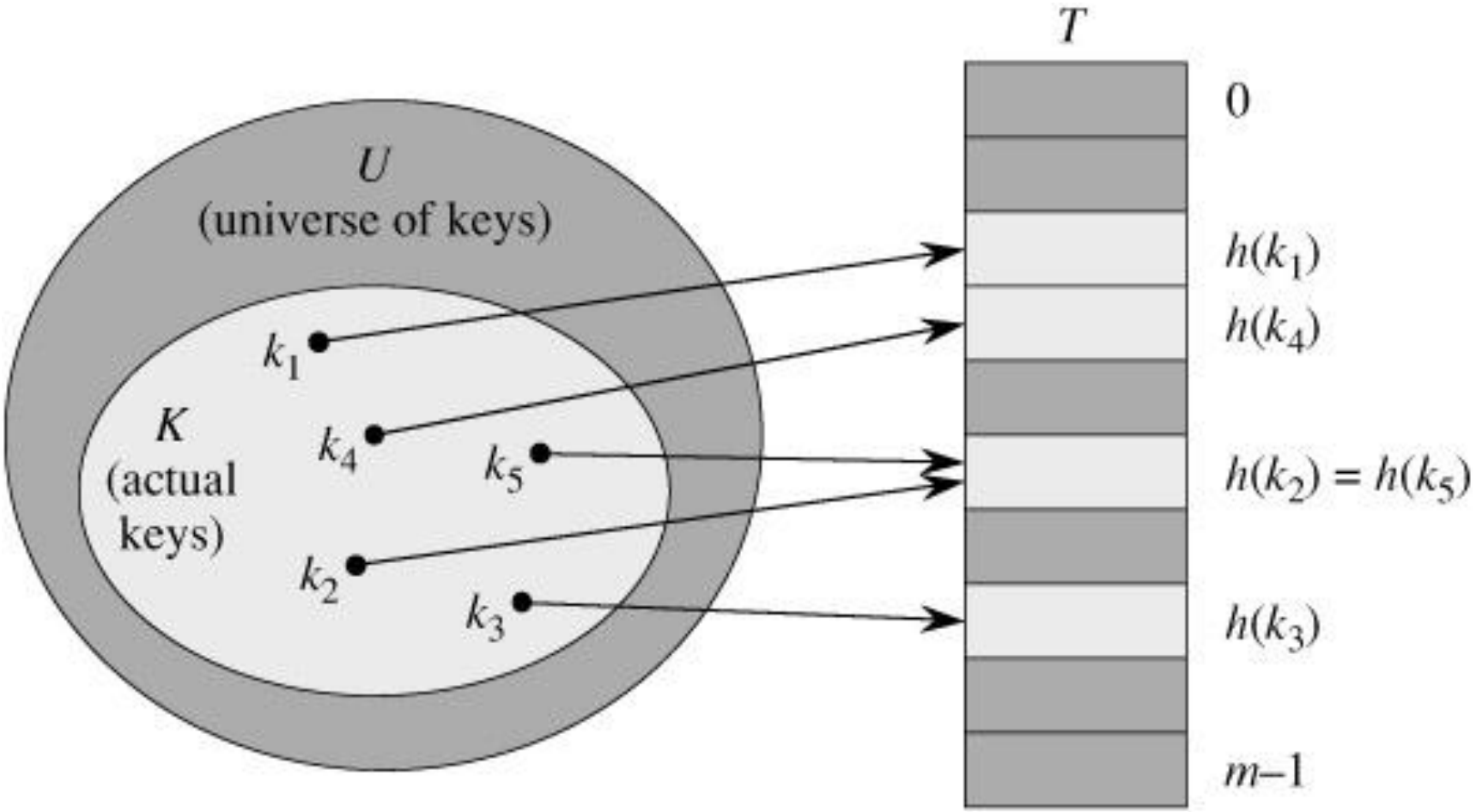


Example Hash Function

```
h( k )
    return k mod m
```

where k is the key and m is the size of the table

Hash Table with Collision



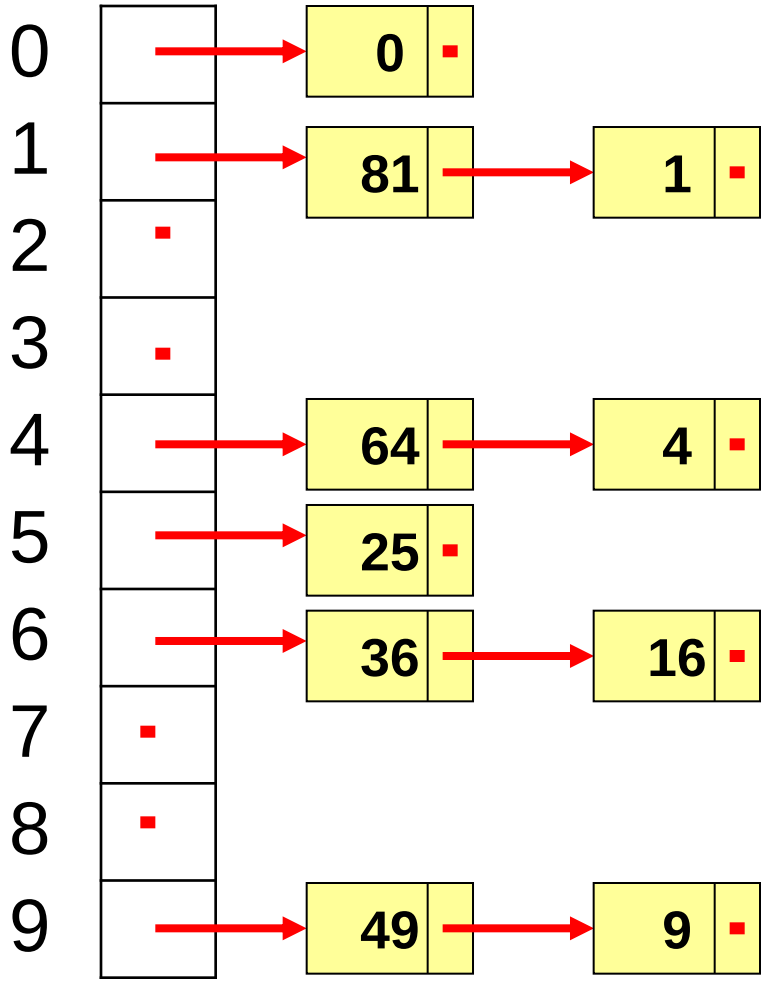
Collision Resolution Schemes: Chaining

The hash table is an array of linked lists

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m



Chaining Algorithms

Chained-Hash-Insert(T, x)

    insert x at the head of list T[h(key[x])]

Chained-Hash-Search(T, k)

    search for an element with key k
    in list T[h(k)]

Chained-Hash-Delete(T, x)

    delete x from the list T[h(key[x])]

Worst-case Analysis of Chaining

Let n = number of elements in the hash table

Let m = hash table size

Let $\lambda = n / m$ (the load factor, i.e, the average number of elements stored in a chain)

What is the worst-case?

Unsuccessful Search:

Successful Search:

Average-Case Analysis of Chaining for an Unsuccessful Search

Let n = number of elements in hash table

Let m = hash table size

Let $\lambda = n / m$ (the load factor, i.e, the average number of elements stored in a chain)

Average-Case Analysis of Chaining for a Successful Search

Let n = number of elements in hash table

Let m = hash table size

Let $\lambda = n / m$ (the load factor, i.e, the average number of elements stored in a chain)

Questions to Ask When Analyzing Resolution Schemes

1. Are we guaranteed to find an empty cell if there is one?
2. Are we guaranteed we won't be checking the same cell twice during one insertion?
3. What should the load factor be to obtain O(1) average-case insert, search, and delete?

