

# CMSC 132: Object-Oriented Programming II

---



## Hashing

Department of Computer Science  
University of Maryland, College Park

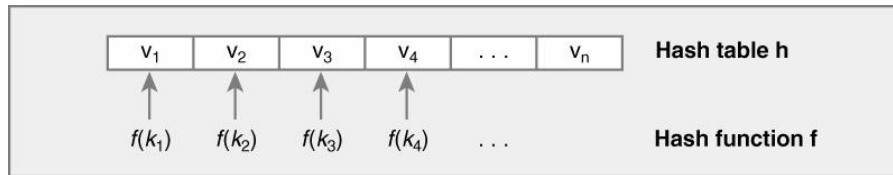
### Overview

- **Hashing**
  - Scattering Hash Values
  - Hash Function
- **Hash Tables**
  - Open Addressing
  - Chaining

# Hashing

## ■ Approach

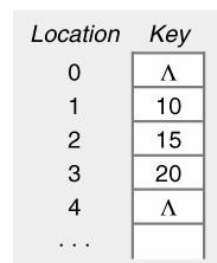
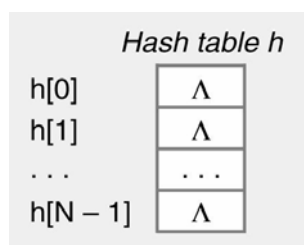
- Use hash function to convert key into number (hash value) used as index in hash table



# 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  $A[\text{hashValue} \% N]$



## Hash Function

- Function for converting key into hash value
- For Java
  - Hash value  $\Rightarrow$  32-bit signed int
  - Default hash function  $\Rightarrow$  int hashCode( )
- For hash table of size N
  - Must reduce hash value to  $0..N - 1$

## Scattering Hash Values

- Should scatter hash values uniformly across range of possible values
  - Hash( <everything> ) = 0
    - Satisfies definition of hash function
    - But not very useful (all keys at same location)
- Could use  $\text{Math.abs}(\text{key.hashCode}() \% N)$ 
  - Might not distribute values well
  - Particularly if N is a power of 2

## Scattering Hash Values

- **Multiplicative congruency method**
  - Produces good hash values
  - Hash value =  $\text{Math.abs}((a * \text{key.hashCode}()) \% N)$
  - Where
    - N is table size
    - a is large prime number

## Beware of % (Modulo Operator)

- The % operator is integer remainder
$$x \% y == x - y * (x / y)$$
- Result may be negative
$$-|y| < x \% y < +|y|$$
- $x \% y$  has same sign as x
  - $-3 \% 2 = -1$
  - $-3 \% -2 = -1$
- Use  $\text{Math.abs}(x \% N)$ 
  - Rather than  $\text{Math.abs}(x) \% N$
  - In case N is negative

## Art and Magic of hashCode()

- There is no “right” hashCode function
  - Art involved in finding good hashCode function
  - Also for finding hashCode to hashBucket function
- From java.util.HashMap

```
static int hashBucket(Object x, int N) {  
    int h = x.hashCode();  
    h += ~(h << 9);  
    h ^= (h >>> 14);  
    h += (h << 4);  
    h ^= (h >>> 10);  
    return Math.abs(h % N);  
}
```

## 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

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

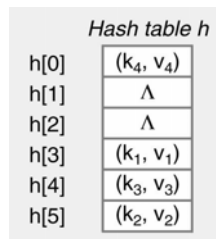
■ **Collision**

- Same hash value for multiple keys

## Types of Hash Tables

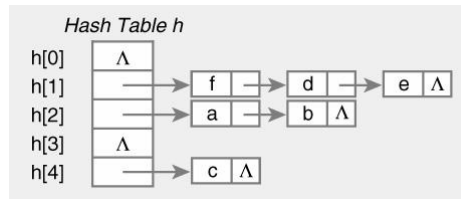
■ **Open addressing**

- Store objects in each table entry



■ **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**

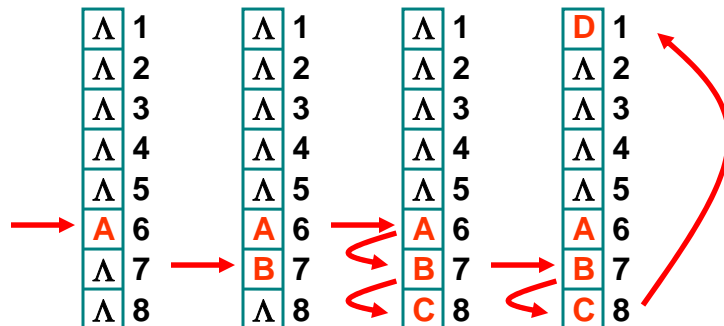
- Collision  $\Rightarrow$  move 1 entry past current location

