

# Hashing

---



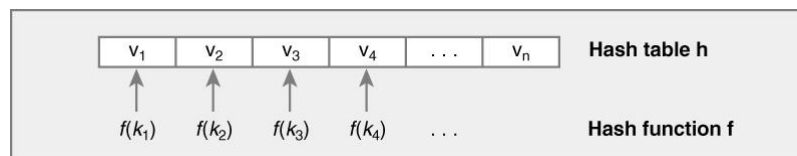
**Nelson Padua-Perez**  
**Chau-Wen Tseng**

**Department of Computer Science**  
**University of Maryland, College Park**

## Hashing

### ■ Approach

- Transform key into number (hash value)
- Use hash value to **index** object in **hash table**
- Use **hash function** to convert key to number



# Hashing

## ■ Hash Table

- Array indexed using hash values
- Hash Table A with size N
- Indices of A range from 0 to N-1
- Store in  $A[\text{hashValue} \% N]$

<i>Hash table h</i>		<i>Location</i>	<i>Key</i>
$h[0]$	$\Lambda$	0	$\Lambda$
$h[1]$	$\Lambda$	1	10
...	...	2	15
$h[N - 1]$	$\Lambda$	3	20
		4	$\Lambda$
		...	

# Hash Function

## ■ Goal

- Scatter values uniformly across range
- $\text{Hash}(\text{<everything>}) = 0$ 
  - Satisfies definition of hash function
  - But not very useful

## ■ Multiplicative congruency method

- Produces good hash values
- Hash value =  $(a \times \text{int}(\text{key})) \% N$
- Where
  - N is table size
  - a, N are large primes

## Hash Function

### ■ Example

hashCode("apple") = 5  
hashCode("watermelon") = 3  
hashCode("grapes") = 8  
hashCode("kiwi") = 0  
hashCode("strawberry") = 9  
hashCode("mango") = 6  
hashCode("banana") = 2

### ■ Perfect hash function

- Unique values for each key

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	
8	grapes
9	strawberry

## Hash Function

### ■ Suppose now

hashCode("apple") = 5  
hashCode("watermelon") = 3  
hashCode("grapes") = 8  
hashCode("kiwi") = 0  
hashCode("strawberry") = 9  
hashCode("mango") = 6  
hashCode("banana") = 2  
hashCode("orange") = 3

### ■ Collision

- Same hash value for multiple keys

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	
8	grapes
9	strawberry

## Types of Hash Tables

### ■ Open addressing

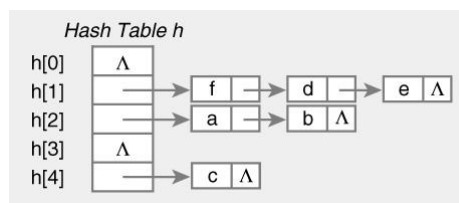
- Store objects in each table entry

Hash table  $h$

$h[0]$	$(k_4, v_4)$
$h[1]$	$\Lambda$
$h[2]$	$\Lambda$
$h[3]$	$(k_1, v_1)$
$h[4]$	$(k_3, v_3)$
$h[5]$	$(k_2, v_2)$

### ■ Chaining (bucket hashing)

- Store lists of objects in each table entry



## Open Addressing Hashing

### ■ Approach

- Hash table contains objects
- Probe  $\Rightarrow$  examine table entry
- Collision
  - Move  $K$  entries past current location
  - Wrap around table if necessary
- Find location for  $X$ 
  1. Examine entry at  $A[\text{key}(X)]$
  2. If entry =  $X$ , found
  3. If entry = empty,  $X$  not in hash table
  4. Else increment location by  $K$ , repeat

## Open Addressing Hashing

### ■ Approach

#### ■ Linear probing

- $K = 1$
- May form **clusters** of contiguous entries

#### ■ Deletions

- Find location for X
- If X inside cluster, leave **non-empty** marker

#### ■ Insertion

- Find location for X
- Insert if X not in hash table
- Can insert X at first non-empty marker