Answers for Chaining:

1.
2.
3.

Collision Resolution Strategies: Open Addressing

All elements stored in the hash table itself (the array). If a collision occurs, try alternate cells until empty cell is found.

Three Resolution Strategies:

- Linear Probing
- Quadratic Probing
- Double Hashing

All these try cells $h(k,0), h(k,1), h(k,2), \ldots, h(k, m-1)$
where $h(k,i) = (h'(k) + f(i)) \bmod m$, with $f(0) = 0$

The function f is the collision resolution strategy and the function h' is the original (now auxiliary) hash function.

Linear Probing

Function f is linear. Typically, f(i) = i

So, h(k, i) = (h′(k) + i) mod m

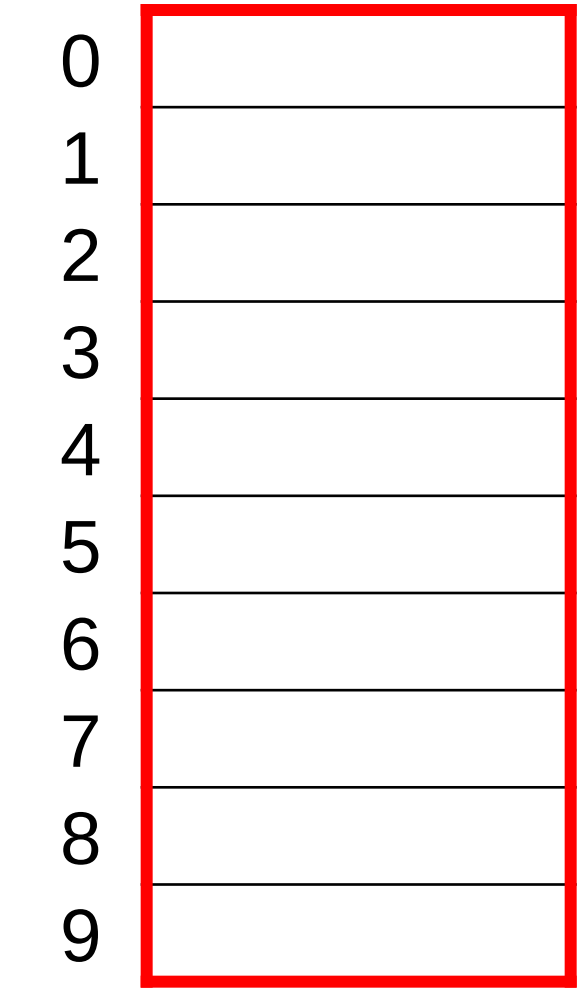
Offsets: 0, 1, 2, …, m-1

With H = h′(k), we try the following cells with wraparound:

H, H + 1, H + 2, H + 3, …

What does the table look like after the following insertions?

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81



General Open Addressing Insertion Algorithm

```
Hash-Insert( T, k )
    i ← 0
    repeat
        j ← h( k, i )
        if T[ j ] = NIL
            then T[ j ] ← k
                 return j
            else  i ← i + 1
    until i = m
    error "hash table overflow"
```

