

# Lecture 7

## Hash Tables

Zonghua Gu (based on Jianchen Shan)  
Department of Computer Science  
Hofstra University

# Lecture Goals

- Describe why hash tables are valuable
- Describe the role of a **hash function** and the **hash code**
- Describe Java's Hash Code **Conventions**
- Describe Java's **implementations** of hash code
- Describe alternative methods for **handling collisions** in a Hash Table
- Identify other **challenges** associated with Hash Tables
- Explain the difference between a **Hash Set**, **Hash Map** and **Hash Table**

# Motivation

Find Ada

Option 1: brute force linear search

Take long time for big array

What if we happen to know the index of the value?

Ada -> 8 ? `myData = Array[8]`

Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe
0	1	2	3	4	5	6	7	8	9	10

Very fast

If you know where in memory the array starts, you can easily determine the address of any element using the index. Accessing an address is an  $O(1)$  operation, and independent of array size.

How can you know which elements of the array contains the value you are looking for?

Each index number can be calculated using the value itself. So the index number is in some way related to the data

Option 2: hash table

# Hash Table

Save items in a **key-indexed table** (index is a function of the key).

**Hash function** is the method for computing array index from key.

Let's repopulate the array to be a hash table with following hash function:

**Index number** = sum Unicodes Mod array size

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10

key # of elements in array

K mod N is a common hash function

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

# Hash Table (Contd.)

**Index number** = sum Unicodes Mod array size

Find Ada

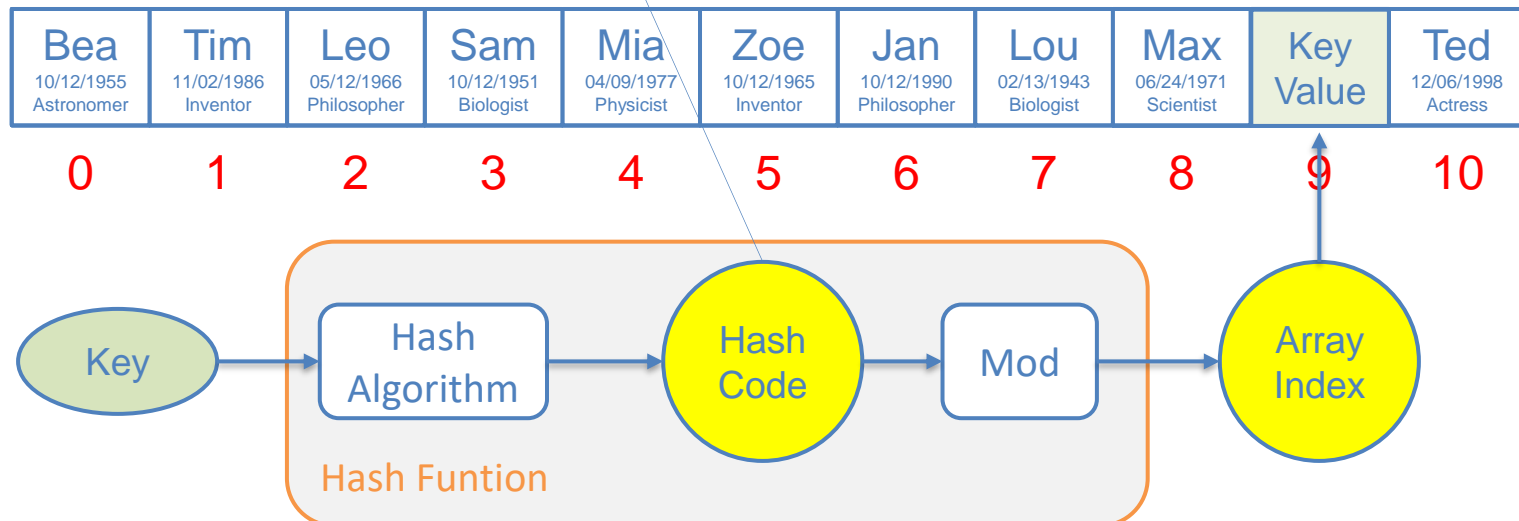
Hash tables are often used to store <key, value> pairs, which can be the objects in java. Key is just one of the object's property

Ada = (65 + 100 + 97) = 262 262 Mod 11 = 9

myData = Array[9]

Ada

03/27/1969  
Inventor



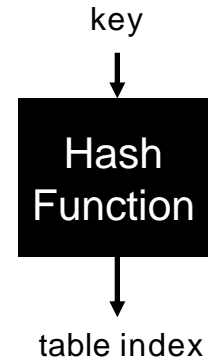
# Hash Function

Calculation applied to a key to transform it into an address (array/table index).

