

# CMSC 132: Object-Oriented Programming II

---

## Sets, Maps, Hashing

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

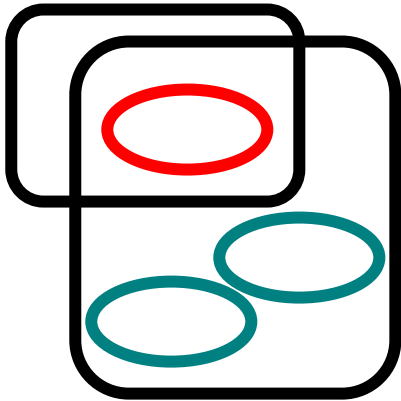


# Overview

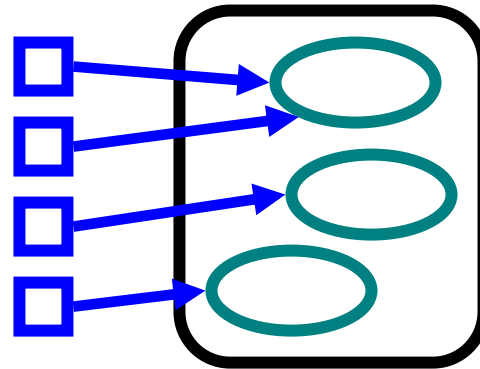
- **Sets**
- **Maps**
- **Hashing**
  - **Scattering Hash Values**
  - **Hash Function**
- **Hash Tables**
  - **Open Addressing**
  - **Chaining**
- **Java equals and hashCode( )**

# Set Data Structures

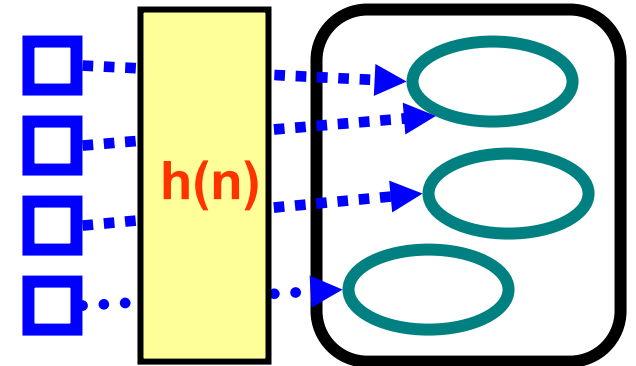
- No relationship between elements
- Types of sets
  - Set
  - Map
  - Hash Table



Set



Map



Hash Table

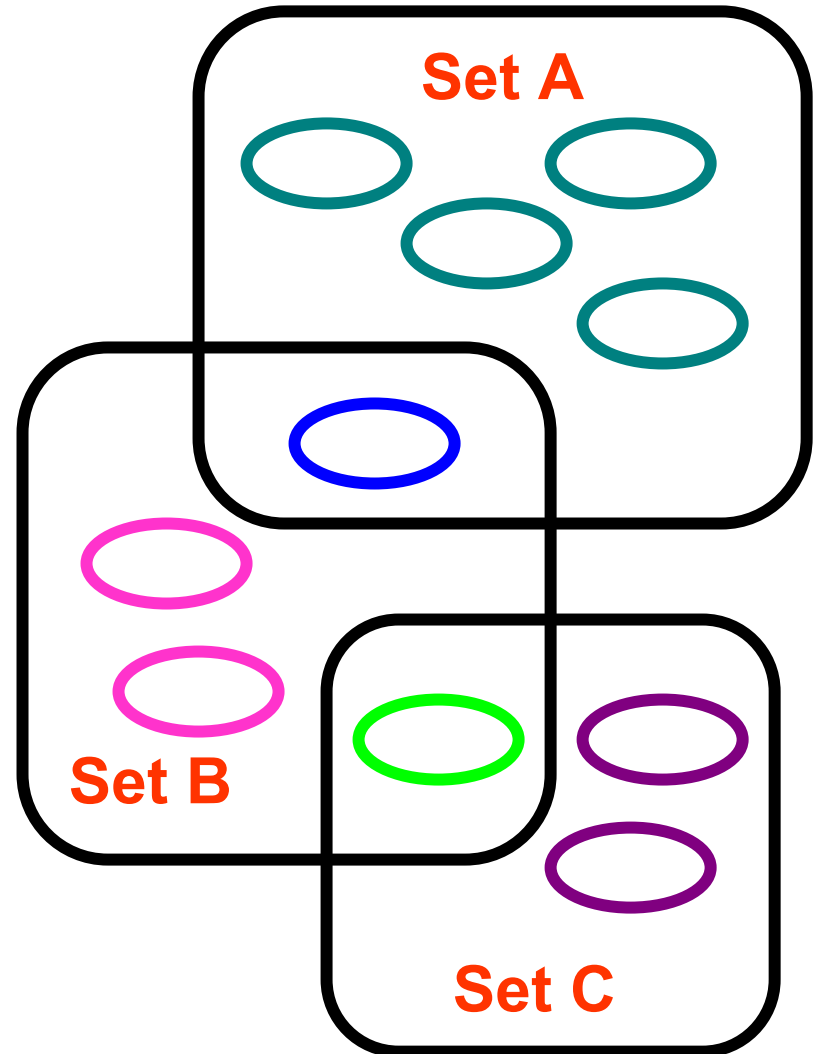
# Sets

## ■ Properties

- Collection of elements without duplicates
- No ordering (i.e., no front or back)
- Order in which elements added doesn't matter

