Hashing

Dynamic Dictionaries

Operations:

- create
- insert
- find
- remove
- max/ min
- write out in sorted order

Only defined for object classes that are **`Comparable`**

Hash tables

Operations:

- create
- insert
- find
- remove
- ~~max/ min~~
- ~~write out in sorted order~~

Only defined for object classes that ~~are `Comparable`~~ have **`equals`** defined

Hash tables

`public boolean equals(Object obj)`

Java specific: From the Java documentation

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

Hash tables – implementation

- Have a table (an array) of a fixed tableSize
- A hash function determines where in this table each item should be stored

2174 % 10 = 4

hash(item) % tableSize

[a positive integer]

THE DESIGN QUESTIONS

1. Choosing tableSize
2. Choosing a hash function
3. What to do when a collision occurs

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hash tables – tableSize

- Should depend on the (maximum) number of values to be stored
- Let λ = [number of values stored]/ tableSize
  - Load factor of the hash table
  - Restrict λ to be at most 1 (or ½)
- Require tableSize to be a prime number
  - to "randomize" away any patterns that may arise in the hash function values
- The prime should be of the form (4k+3)

  [for reasons to be detailed later]

Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs) hash(k) = k is generally OK, unless the keys have "patterns"

Otherwise, some "randomized" way to obtain an integer

```
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal += key.charAt( i );

    return hashVal % tableSize;
}
```

Figure 5.2 A simple hash function

Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs) hash(k) = k is generally OK, unless the keys have "patterns"

Otherwise, some "randomized" way to obtain an integer

```
public static int hash( String key, int tableSize )
{
    return ( key.charAt( 0 ) + 27 * key.charAt( 1 ) +
                729 * key.charAt( 2 ) ) % tableSize;
}
```

Figure 5.3 Another possible hash function—not too good

```java
/**
 * A hash routine for String objects.
 * @param key the String to hash.
 * @param tableSize the size of the hash table.
 * @return the hash value.
 */
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}
```

Figure 5.4 A good hash function

Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs) hash(k) = k is generally OK, unless the keys have "patterns"

Otherwise, some "randomized" way to obtain an integer

Java-specific

- Every class has a default `hashCode()` method that returns an integer
- May be (<u>should</u> be) overridden
- Required properties

  consistent with the class's `equals()` method

  need not be consistent across different runs of the program

  different objects may return the same value!

Hash tables – the hash function

) = k is

From the Java 1.5.0 documentation

:erns”

http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html#hashCode%28%29

> As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

need not be consistent across different runs of the program

different objects may return the same value!

Hash tables – collision resolution

The universe of possible items is usually far greater than tableSize

Collision: when multiple items hash on to the same location (aka cell or bucket)

Collision resolution strategies specify what to do in case of collision

1. Chaining (closed addressing)
2. Probing (open addressing)
    a. Linear probing
    b. Quadratic probing
    c. Double Hashing
    d. Perfect Hashing
    e. Cuckoo Hashing

Hash tables – implementation

- Have a table (an array) of a fixed tableSize
- A hash function determines where in this table each item should be stored

hash(item) % tableSize

[a positive integer]

THE DESIGN QUESTIONS

1. Choosing tableSize
2. Choosing a hash function
3. What to do when a collision occurs

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hash tables – tableSize

Restrict the load factor λ = [number of values stored]/ tableSize to be

at most 1 (or ½)

Require tableSize to be a prime number of the form (4k + 3)

Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs) hash(k) = k is generally OK, unless the keys have "patterns"

Otherwise, some "randomized" way to obtain an integer

Java-specific

- Every class has a default **`hashCode()`** method that returns an integer
- May be overridden
- Required properties

  consistent with the class's **`equals()`** method

  need not be consistent across different runs of the program

  different objects may return the same value!

Hash tables – collision resolution

The universe of possible items is usually far greater than tableSize

Collision: when multiple items hash on to the same location (aka cell or bucket)

Collision resolution strategies specify what to do in case of collision

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

Hash tables – collision resolution: chaining

Maintain a linked list at each cell/ bucket

(The hash table is an array of linked lists)

Insert: at front of list

- if pre-condition is "not already in list," then faster

- in any case, later-inserted items often accessed more frequently (the LRU principle)

Example: Insert 0^2, 1^2, 2^2, …, 9^2 into an initially empty hash table with tableSize = 10

[Note: bad choice of tableSize – only to make the example easier!!]