Idealistic goal: Scramble the keys uniformly to produce a table index.

Efficiently computable.

Each table index equally likely for each key.



Ex. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

But actually, in most cases, you want  
to find the way to use all the data

Practical challenge: need different approach for each key type.

- For **numeric keys**, divide the key by the number of available addresses,  $n$ , and take the remainder.

$$\text{address} = \text{key} \text{ Mod } n$$

- For **alphanumeric keys**, divide the sum of Unicodes in a key by the number of available addresses,  $n$ , and take the remainder.
- **Folding method** divides key into equal parts then adds the parts together
  - The telephone number 5164635712 becomes  $51+64+63+57+12 = 247$
  - Depending on size of table, may then divide by some constant and take remainder
  - ensures that all the digits contribute to the hash code

# Java's Hash Code Conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit int.

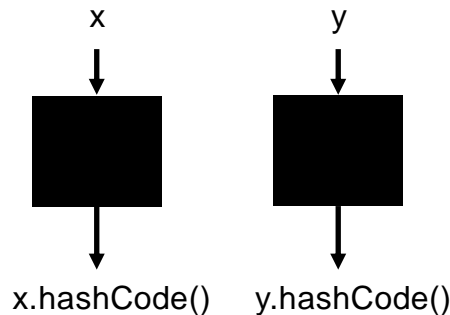
Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

`==` tests for reference equality (whether they are the same object).

`.equals()` tests for value equality (whether they are logically "equal").

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.

collision



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.

Meets the two requirements for Java. But it doesn't meet the idea that every table position should be **equally likely** mapped from the keys.

- **Default implementation.** Memory address of `x`.
- **Legal (but poor) implementation.** Always return 17. collision
- **Customized implementations.** Integer, Double, String, File, URL, Date, ...
- **User-defined types.** Users are on their own.

# Implementing Hash Code: Integers, Booleans

## Java library implementation

```
public final class Integer {  
    private final int value;  
    ...  
    public int hashCode() {  
        return value;  
    }  
}
```

```
public final class Boolean {  
    private final boolean value;  
    ...  
    public int hashCode() {  
        if (value) return 1231;  
        else      return 1237;  
    }  
}
```

### Two large prime numbers

1. **avoid collision**  
(e.g., better than 1000 and 2000.)

2. **larger impact** on the hash code of a composite object (e.g., better than 0 and 1)

$1000 \bmod 8 = 2000 \bmod 8$   
 $1000 \bmod 10 = 2000 \bmod 10$   
 $1000 \bmod 20 = 2000 \bmod 20$

```
class InterviewCandidate {  
    String candidateName;  
    Boolean isSelected;  
}
```

To write the hashcode for this class, typically you will find hashcode for candidateName, hashcode for isSelected, multiply them with some prime number and then add them up



# Implementing Hash Code: Doubles

## Java library implementation

```
public final class Double {
    private final double value;
    ...
    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

1. convert to IEEE 64-bit representation.

2. xor most significant 32-bits with least significant 32-bits.

XOR is a binary operation, it stands for "exclusive or", that is to say the resulting bit evaluates to one if only exactly *one* of the bits is set.

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

**10.24**    01000000 00100100 01111010 11100001    01000111 10101110 00010100 01111011

XOR    01000000 00100100 01111010 11100001

01000111 10101110 00010100 01111011

00000111 10001010 01101110 10011010

- return 126512794
- all the digits contribute to the hash code

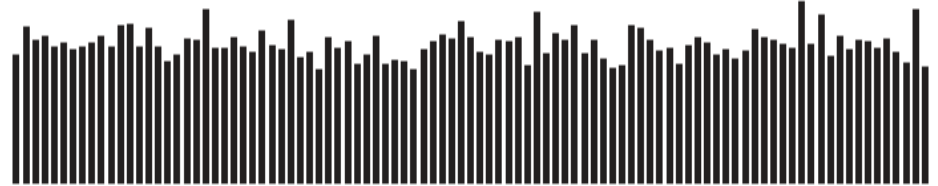
# Implementing Hash Code: Strings

## Java library implementation