## ■ Implementation goal

- Offer the ability to find / remove element quickly
- Without searching through all elements



# How Do Sets Work in Java?

- Finding matching element is based on equals( )
- To build a collection for a class
  - Need to define your own equals(Object) method
  - Default equals( ) uses reference comparison
    - I.e., a.equals(b) → a == b
    - a, b equal only if reference to same object
  - Many classes have predefined equals( ) methods
    - Integer.equals( ) → compares value of integer
    - String.equals( ) → compares text of string

# Set Concrete Classes

## ■ HashSet

- Elements must implement hashCode( ) method

## ■ LinkedHashSet

- HashSet supporting ordering of elements
- Elements can be retrieved in order of insertion

## ■ TreeSet

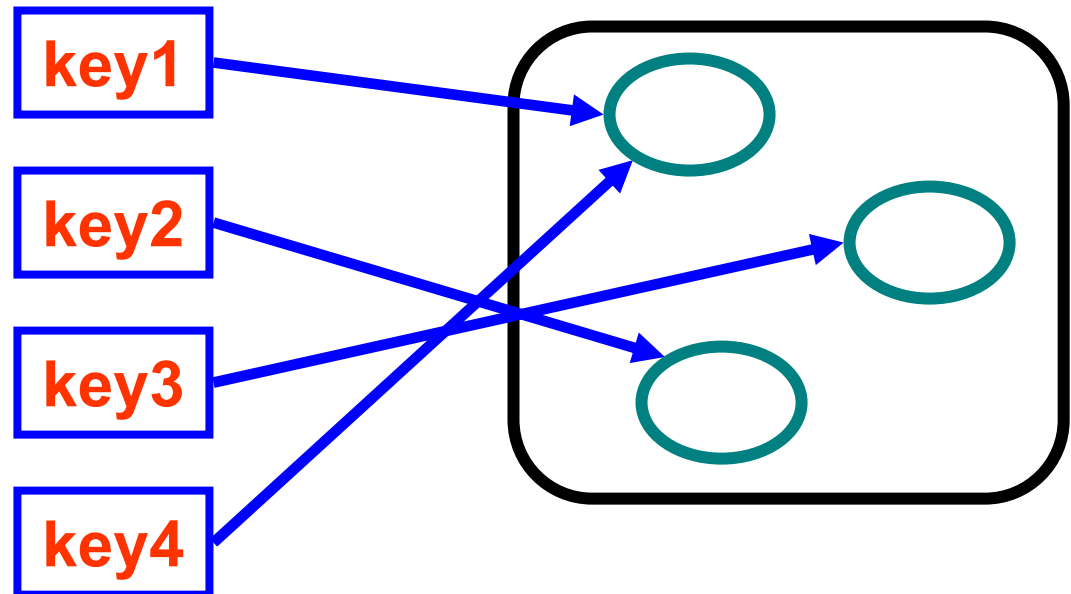
- Elements must be comparable
  - Implement Comparable or provide Comparator
- Guarantees elements in set are sorted

# Map Definition

- **Map (associative array)**
  - Unordered collection of **keys**
  - For each key, an associated object
  - Can use key to retrieve object
- Can view as array indexed by **any** (key) value

- **Example**

A["key1"] = ...



# Map Interface Methods

## ■ Methods

- `void put( Key, Object )` // inserts element
- `Object get( Key )` // returns element
- `void remove( Key )` // removes element
- `Boolean containsKey( Key )` // looks for key
- `Set keySet( )` // entire set of keys

# Map Properties

## ■ Map keys & map objects

### ■ Can also treat keys & values as collections

- Access using `keySet( )`, `values( )`

### ■ Aliasing

- Each key refers only a single object

- But object may be referred to by multiple keys

### ■ Keys & values may be of complex type

- `Map<Object Type1, Any Object Type2>`

- Including other collections, maps, etc...