Hash tables – collision resolution: chaining

Maintain a linked list at each cell/ bucket

(The hash table is an array of l

Insert: at front of list

- if pre-cond is that not already in

- in any case, later-inserted items

Example: Insert 0^2, 1^2, 2^2, …, 9^2 int

[Note: bad choice of tableSize

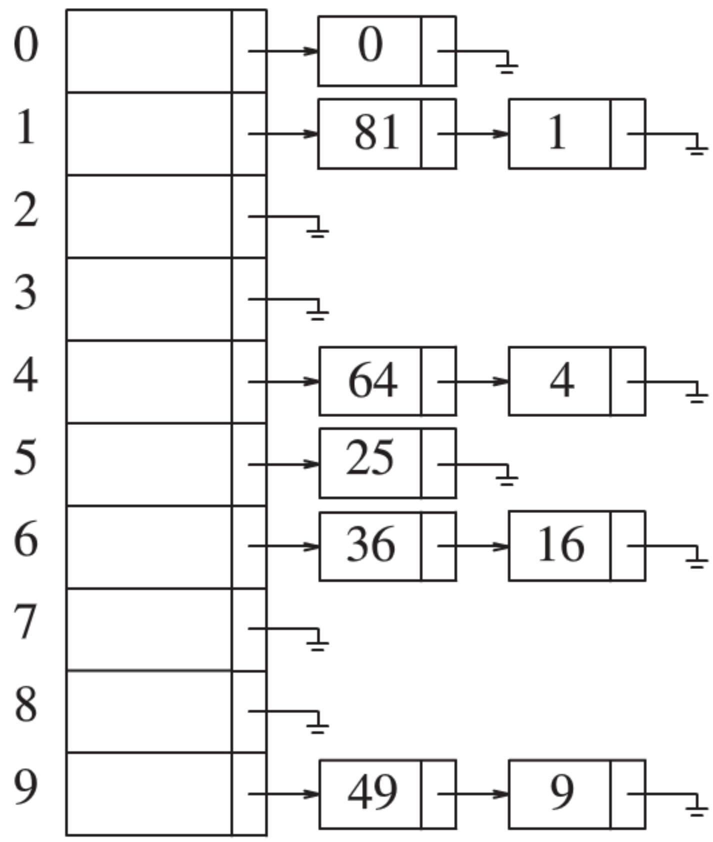


Figure 5.5 A separate chaining hash table

Hash tables – collision resolution: chaining

Maintain a linked list at each cell/ bucket

(The hash table is an array of linked lists)

Insert: at front of list

The load factor: [number of items stored]/tableSize

- if pre-cond is that not already in list, then faster

-in any case, later-inserted items often accessed more frequently

Find and Remove: obvious implementations

Worst-case run-time: $\Theta(N)$ per operation (all elements in the same list)

Average case: $O(\lambda)$ per operation

Design rule: for chaining, keep $\lambda \leq 1$

If λ becomes greater than 1, **rehash** (later)

Hash tables – collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

Avoids the use of dynamic memory

f(i) is a linear function of i – typically, f(i) = i

In case of collision, try alternative locations until an empty cell is found

- [Open address]

Probe sequence: $h_o(x)$, $h_1(x)$, $h_2(x)$, …, with $h_i(x)$ = [hash(x) + f(i)] % tableSize

The function f(i) is different for the different probing methods

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using linear probing

Hash tables – collision resolution: linear probing

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Hash table with linear probing, after each insertion