```
public final class String {
    private final char[] s;
    ...

    public int hashCode() {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

←  $i$ th character of s



Hash value frequencies for words in Tale of Two Cities ( $M = 97$ )

Java's Stringdata uniformly distribute the keys of Tale of Two Cities

'b'	98
'c'	99
...	...

- Horner's method to hash string of length  $n$ :  $n$  multiplies/adds.
- Equivalent to  $h = s[0] \cdot 31^{n-1} + \dots + s[n-3] \cdot 31^2 + s[n-2] \cdot 31^1 + s[n-1] \cdot 31^0$ .

Ex. `String s = "call";`  
`int code = s.hashCode();`

$$\begin{aligned}
 3045982 &= 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \\
 &= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))
 \end{aligned}$$

(Horner's method)

- return 3045982
- It involves all the characters of the string in computing the hash function.

# Implementing Hash Code: Strings (Contd.)

## Java library implementation

```
public final class String {  
    private final char[] s;  
    private int hash = 0;  ← cache of hash code  
    ...  
    public int hashCode() {  
        int h = hash;  
        if (h != 0) return h;  ← return cached value  
        for (int i = 0; i < length(); i++)  
            h = s[i] + (31 * h);  
        hash = h;  ← store cache of hash code  
        return h;  
    }  
}
```

- Performance optimization.
  - Cache the hash value in an instance variable.
  - Return cached value.

# Implementing Hash Code: User-defined Types

## Java library implementation

```
public final class Transaction {
    private final String who;
    private final Date when;
    private final double amount;
```

```
    public int hashCode() {
        int hash = 17;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

nonzero constant

for reference types,  
use hashCode()

for primitive types,  
use hashCode() of  
wrapper type

typically a small prime

- 31 is a prime number. The product of a prime with any other number has the best chance of being unique. The value was chosen for better distribution.
- A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance:  $31 * i == (i \ll 5) - i$ .
- Using primes is an old technique.

"Standard" recipe for user-defined types. (works well and used in java libraries)

- Combine each significant field using the  $31x + y$  rule.
- If field is a primitive type, use wrapper type hashCode().
- If field is null, return 0.
- If field is a reference type, use hashCode(). applies rule recursively
- If field is an array, apply to each entry. or use `Arrays.deepHashCode()`

**Basic rule.** Need to use the whole key to compute hash code;

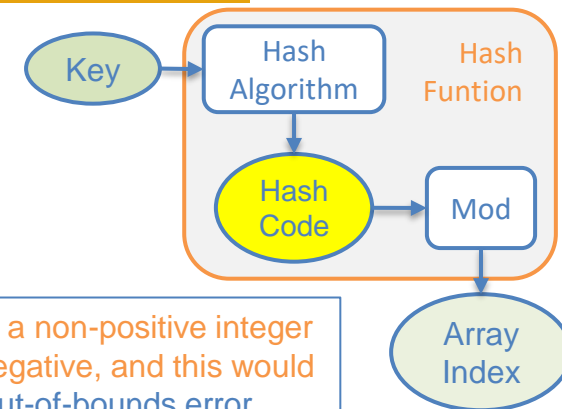
# Modular Hashing

**Hash code:** an int between  $-2^{31}$  and  $2^{31} - 1$ .

M is the size of the array, which is typically a prime or power of 2

**Hash function:** an int between 0 and  $M - 1$  (for use as array index).

Since our goal is an array index, not a 32-bit integer, we combine `hashCode()` with modular hashing produce integers between 0 and  $M-1$ , which is used as an array/table index.



```
private int hash(Key key) {
    return key.hashCode() % M;
}
```

**bug**

The % operator returns a non-positive integer if its first argument is negative, and this would create an array index out-of-bounds error.