# Map Implementation

## ■ Implementation approaches

### ■ Two parallel arrays

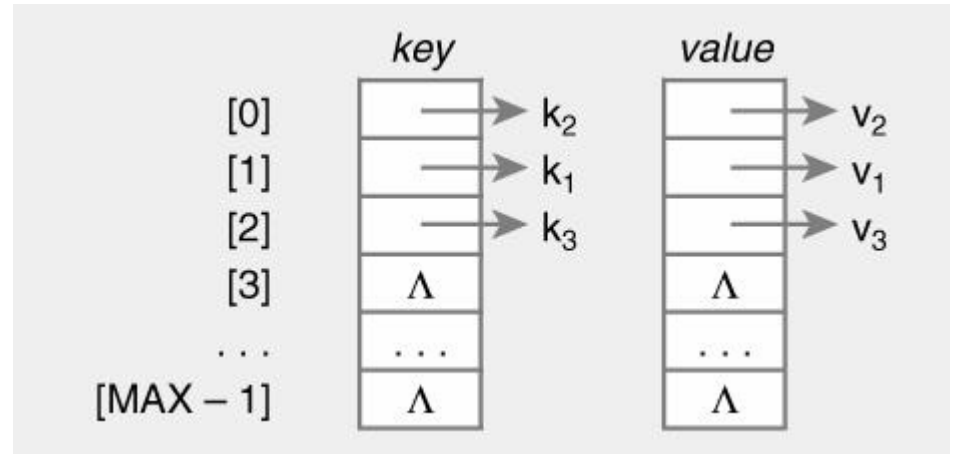
■ Unsorted

■ Sorted

### ■ Linked list

### ■ Binary search tree

### ■ Hash table

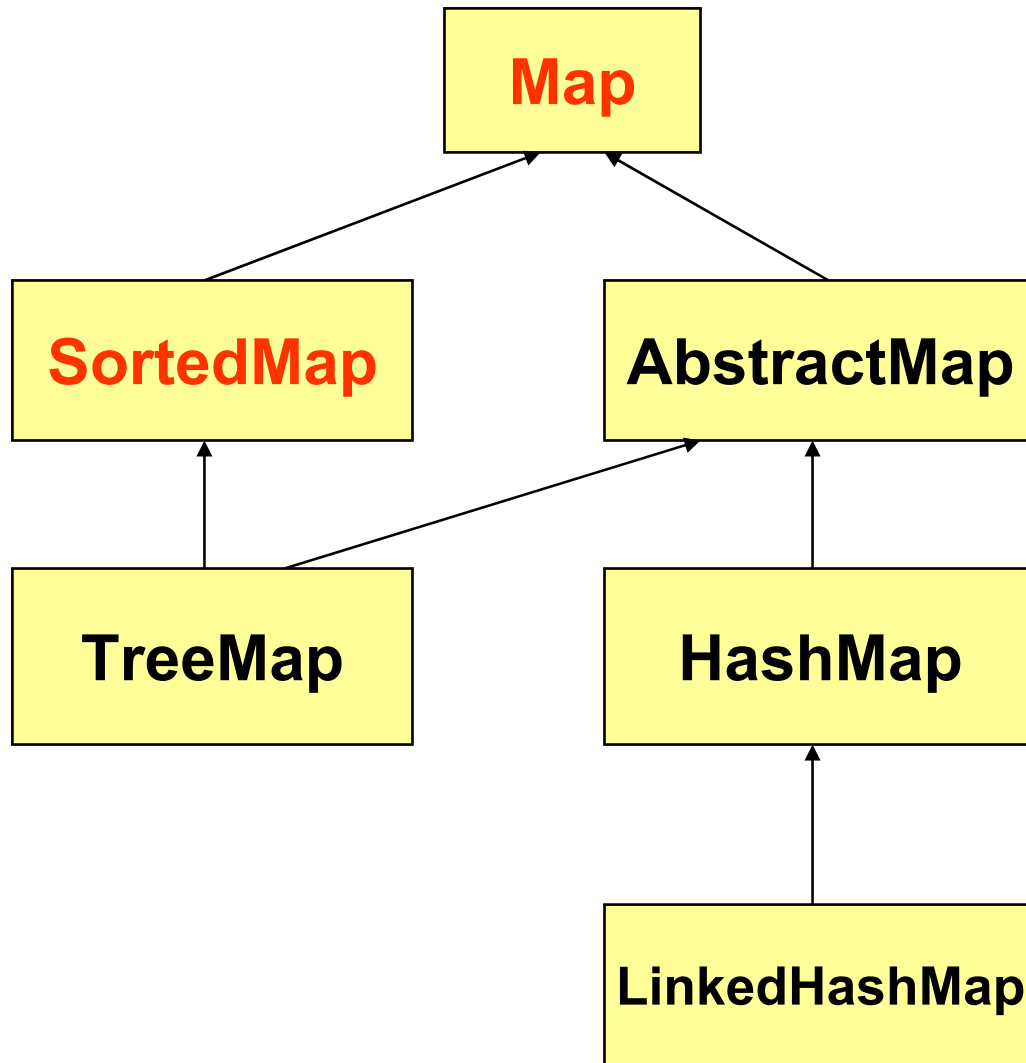


## ■ Java Collections Framework

■ TreeMap → uses red-black (balanced) tree

■ HashMap → uses hash table

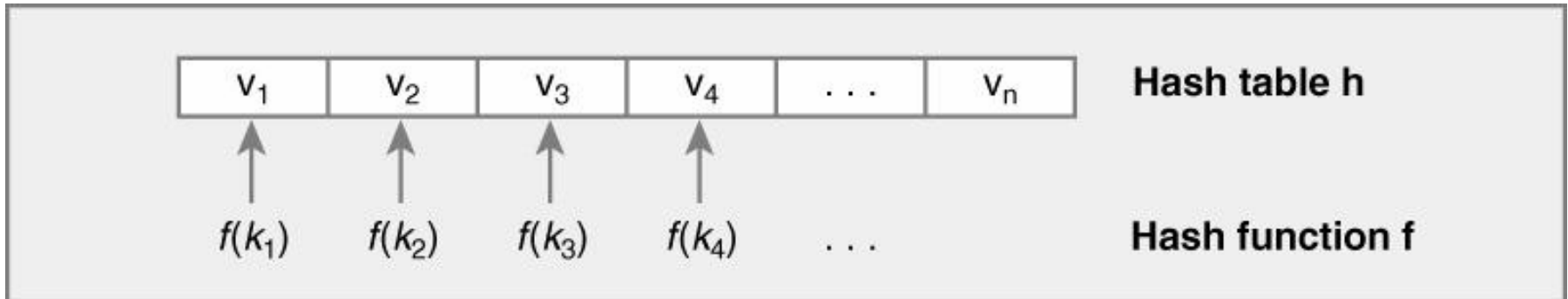
# Java Collections Map Hierarchy



# Hashing

## ■ Approach

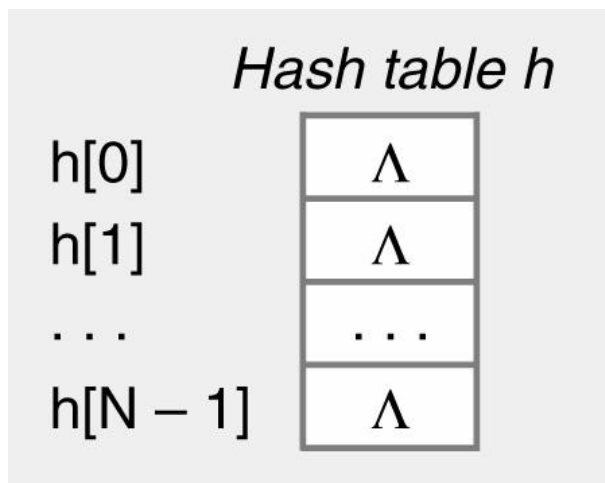
- Use **hash function** to convert key into number (**hash value**) used as index in **hash table**



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



<i>Location</i>	<i>Key</i>
0	$\Lambda$
1	10
2	15
3	20
4	$\Lambda$
$\dots$	

# Hash Function

- **Function for converting key into hash value**

- **For Java**

- **Hash value** ⇒ **32-bit signed int**

- **Default hash function** ⇒ **int hashCode( )**

- **For hash table of size N**

- **Must reduce hash value to  $0..N - 1$**

- **Can use modulo operator**

- **Math.abs(hash value % N)**

# Scattering Hash Values

- Hash function should **scatter** hash values uniformly across range of possible values
  - Reduces likelihood of conflicts between keys
- $\text{Hash}(\langle \text{everything} \rangle) = 0$ 
  - 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 `Math.abs( x % N )`, not `Math.abs(x) % N`