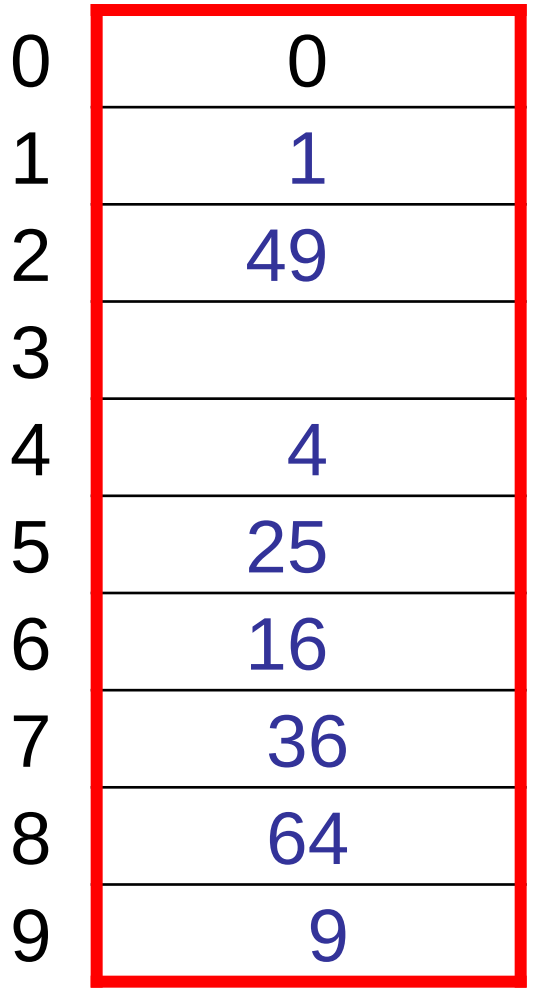
General Open Addressing Search Algorithm

```
Hash-Search( T, k )
    i ← 0
    repeat
        j ← h( k, i )
        if T[ j ] = k
            then return j
        i ← i + 1
    until T[ j ] = NIL or i = m
    return NIL
```

Linear Probing Deletion

How do we delete 9?

How do we find 49 after deleting 9?

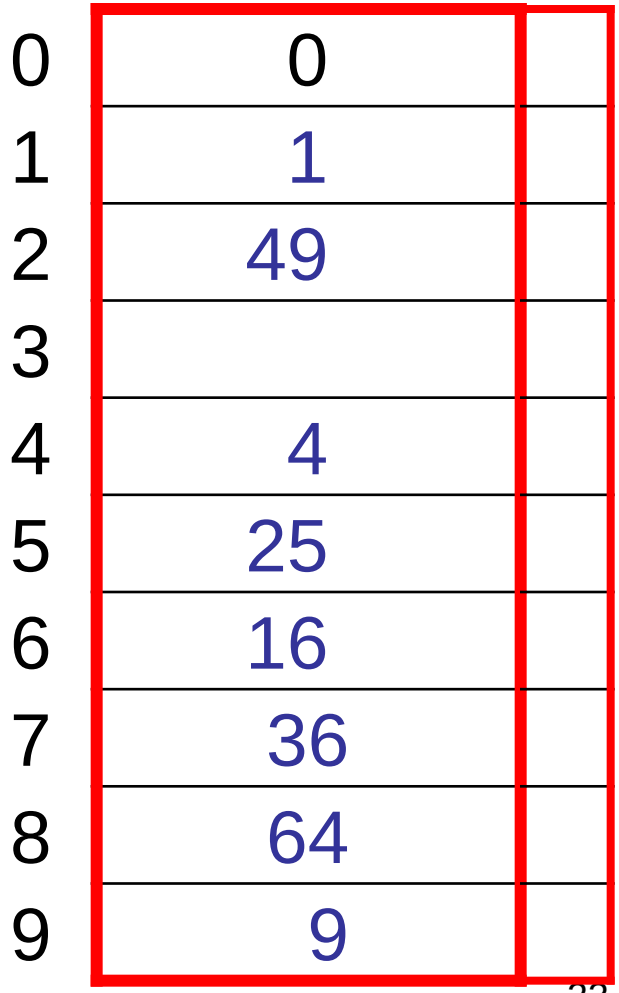


Lazy Deletion

Empty: Null reference

Active: A

Deleted: D



Questions to Ask When Analyzing Resolution Schemes

1. Are we guaranteed to find an empty cell if there is one?
2. Are we guaranteed we won't be checking the same cell twice during one insertion?
3. What should the load factor be to obtain O(1) average-case insert, search, and delete?