```
private int hash(Key key) {
    return Math.abs(key.hashCode()) % M;
}
```

**1-in-a-billion bug**

the absolute value of `Integer.MIN_VALUE` is itself!  
Famously, `hashCode()` of "polygenelubricants" is  $-2^{31}$

```
private int hash(Key key) {
    return (key.hashCode() & 0x7fffffff) % M;
}
```

**correct**

The code masks off the sign bit (to turn the 32-bit integer into a 31-bit nonnegative integer) and then computing the remainder when dividing by M

# Absolute value of Integer.MIN\_VALUE is itself

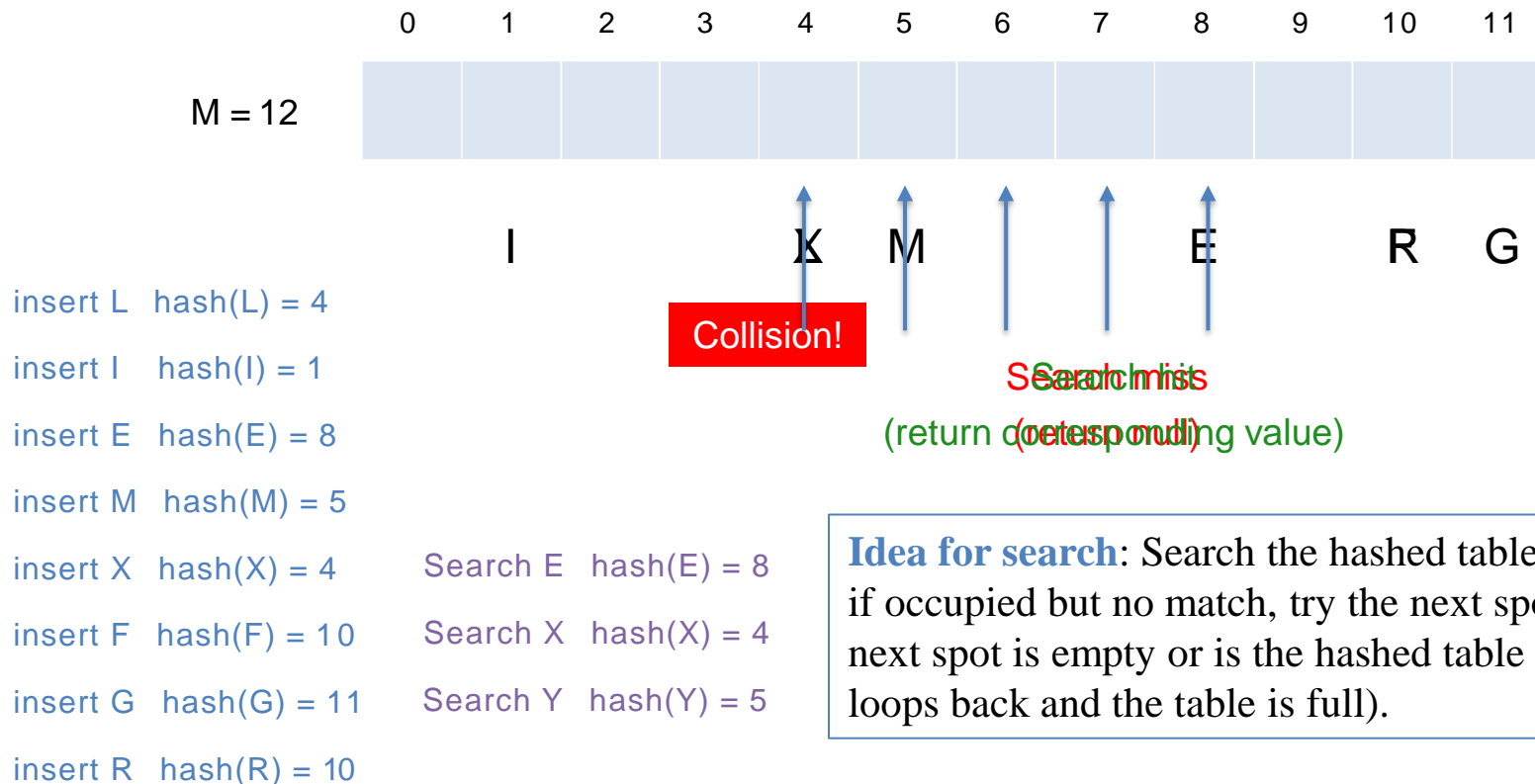
- Integer representation: In Java, integers are stored using 32 bits in two's complement format.
- Range of int: The range of int in Java is from  $-2^{31}$  to  $2^{31} - 1$ , which is -2,147,483,648 to 2,147,483,647.
- Integer.MIN\_VALUE: This constant represents the minimum value an int can hold, which is -2,147,483,648.
- No positive counterpart: There is no positive 32-bit integer that can represent 2,147,483,648 (which would be the absolute value of -2,147,483,648).
- The Math.abs() Function
  - When you try to get the absolute value of Integer.MIN\_VALUE using Math.abs(), here's what happens:
    - `int minValue = Integer.MIN_VALUE;`
    - `int absValue = Math.abs(minValue);`
    - `System.out.println(minValue == absValue); // This prints true`

# Collision and Resolution: Open Addressing

**Collision:** two distinct keys hashing to same index.

**Solution:** Linear probing

**Idea:** Just put it in the next open spot



**Idea for search:** Search the hashed table index, and if occupied but no match, try the next spot until the next spot is empty or is the hashed table index (i.e., loops back and the table is full).

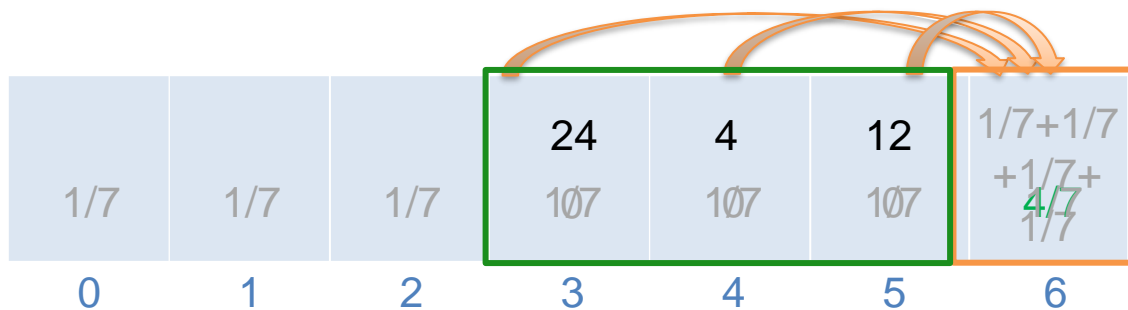