- Since `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE !`

- Will happen 1 in  $2^{32}$  times (on average) for random int values

# 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

## ■ Collision

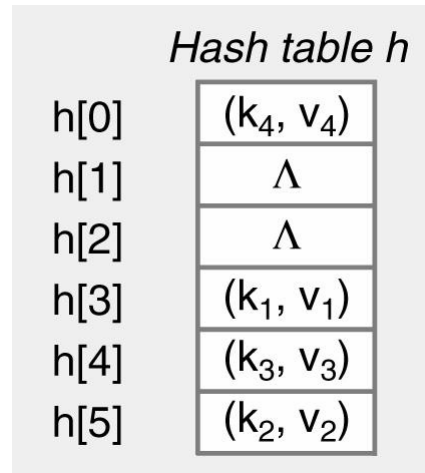
- Same hash value for multiple keys

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

# Types of Hash Tables

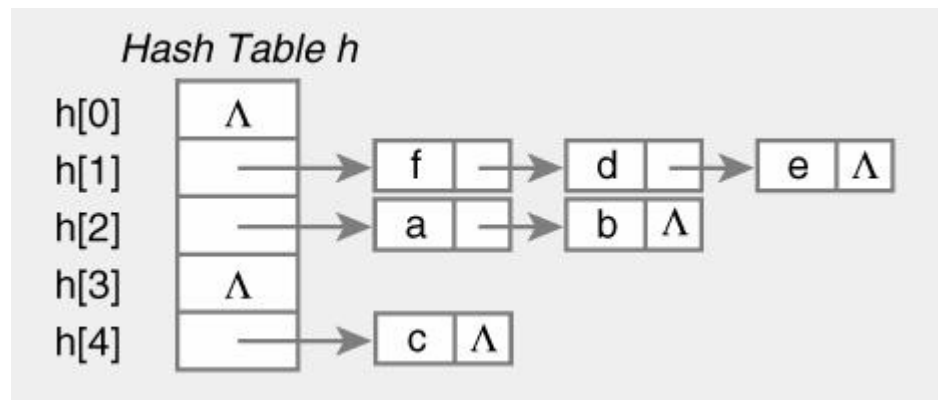
## ■ Open addressing

- Store objects in each table entry



## ■ 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

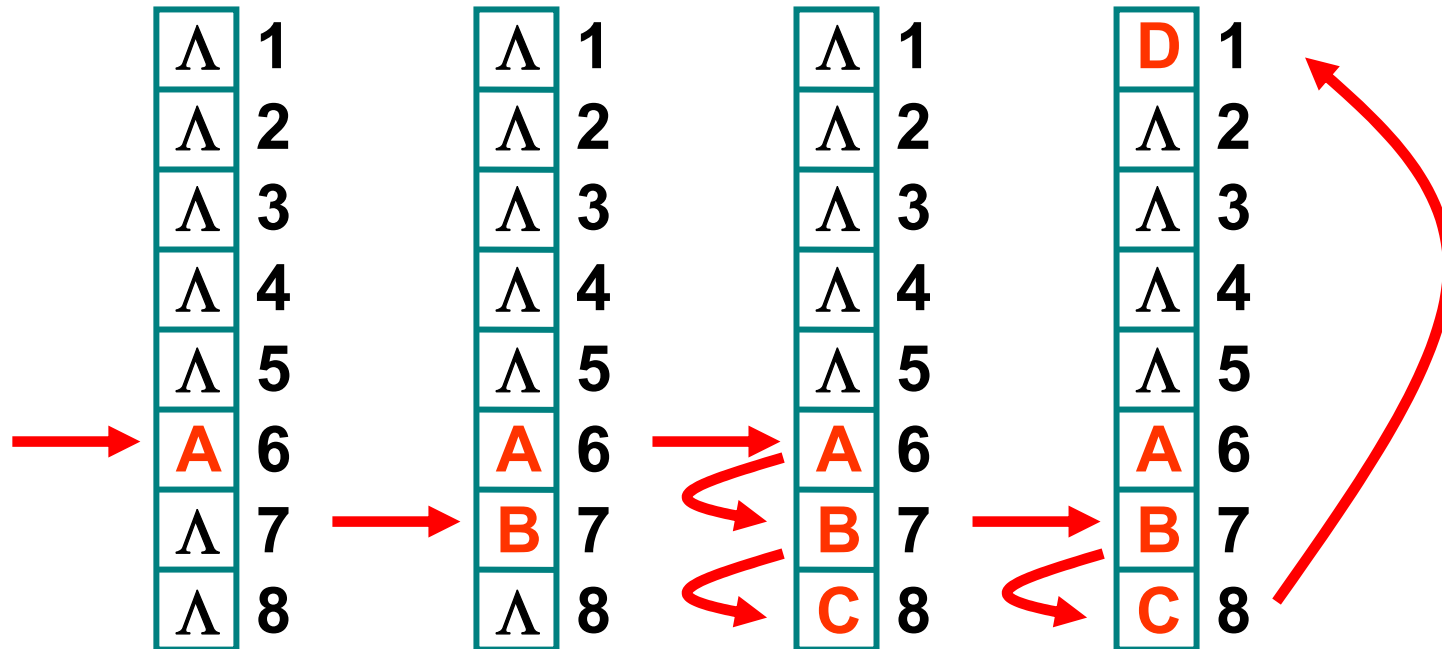
■ Collision  $\Rightarrow$  move 1 entry past current location