Answers for Linear Probing:

1.
2.
3.

Primary Clustering

Linear Probing is easy to implement, but it suffers from the problem of *primary clustering*:

Hashing several times in one area results in a cluster of occupied spaces in that area. Long runs of occupied spaces build up and the average search time increases.

Collision Resolution Comparison

	Advantages?	Disadvantages?
Chaining		
Linear Probing		

Rehashing

Problem with both chaining & probing:

When the table gets too full, the average search time deteriorates from O(1) to O(n).

Solution: Create a larger table and then rehash all the elements into the new table

Time analysis:

Quadratic Probing

Function f is quadratic. Typically, $f(i) = i^2$

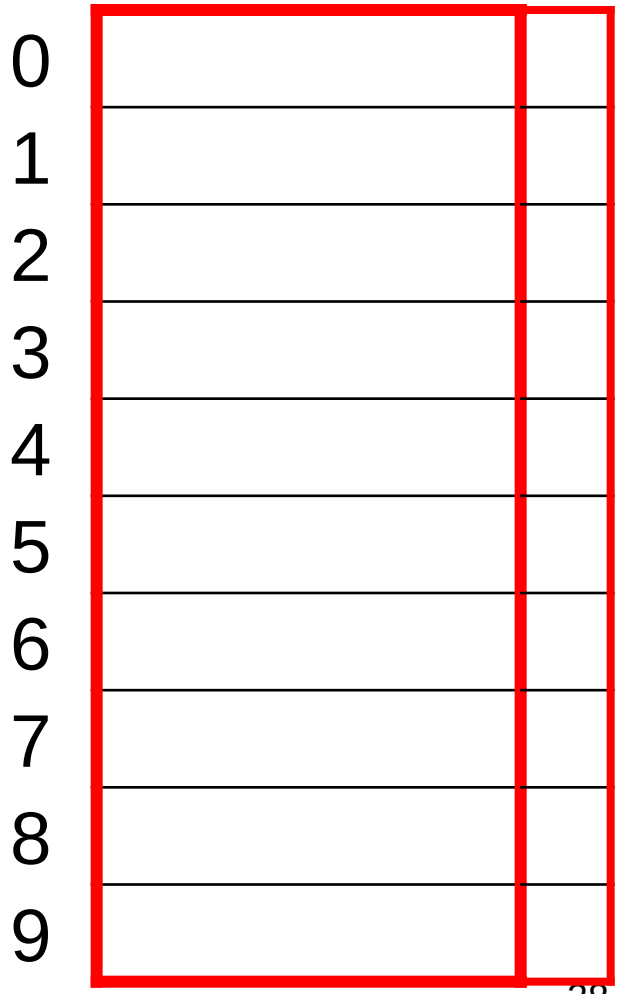
So, $h(k, i) = (h'(k) + i^2) \bmod m$

Offsets: 0, 1, 4, …

With $H = h'(k)$, we try the following cells with wraparound:

$H, H + 1^2, H + 2^2, H + 3^2$ …

Insert Keys: 10, 23, 14, 9, 16, 25, 36, 44, 33



Questions to Ask When Analyzing Resolution Schemes

1. Are we guaranteed to find an empty cell if there is one?
2. Are we guaranteed we won't be checking the same cell twice during one insertion?
3. What should the load factor be to obtain O(1) average-case insert, search, and delete?

Answers for Quadratic Probing:

1.

2.

3.

Secondary Clustering

Quadratic Probing suffers from a milder form of clustering called *secondary clustering*:

As with linear probing, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1,0) = h(k_2,0)$ implies $h(k_1,1) = h(k_2,1)$. So only m distinct probes are used.

Therefore, clustering can occur around the probe sequences.

Advantages/Disadvantages of Quadratic Probing?