# Linear Probing: Primary Clustering

What is the probability of next key going in each slot?

$$\text{Hash}(k) = k \bmod 7$$

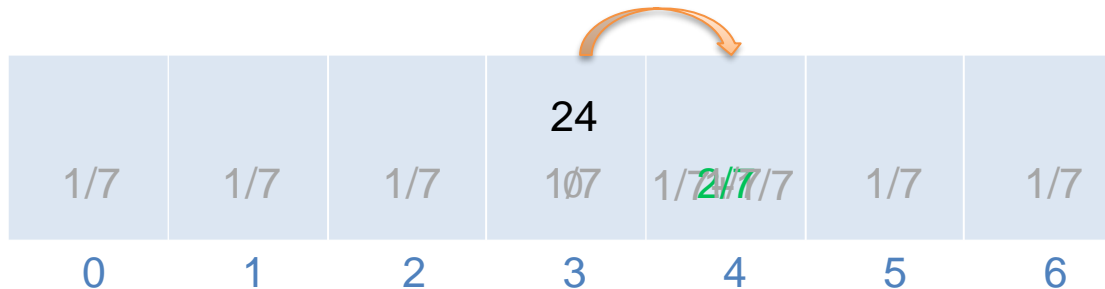
All keys equally likely

Cluster is a contiguous block of items.



Observation. New keys likely to hash into middle of big clusters.

Higher insert and search costs -  $O(n)$





# Linear Probing: Primary Clustering (Contd.)

Three methods to mitigate the problem

1. Better-designed hash function

3. Resize the hash table when it's "full"

2. Alternative probing methods

$$\text{Load factor} = \frac{\text{Total number of items stored}}{\text{Size of the array}}$$

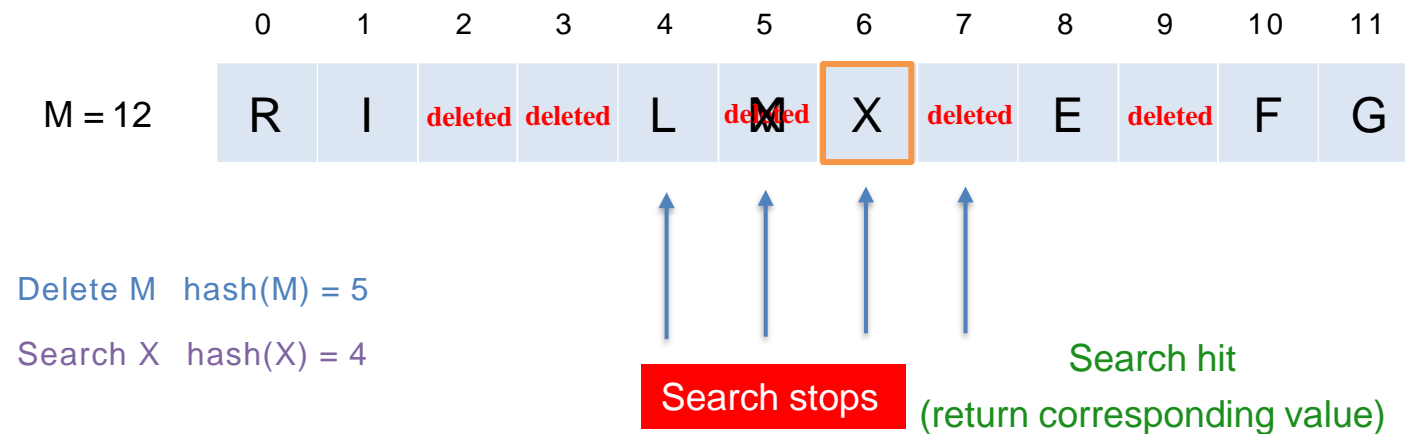
## Open addressing

- linear probing
  - try next open spot
- Plus 3 rehash
  - skip 3 spots, not just next open spot
- Quadratic probing
  - skip (failed attempts)<sup>2</sup> spots
- Double hashing
  - apply a second hash function to key when a collision occurs

Repopulate the items into a larger array, when load factor > 70%

# Linear Probing: Delete

How to delete item from a hash table?



**Method 1:** mark the spot as “empty but deleted”. Probing is continued when encountering such spot. An add operation can store data in such spot.

**Table pollution issue:** These deleted flags may bridge together otherwise unrelated data (different hashcodes). In the worst case, search is linear time.

The only solution is to repopulate the key-value pairs into a new table, and discard the old one.

**Method 2:**

1. Find and remove the desired element
2. Go to the next spot
3. If the spot is empty, quit
4. If the spot is full, delete the element in that spot and re-add it to the hash table using the normal means. The item must be removed before re-adding, because it is likely that the item could be added back into its original spot.