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using linear probing

Hash tables - review

Supports the basic dynamic dictionary ops: insert, find, remove

Does not need class to be Comparable

Three design decisions: tableSize, hash function, collision resolution

Table size

- a prime of the form (4k+3), keeping load factor constraints in mind

Hash function

- should "randomize" the items
- Java's hashCode() method

Collision resolution: chaining

Collision resolution: probing (open addressing) – linear probing

The clustering problem

Hash tables - clustering

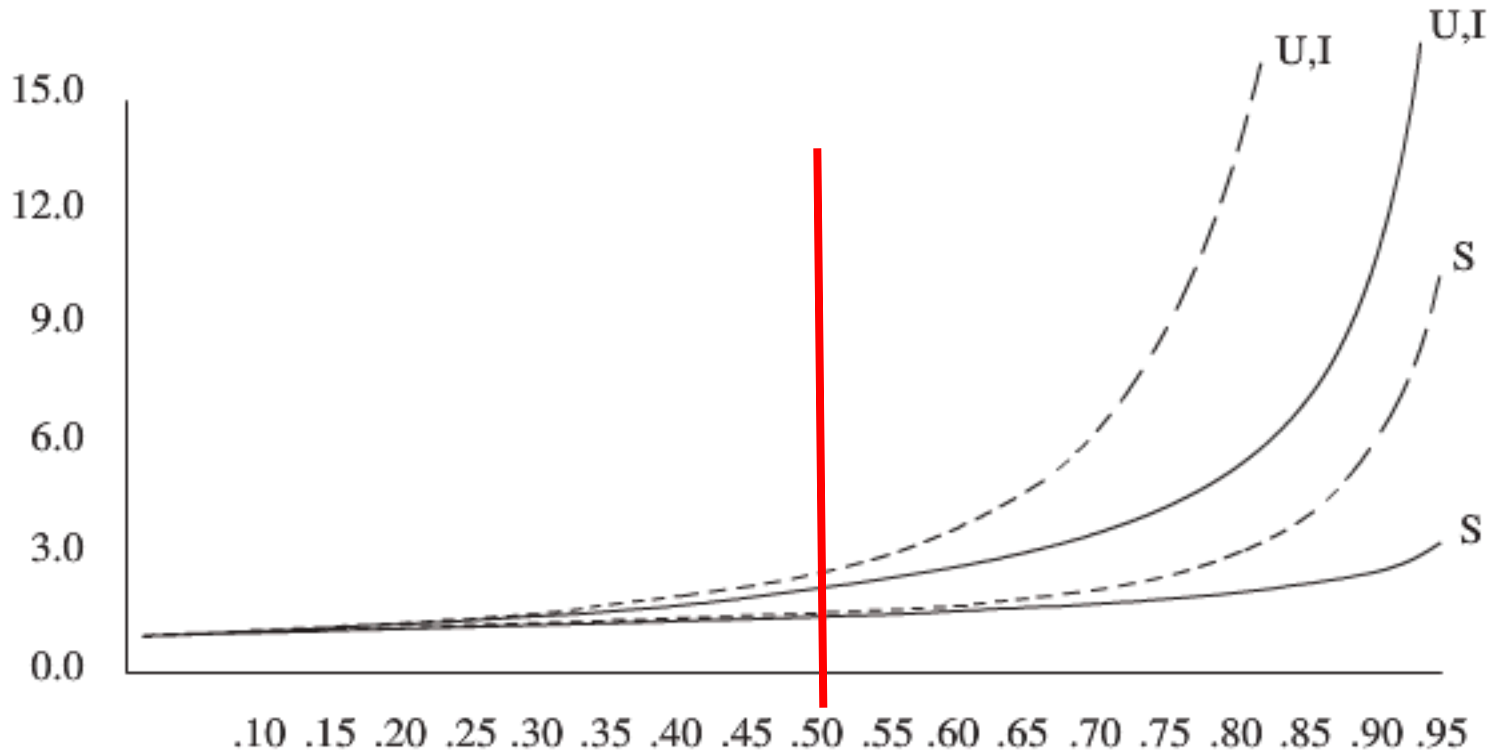
Two causes of clustering:

multiple keys hash on to the same location (secondary clustering)

multiple keys hash on to the same cluster (primary clustering)

Secondary clustering caused by hash function; primary, by choice of probe sequence

Number of probes per operation increases with load factor



Hash tables – collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

f(i) is a quadratic function of i (e.g., $f(i) = i^2$)

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using quadratic probing

Hash tables – collision resolution: quadratic probing

Empty Table
0
1
2
3
4
5
6
7
8
9

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using quadratic probing

Hash tables – collision resolution: quadratic probing

Two causes of clustering:

multiple keys hash on to the same location (secondary clustering)

multiple keys hash on to the same cluster (primary clustering)

Which one does quadratic probing solve?

primary clustering

Efficient implementation of $i^2 \rightarrow (i+1)^2$: (i+1) and (2i+1) in parallel, and then add i^2 and (2i+1)

Choosing tableSize:

-prime: at least half the table gets probed

-prime of the form (4k+3) and probe sequence is $\pm\ i^2$: entire table gets probed

Remove: lazy delete must be used

Hash tables – collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

To get rid of secondary clustering

Use two hash functions: $hash_1(.)$ and $hash_2(.)$

Probe sequence "step" size is $hash_2(.)$

- [Unlikely distinct items agree on <u>both</u> $hash_1(.)$ and $hash_2(.)$]

$hash_2(.)$ must never evaluate to zero!

A common (good) choice: R – (x mod R), for R a prime smaller than tableSize

<u>Example</u>: insert 89, 18, 49, 58, and 69 into a table of size 10, using double hashing with $hash_2(x) = 7 – x \bmod 7$

Hash tables – collision resolution: double hashing

Empty Table
0
1
2
3
4
5
6
7
8
9

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using double hashing with $hash_2(x) = 7 - x \bmod 7$

Hash tables – collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

Hash tables – collision resolution: Cuckoo hashing

Goal: constant-time O(1) find in the worst case

Example application: network routing tables

[remove also takes O(1) time]