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

# Open Addressing Example

## ■ Operations

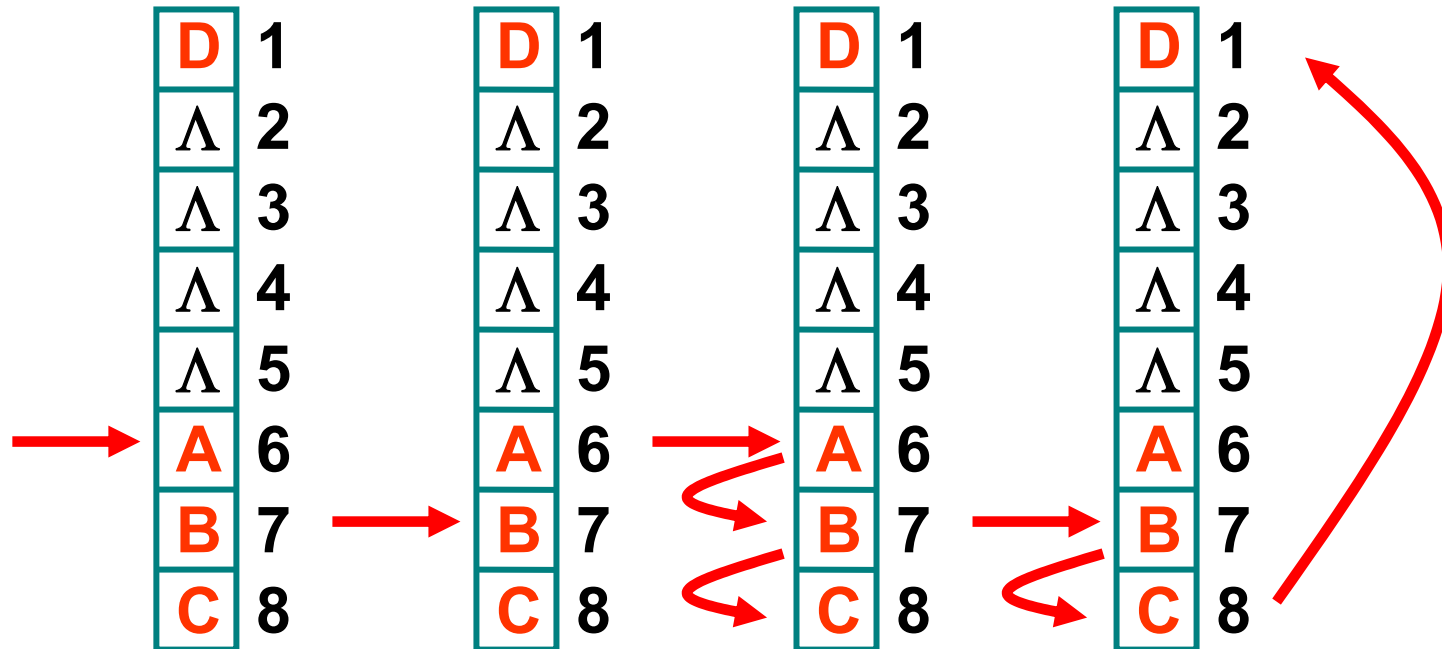
- Insert A, Insert B, Insert C, Insert D