5. Repeat step 2.

This technique keeps your table tidy at the expense of slightly slower deletions.

# Closed Addressing: Separate Chaining

Use an array of M linked lists.

Insert: put in the end of hashed chain

Search: need to search only hashed chain

M = 12

insert L hash(L) = 4

insert I hash(I) = 1

insert E hash(E) = 8

insert M hash(M) = 5

insert X hash(X) = 4

insert F hash(F) = 10

insert G hash(G) = 11

insert R hash(R) = 10

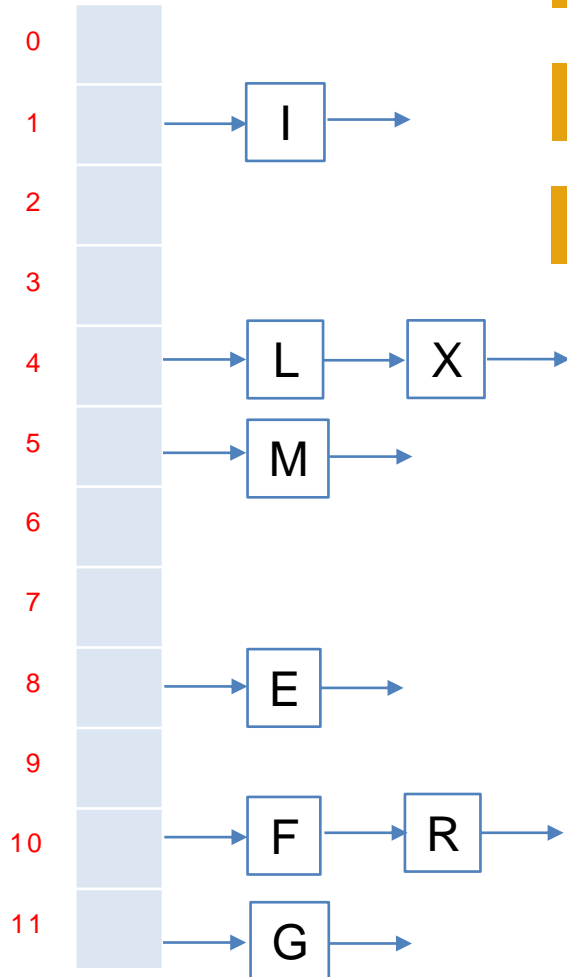
The size of the array can be smaller than the stored number.

**Variant:** Hash to two positions, insert key in shorter of the two chains.

Search E hash(E) = 8

Search X hash(X) = 4

Search Y hash(Y) = 5



# Separate Chaining vs Linear Probing

- **Separate chaining.**

- Easier to implement delete.
- Performance degrades gracefully.
- less sensitive to poorly-designed hash function

“With a bad choice of hash function, **primary clustering** can cause the performance of the table to degrade significantly. While chaining can still suffer from bad hash functions, it's less sensitive to elements with nearby hash codes, which don't adversely impact the runtime.”

- **Linear probing.**

- Less wasted space.
- Better cache performance.

“**In practice**, linear probing is typically significantly faster than chaining due to locality of reference, although it has the primary clustering problem. It's faster to access a series of elements in an array than it is to follow pointers in a linked list, so linear probing tends to outperform chaining even if it has to investigate more elements. Another win in chaining is that the insertions into a linear probing hash table don't require any new allocations.”