Insert has worst-case Θ(N) run-time

Keep two hash tables, and use two different hash functions

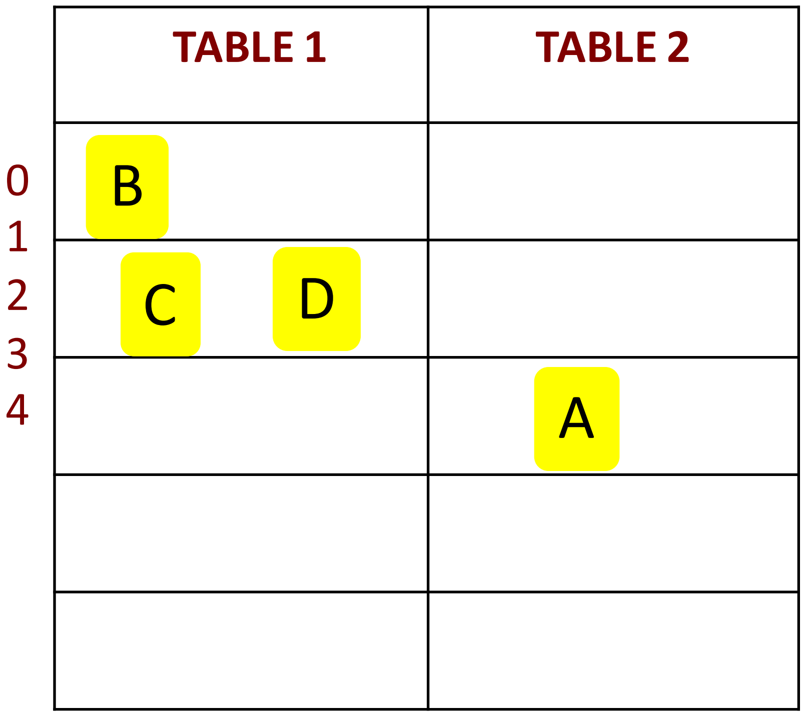
Hash tables – collision resolution: Cuckoo hashing