# Open Addressing Example

## ■ Operations

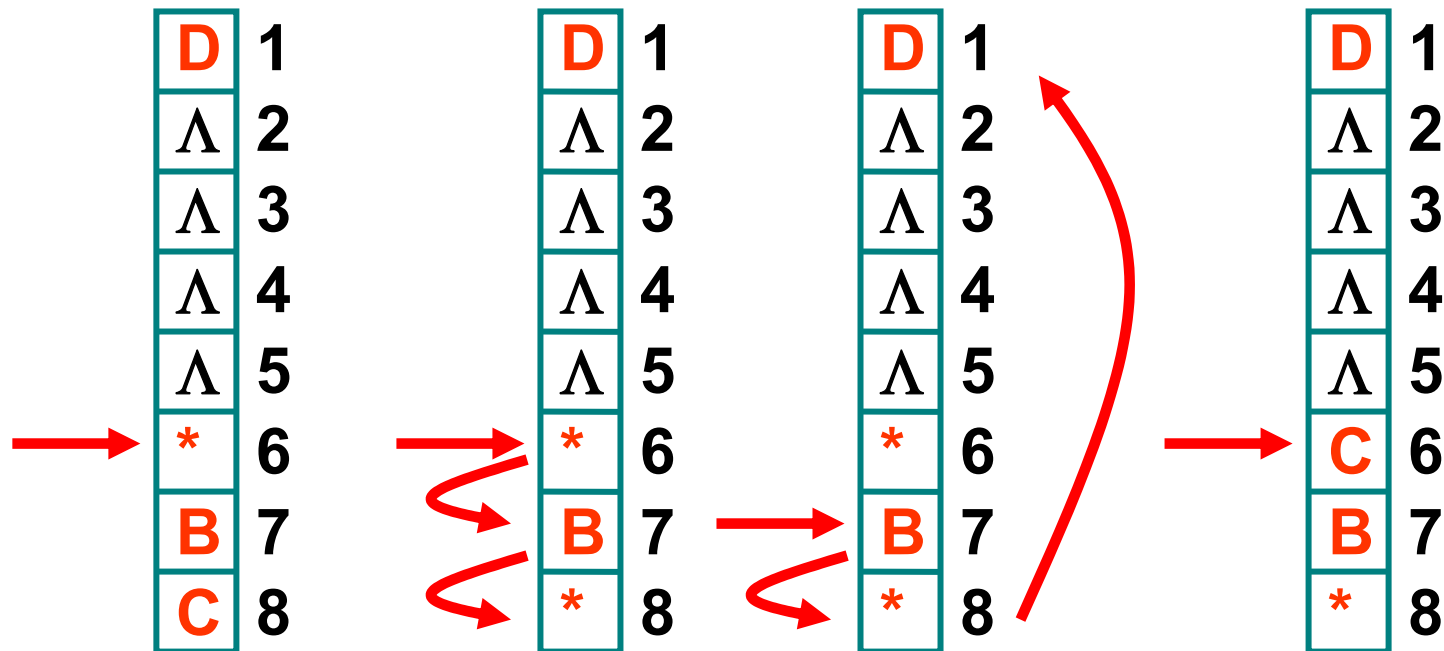
■ Find A, Find B, Find C, Find D



# 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	O(log N)	
0.7	2.17		
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[ 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	$\Lambda$

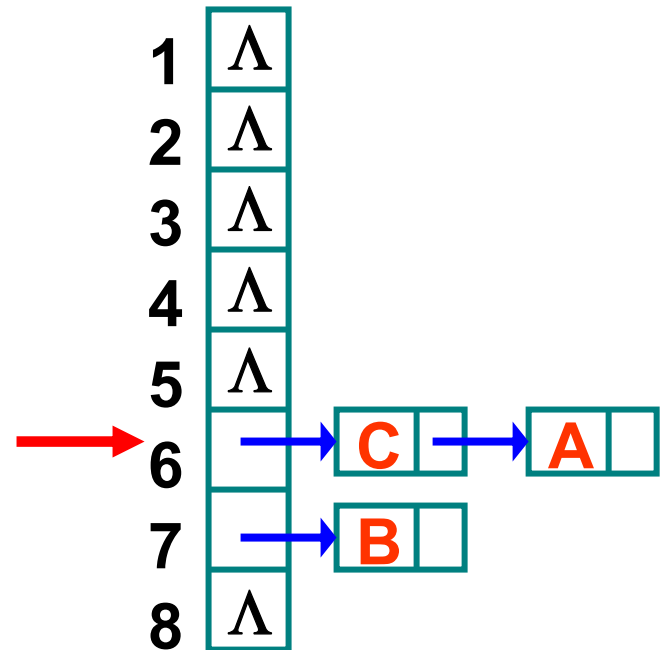
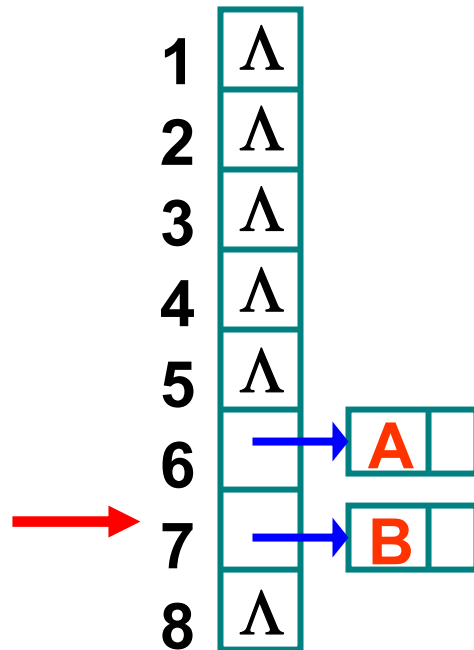
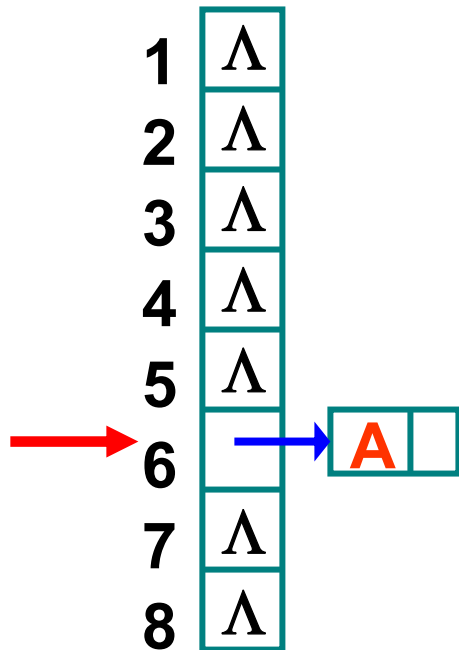
# Chaining Example