# Hashtable vs. HashMap vs. HashSet in Java

- Hashtable and HashMap both implement **Map** interface.
  - A Map is a mapping of key-value pairs.
- HashSet implements **Set** interface.
  - A Set is a collection of objects.
- Hashtable is a **legacy** class
  - almost always should be avoided in favor of HashMap
- HashSet has **nothing** to do with HashMap.
  - It happens to use a HashMap internally to implement the Set functionality
- HashSet does not allow **duplicate** values.
  - HashMap allows duplicate values.
  - provides add method rather put method

```
public static void main(String args[]) {  
    HashMap<Integer,String> hm=new HashMap<Integer,String>();  
    hm.put(100,"Amit");  
    hm.put(104,"Amit"); // allows duplication  
    hm.put(101,"Vijay");  
    hm.put(102,"Rahul");  
    for (Map.Entry m:hm.entrySet()) {  
        System.out.println(m.getKey()+" "+m.getValue());  
    }  
}
```

```
100 Amit  
101 Vijay  
102 Rahul  
104 Amit
```

```
HashSet<String> mySet= new HashSet<String>();  
mySet.add ("First");  
mySet.add ("Second");  
mySet.add ("Third");  
  
if(mySet.contains("First")){  
    System.out.println("The Set already contains First");  
}
```

# Additional Resources

- Hashtables and HashMaps
  - [http://www.tutorialspoint.com/data\\_structures\\_algorithms/hash\\_data\\_structure.htm](http://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm) -- about hashing, linear probing, search, delete, insert
  - <https://www.youtube.com/watch?v=MfhjkfocRR0> -- video about hashtable
  - <http://stackoverflow.com/questions/40471/differences-between-hashmap-and-hashtable> -- difference between hashtables and hashmaps