	TABLE 1	TABLE 2
0	B A	
1		
2		
3		
4		

A: $hash_1(A) = 0$, $hash_2(A) = 2$

B: $hash_1(B) = 0$, $hash_2(B) = 0$

Hash tables – collision resolution: Cuckoo hashing



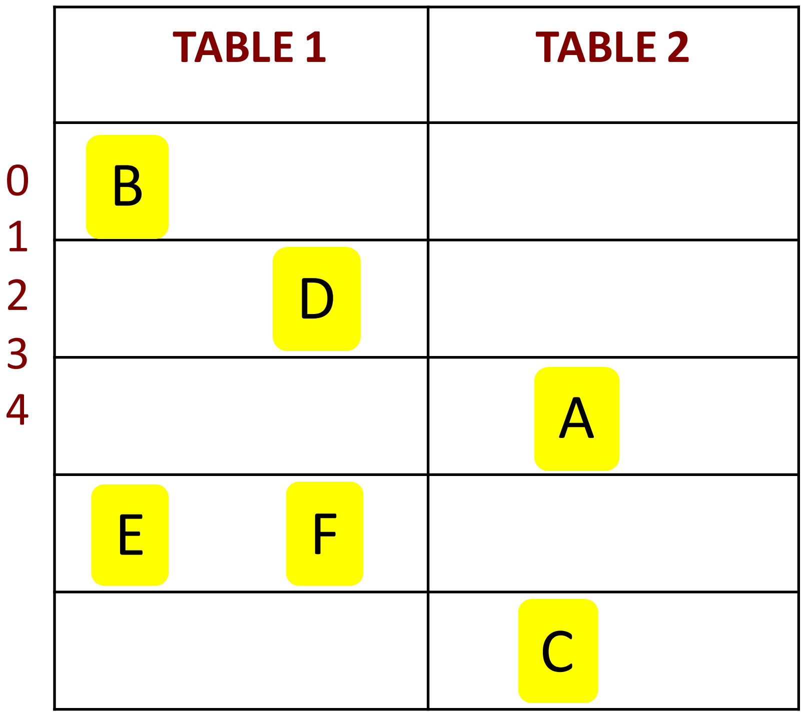
A: $hash_1(A) = 0$, $hash_2(A) = 2$

B: $hash_1(B) = 0$, $hash_2(B) = 0$

C: $hash_1(C) = 1$, $hash_2(C) = 4$

D: $hash_1(D) = 1$, $hash_2(D) = 0$

Hash tables – collision resolution: Cuckoo hashing



A: $hash_1(A) = 0$, $hash_2(A) = 2$

B: $hash_1(B) = 0$, $hash_2(B) = 0$

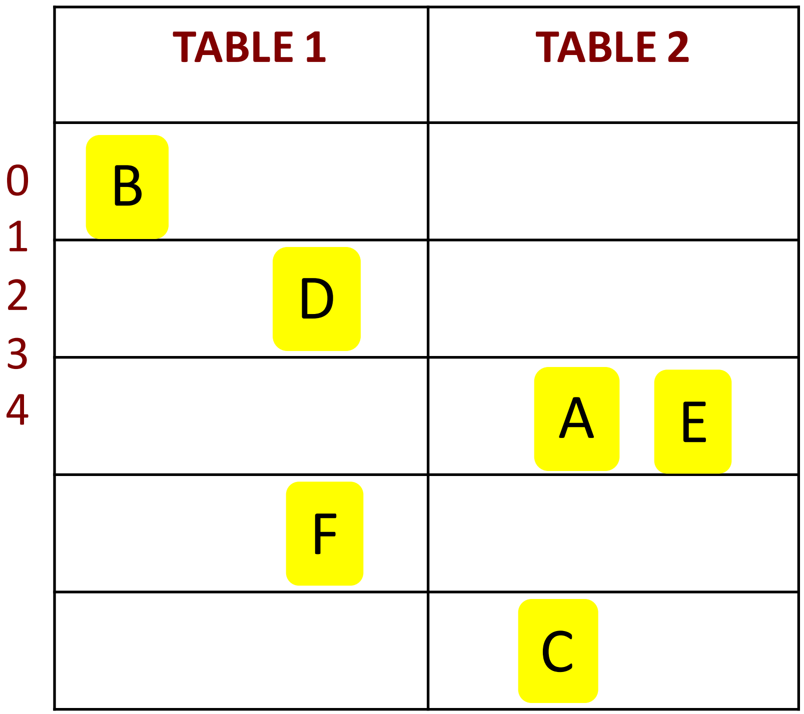
C: $hash_1(C) = 1$, $hash_2(C) = 4$

D: $hash_1(D) = 1$, $hash_2(D) = 0$

E: $hash_1(E) = 3$, $hash_2(E) = 2$

F: $hash_1(F) = 3$, $hash_2(F) = 4$

Hash tables – collision resolution: Cuckoo hashing



A: $hash_1(A) = 0$, $hash_2(A) = 2$

B: $hash_1(B) = 0$, $hash_2(B) = 0$

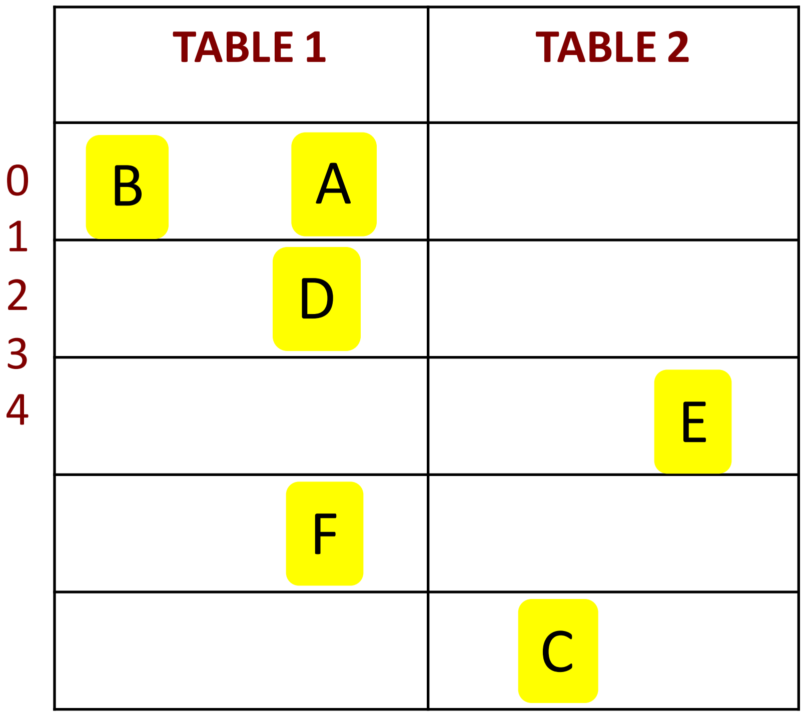
C: $hash_1(C) = 1$, $hash_2(C) = 4$

D: $hash_1(D) = 1$, $hash_2(D) = 0$

E: $hash_1(E) = 3$, $hash_2(E) = 2$

F: $hash_1(F) = 3$, $hash_2(F) = 4$

Hash tables – collision resolution: Cuckoo hashing



A: $hash_1(A) = 0$, $hash_2(A) = 2$

B: $hash_1(B) = 0$, $hash_2(B) = 0$

C: $hash_1(C) = 1$, $hash_2(C) = 4$

D: $hash_1(D) = 1$, $hash_2(D) = 0$

E: $hash_1(E) = 3$, $hash_2(E) = 2$

F: $hash_1(F) = 3$, $hash_2(F) = 4$

Hash tables – collision resolution: Cuckoo hashing

	TABLE 1	TABLE 2
0	A	B
1	D	
2		E
3	F	
4		C

A: $hash_1(A) = 0$, $hash_2(A) = 2$

B: $hash_1(B) = 0$, $hash_2(B) = 0$

C: $hash_1(C) = 1$, $hash_2(C) = 4$

D: $hash_1(D) = 1$, $hash_2(D) = 0$

E: $hash_1(E) = 3$, $hash_2(E) = 2$

F: $hash_1(F) = 3$, $hash_2(F) = 4$

Hash tables – collision resolution: Cuckoo hashing

Insert

- Insert into Table 1, using $hash_1$
- If cell is already occupied
  - bump item into other table (using appropriate hash function)