## Open Addressing Example

### ■ Hash codes

- $H(A) = 6$      $H(C) = 6$
- $H(B) = 7$      $H(D) = 7$

### ■ Hash table

- Size = 8 elements
- $\Lambda$  = empty entry
- \* = non-empty marker

### ■ Linear probing

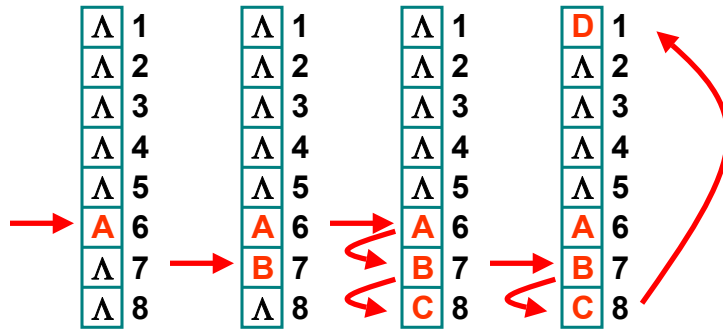
- Collision  $\Rightarrow$  move 1 entry past current location

$\Lambda$	1
$\Lambda$	2
$\Lambda$	3
$\Lambda$	4
$\Lambda$	5
$\Lambda$	6
$\Lambda$	7
$\Lambda$	8

## Open Addressing Example

### Operations

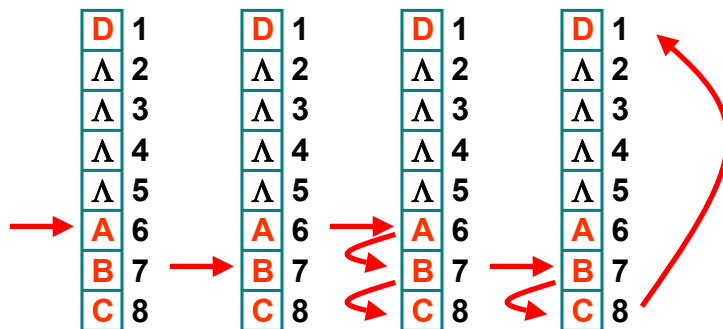
- Insert A, Insert B, Insert C, Insert D



## Open Addressing Example

### Operations

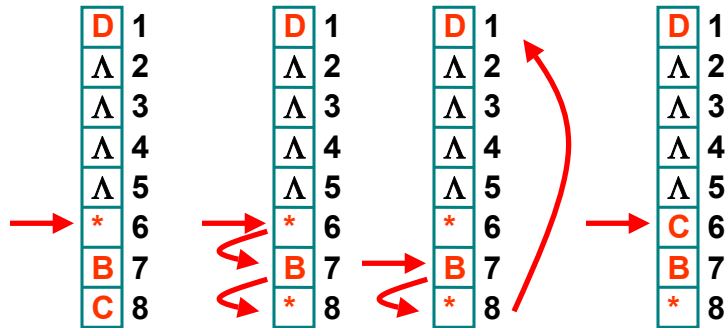
- Find A, Find B, Find C, Find D



## Open Addressing Example

### Operations

- Delete A, Delete C, Find D, Insert C



## Efficiency of Open Hashing

- Load factor = entries / table size
- Hashing is efficient for load factor < 90%

$\alpha$	Number of Comparisons	Approximate Behavior	(Table Size $N = 100$ )
0.1	1.06	O(1)	
0.2	1.13		
0.3	1.21		
0.4	1.33		
0.5	1.50		
0.6	1.75		
0.7	2.17	O(log N)	
0.8	3.00		
0.9	5.50		
0.95	10.5	O(N)	
0.98	26.5		
0.99	50.5		