## ■ Operations

■ Insert A,

Insert B,

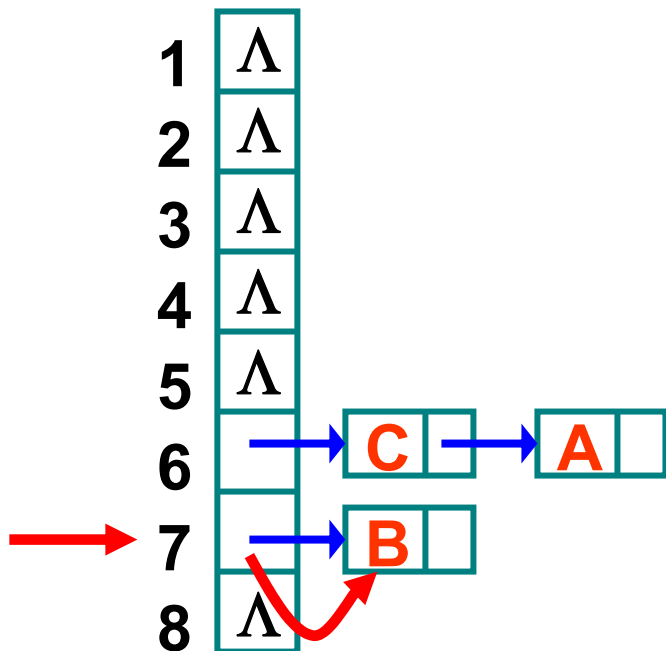
Insert C



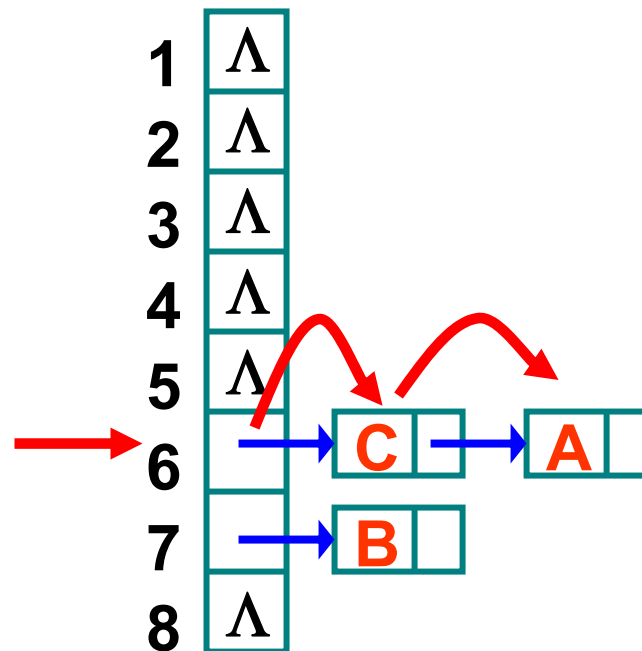
# Chaining Example

## ■ 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

- Object class has built-in support for hashing
  - Method `int hashCode( )` provides
    - Numerical hash value for any object
- `hashCode( )` provides **pre-filter** for `equals( )`
  - Check `equals( )` only if `hashCode( )` is identical
  - Example
    - `if ( a.hashCode( ) == b.hashCode( ) )`  
`result = a.equals( b );`  
`else result = false;`
  - Efficient if `hashCode( )` is faster than `equals( )`

# Hashing in Java

- **Default hashCode( ) implementation**
  - Usually just address of object in memory
  
- **Can override with new user definition**
  - Must work with equals( )
  - Following Java “**hashcode contract**”

# Java Hash Code Contract

## ■ hashCode()

- Must return same value for object in each execution, provided information used in equals( ) comparisons on the object is not modified

## ■ equals()

- if `a.equals(b) == true`, then must **guarantee**  
`a.hashCode( ) == b.hashCode( )`
- Inverse is not true → `!a.equals(b)` does not imply  
`a.hashCode( ) != b.hashCode( )`
  - Though Java libraries may be more efficient
- Converse is also not true → `a.hashCode( ) == b.hashCode( )` does not imply `a.equals(b) == true`

# Java hashCode()

## ■ Implementing hashCode( )

- Include only information used by equals( )
  - Else 2 “equal” objects → different hash values
- Using all / more of information used by equals( )
  - Help avoid same hash value for unequal objects

## ■ Example hashCode( ) functions

- For pair of Strings
  - 1<sup>st</sup> letter of 1<sup>st</sup> str
  - 1<sup>st</sup> letter of 1<sup>st</sup> str + 1<sup>st</sup> letter of 2<sup>nd</sup> str
  - Length of 1<sup>st</sup> str + length of 2<sup>nd</sup> str
  - $\sum$  letter(s) of 1<sup>st</sup> str +  $\sum$  letter(s) of 2<sup>nd</sup> str