  - Repeat
- Rehash after k repetitions

Each table should be more than half empty

Stronger condition than load factor ≤ ½

Rehashing

When load factor becomes too large…

(Approximately) double tableSize

Scan old table, inserting each non-deleted item into the new table

Worst-case time?

- $O(N^2)$

Average-case: $O(N)$

Amortized analysis

Average cost per insert, over a sequence of repeated re-hashings

[Not great for interactive applications…]

Hash tables - review

Supports the basic dynamic dictionary ops: insert, find, remove

Three design decisions: tableSize, hash function, collision resolution

Table size: a prime of the form (4k+3), keeping load factor constraints in mind

Hash function

Java's hashCode() method

item goes to hash(item) % tableSize

Collision: multiple items at the same location

Collision resolution:-chaining

-probing (open addressing)

- Linear probing
- Quadratic probing
- Double Hashing
- Cuckoo Hashing

Java-specific – hashCode() and equals()

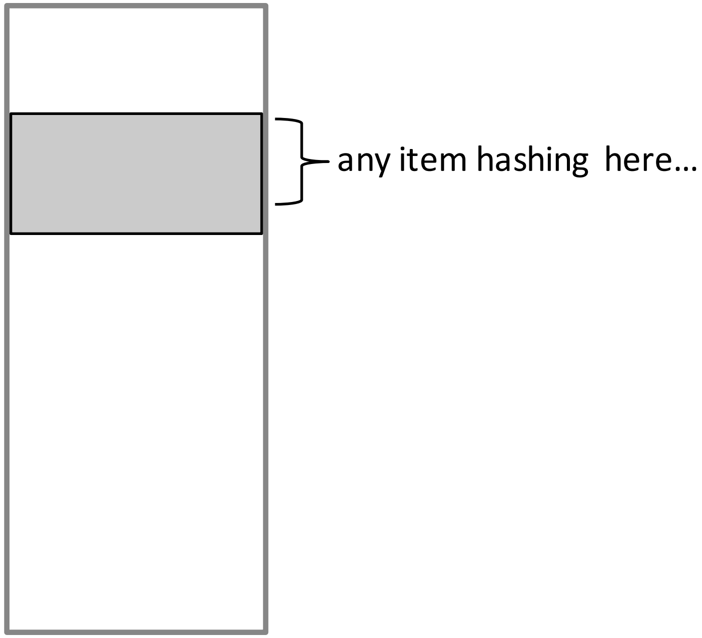
```
public class Employee {
        String name;
        int id;
        public Employee(String n, int i){name = n; id = i;}

                                        …
                                        …
}

public static void main(String[] args) {
        Employee e1=new Employee("weiss", 001);
        Employee e2 = e1;
        System.out.println(e1.hashCode() + ", " + e2.hashCode());
        System.out.println(e1 == e2);
        System.out.println(e1.equals(e2));
```

Hash tables – collision resolution: linear probing

f(i) can be any linear function (a * i + b)

If gcd(a, tableSize) = 1, then linear probing will probe the entire table

Primary clustering: blocks of occupied cells start forming even in a relatively empty table