Double Hashing

If a collision occurs when inserting, apply a second auxiliary hash function, $h_2(k)$, and probe at a distance $h_2(k)$, $2 * h_2(k)$, $3 * h_2(k)$, etc. until find empty position.

So, $f(i) = i * h_2(k)$ and we have two auxiliary functions:

$$h(k, i) = (h_1(k) + i * h_2(k)) \bmod m$$

With $H = h_1(k)$, we try the following cells in sequence with wraparound:

H

H + $h_2(k)$

H + 2 * $h_2(k)$

H + 3 * $h_2(k)$

…

Double Hashing

In order for the entire table to be searched, the value of the second hash function, $h_2(k)$, must be relatively prime to the table size m.

One of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations

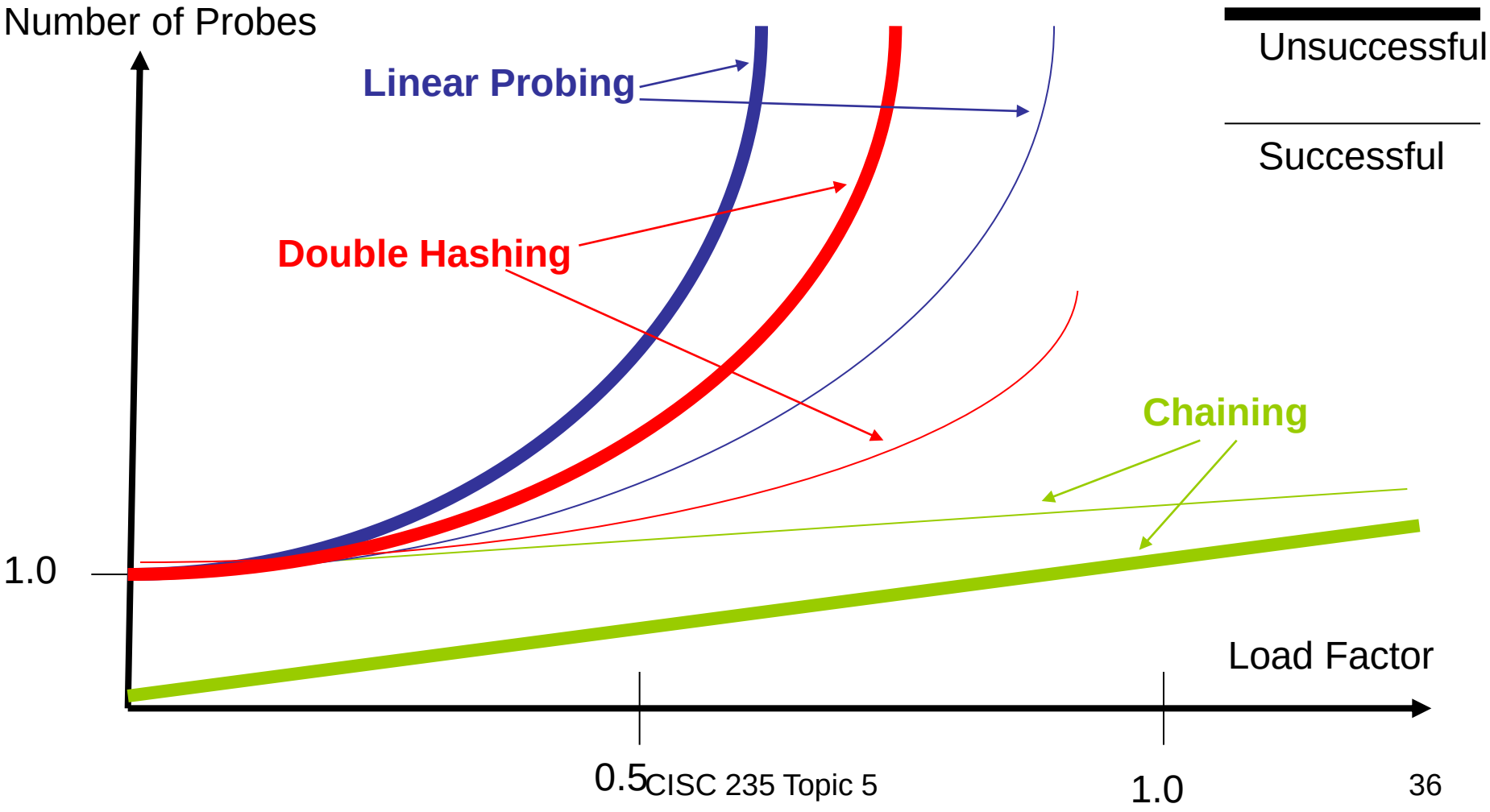
Advantages/Disadvantages of Double Hashing?