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

## Open Addressing Example

■ **Operations**

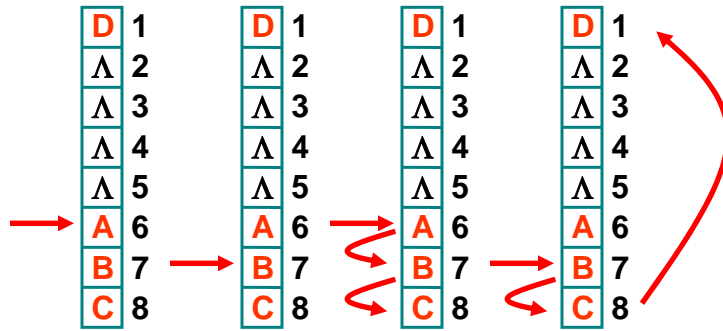
- Insert A, Insert B, Insert C, Insert D



## Open Addressing Example

### ■ Operations

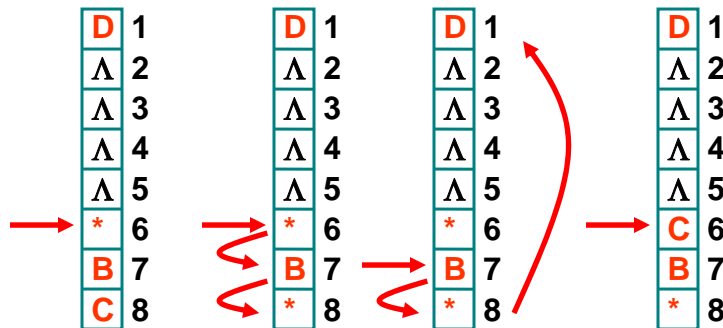
- Find A, Find B, Find C, Find D



## 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	O(N)	
0.95	10.5		
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	$\Lambda$

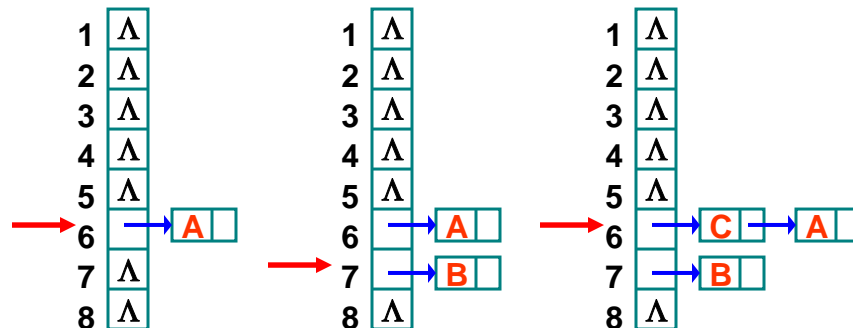
## Chaining Example

### ■ Operations

■ Insert A,

Insert B,

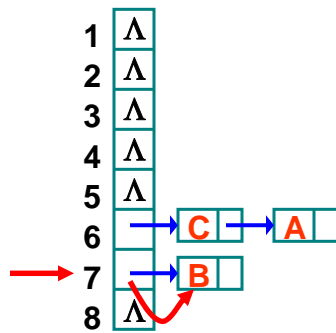
Insert C



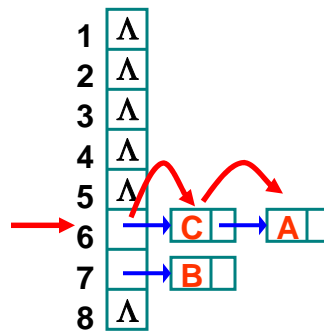
## Chaining Example

### ■ Operations

#### ■ Find B,



#### Find A



## 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
  - if `a.equals(b) == true`
    - then `a.hashCode() == b.hashCode()` must be true