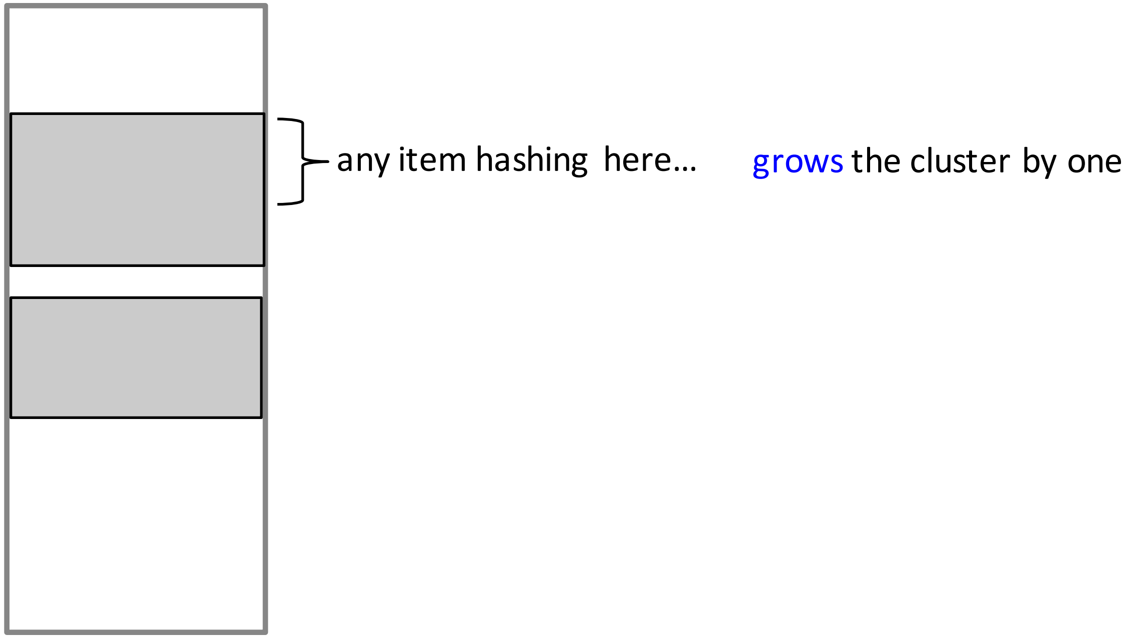


Hash tables – collision resolution: linear probing

f(i) can be any linear function (a * i + b)

If gcd(a, tableSize) = 1, then linear probing will probe the entire table

Primary clustering: blocks of occupied cells start forming even in a relatively empty table

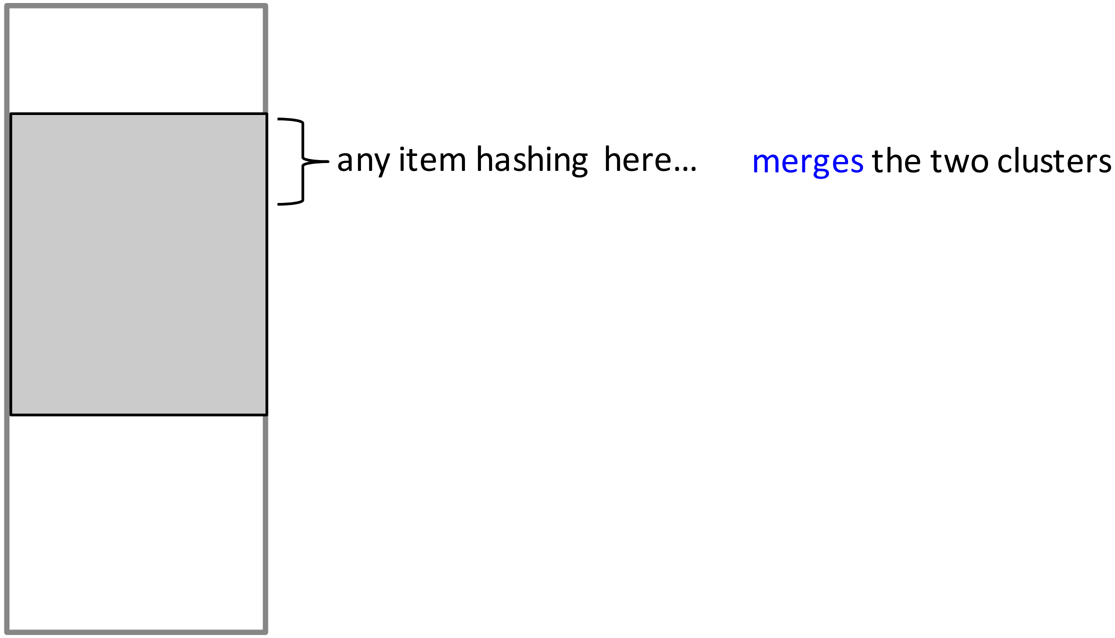


Hash tables – collision resolution: linear probing

f(i) can be any linear function (a * i + b)

If gcd(a, tableSize) = 1, then linear probing will probe the entire table

Primary clustering: blocks of occupied cells start forming even in a relatively empty table