## Chaining (Bucket Hashing)

### ■ Approach

- Hash table contains lists of objects
- Find location for **X**
  - Find hash code **key** for X
  - Examine list at table entry  $A[\text{key}]$
- Collision
  - Multiple entries in list for entry

## Chaining Example

### ■ Hash codes

- $H(A) = 6$       $H(C) = 6$
- $H(B) = 7$       $H(D) = 7$

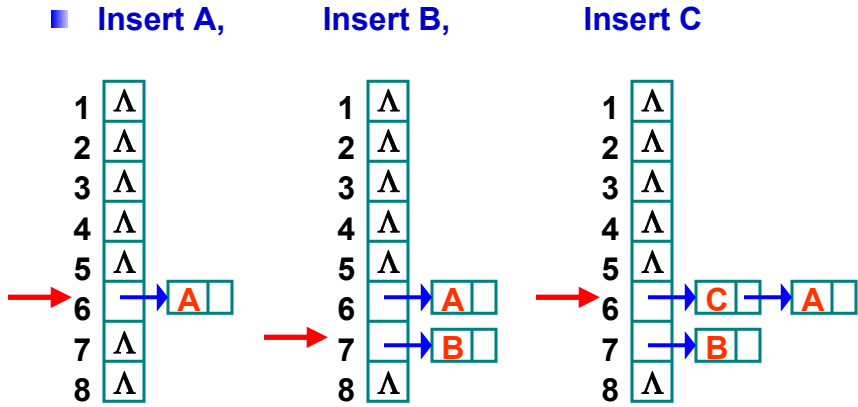
### ■ Hash table

- Size = 8 elements
- $\Lambda$  = empty entry

1	$\Lambda$
2	$\Lambda$
3	$\Lambda$
4	$\Lambda$
5	$\Lambda$
6	$\Lambda$
7	$\Lambda$
8	$\Lambda$

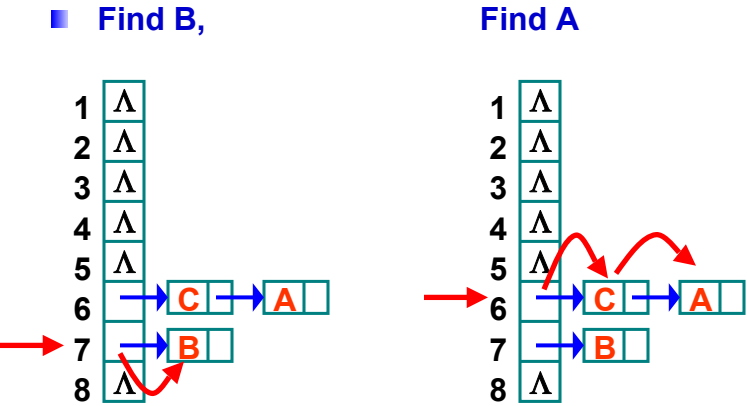
# Chaining Example

Operations



# Chaining Example

Operations



## Efficiency of Chaining

- **Load factor = entries / table size**
- **Average case**
  - Evenly scattered entries
  - Operations =  $O(\text{load factor})$
- **Worse case**
  - Entries mostly have same hash value
  - Operations =  $O(\text{entries})$

## Hashing in Java

- **Collections**
  - `HashMap` & `HashSet` implement hashing
- **Objects**
  - Built-in support for hashing
    - `boolean equals(Object o)`
    - `int hashCode()`
  - Can override with own definitions
  - Must be careful to support Java contract

## Java Contract

- **hashCode()**
  - Must return same value for object in each execution, provided no information used in equals comparisons on the object is modified
- **equals()**
  - if `a.equals(b)`, then `a.hashCode()` must be the same as `b.hashCode()`
  - if `a.hashCode() != b.hashCode()`, then `!a.equals(b)`
- **`a.hashCode() == b.hashCode()`**
  - Does not imply `a.equals(b)`
  - Though Java libraries will be more efficient if it is true