Collision Resolution Comparison:
Expected Number of Probes in Searches

Let $\lambda = n / m$ (load factor)

	Unsuccessful Search	Successful Search
Chaining	λ (average number of elements in chain)	$1 + \lambda/2 - \lambda/(2n)$ (1 + average number before element in chain)
Open Addressing (assuming uniform hashing)	$1 / (1 - \lambda)$	$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$

Expected Number of Probes vs. Load Factor



Collision Resolution Comparison

Let $\lambda = n / m$ (load factor)

	Recommended Load Factor
Chaining	$\lambda \leq 1.0$
Linear or Quadratic Probing	$\lambda \leq 0.5$ (half full)
Double Hashing	$\lambda \leq 0.5$ (half full)

Note: If a table using quadratic probing is more than half full, it is not guaranteed that an empty cell will be found

Collision Resolution Comparison

	Advantages?	Disadvantages?
Chaining		
Linear Probing		
Quadratic Probing		
Double Hashing		

Choosing Hash Functions

A good hash function must be O(1) and must distribute keys evenly.

Division Method Hash Function for Integer Keys:

$h(k) = k \bmod m$

Hash Function for String Keys?

Hash Functions for String Keys (assume English words as keys)

Option 1: Use all letters of key

h(k) = (sum of ASCII values in Key) mod m

So,

$$h(k) = \left(\sum_{i=0}^{keysize-1} (int)k[i] \right) \bmod m$$

Good hash function?

Hash Functions for String Keys (assume English keys)

Option 2: Use first three letters of a key & multiplier

```
h( k ) =
    ( (int) k[0] +
      (int) k[1] * 27 +
      (int) k[2] * 729 ) mod m
```

Note: 27 is number of letters in English + blank

729 is 27^2

Using 3 letters, so $26^3 = 17,576$ possible combos, not including blanks

Good hash function?

Hash Functions for String Keys (assume English keys)

Option 3: Use all letters of a key & multiplier

$$h(k) = \left(\sum_{i=0}^{keysize-1} (int)k[i] * 128^i \right) \bmod m$$

Note: Use Horner's rule to compute the polynomial efficiently

Good hash function?

Requirement: Prime Table Size for Division Method Hash Functions

If the table is not prime, the number of alternative locations can be severely reduced, since the hash position is a value mod the table size

Example: Table Size 16, with Quadratic Probing

h′(k)	+	Offset
0	+	1 mod 16 = 1
		4 mod 16 = 4
		9 mod 16 = 9
		16 mod 16 = 0
		25 mod 16 = 9
		36 mod 16 = 4
		49 mod 16 = 1
		…

Important Factors When Designing Hash Tables

To Minimize Collisions:

1. Distribute the elements evenly.
   - Use a hash function that distributes keys evenly
   - Make the table size, m, a prime number not near a power of two if using a division method hash function
2. Use a load factor, $\lambda = n / m$, that's appropriate for the implementation.
   - 1.0 or less for chaining (i.e., $n \leq m$).
   - 0.5 or less for linear or quadratic probing or double hashing (i.e., $n \leq m / 2$)