Hash tables - clustering

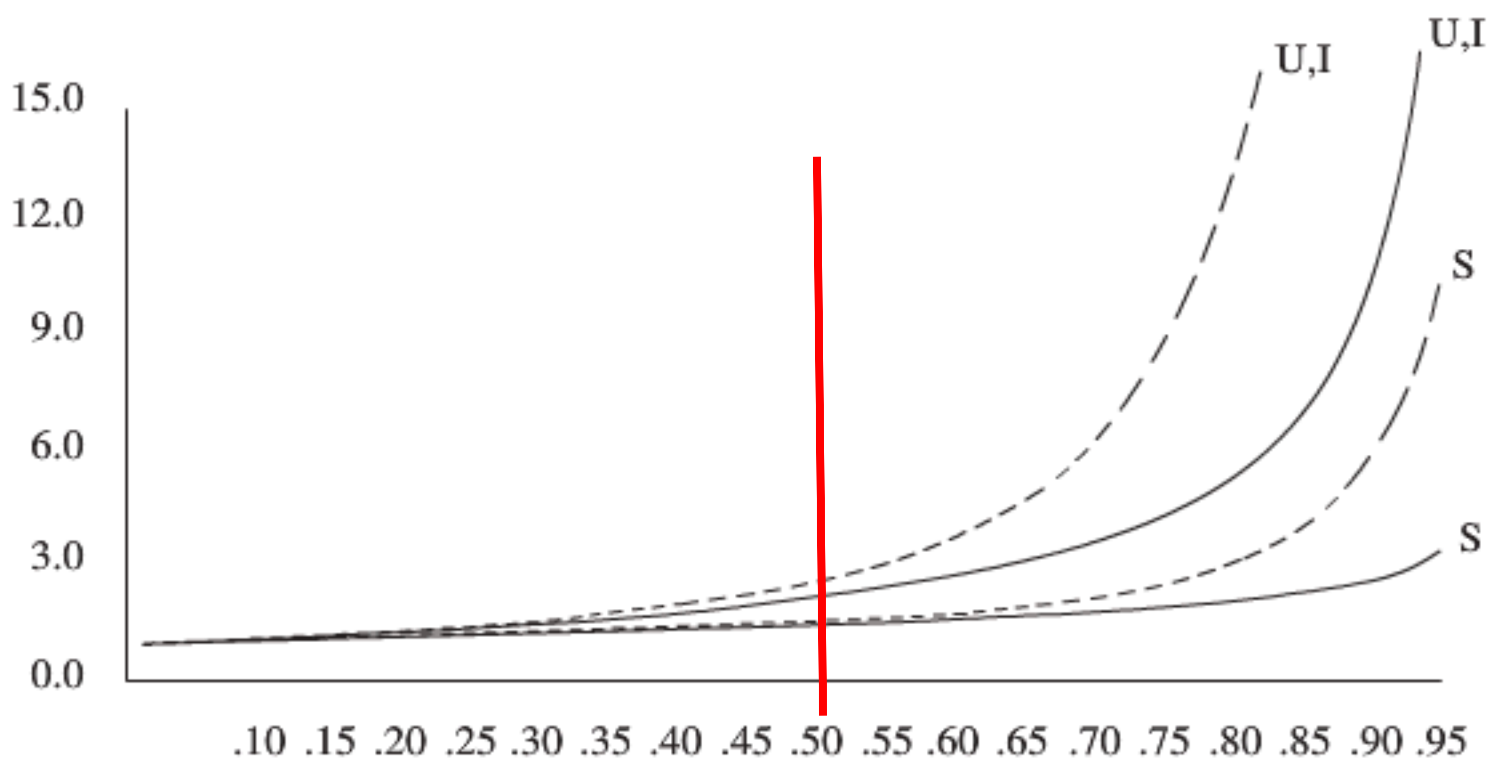
Two causes of clustering:

multiple keys hash on to the same location (secondary clustering)

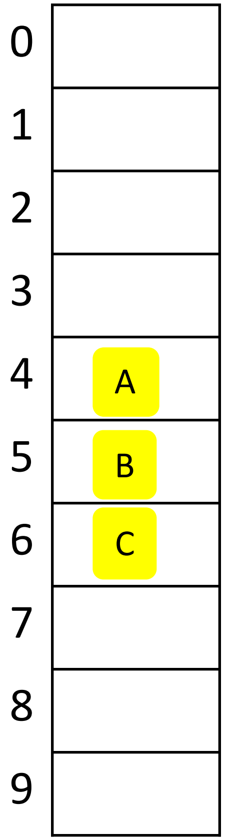
multiple keys hash on to the same cluster (primary clustering)

Secondary clustering caused by hash function; primary, by choice of probe sequence

Number of probes per operation increases with load factor



Hash tables – linear probing: remove



insert A; hash(A) = 4

insert B; hash(B) = 5

insert C; hash(C) = 4

remove B

find C

Remove must be implemented as lazy delete!!

- Load factor computed including lazy-deleted items
- In inserts, may “reclaim” lazy-deleted cells