# Lecture 7
# Hash Tables

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

# 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?

Option 2: hash table

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

# 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 Unicode of each character) mod array size (11)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

| char | Unicode |
|---|---|
| … | … |
| 'a' | 97 |
| 'b' | 98 |
| 'c' | 99 |
| … | … |

key # of elements in array

K mod N is a common hash function

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

# Hash Table (Contd.)

Index number = sum Unicodes mod array size

Find Ada

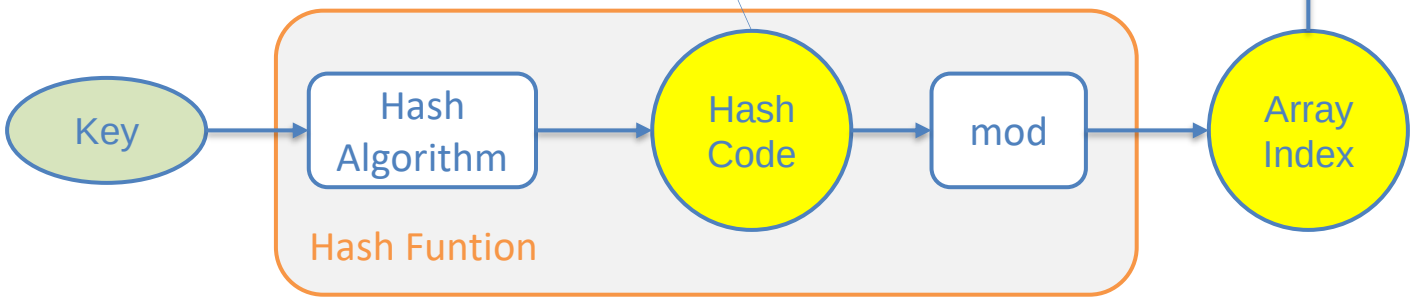
Hash tables are often used to store <key, value> pairs, which can be the objects in java. Key is just one of the object's property

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

myData = Array[9]

Ada
03/27/1969
Inventor

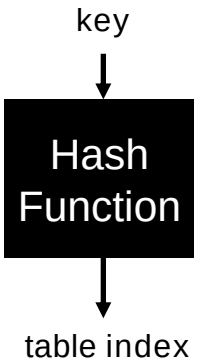
| Bea | Tim | Leo | Sam | Mia | Zoe | Jan | Lou | Max | Key Value | Ted |
|---|---|---|---|---|---|---|---|---|---|---|
| 10/12/1955 Astronomer | 11/02/1986 Inventor | 05/12/1966 Philosopher | 10/12/1951 Biologist | 04/09/1977 Physicist | 10/12/1965 Inventor | 10/12/1990 Philosopher | 02/13/1943 Biologist | 06/24/1971 Scientist | | 12/06/1998 Actress |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |



# Hash Function

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

Ideal 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.

Preferably, you want to use all the data for computing the hash code

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.

$$address = key \bmod n$$

- 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

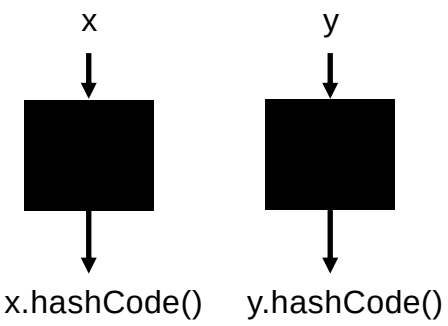
All Java classes inherit a method hashCode(), which returns a 32-bit int.

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 slot 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 1231 and 1237 are used as hashCode of a Boolean value of true or false, respectively

1. avoid collision
(e.g., better than even numbers like 1000 and 2000.)

1000 mod 8 = 2000 mod 8
1000 mod 10 = 2000 mod 10
1000 mod 20 = 2000 mod 20

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

```
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 XOR b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

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

**01000000 00100100 01111010 11100001**

XOR **01000111 10101110 00010100 01111011**

---

**00000111 10001010 01101110 10011010**

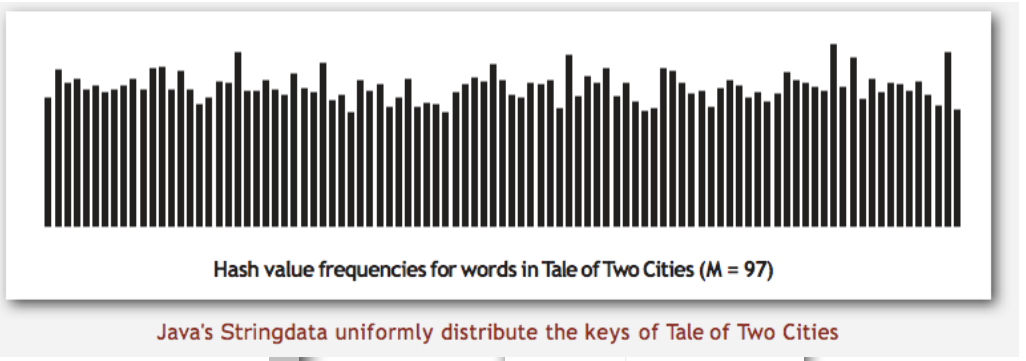
- return 126512794
- it involves all digits in the number for computing the hash code.

# 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;
     }
}
```

ith character of s




| | |
|---|---|
| 'b' | 98 |
| 'c' | 99 |
| ... | ... |

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

Ex.

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

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

(Horner's method)

- return 3045982
- it involves all characters in the string for computing the hash code.

# Implementing Hash Code: Strings (Contd.)

**Java library implementation**

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

cache of hash code

return cached value

ache the hash code for future use

- Performance optimization.
  - Cache the hash value in an instance variable.
  - Return cached value if already computed before, to avoid redundant computation.

# 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

typically a small prime

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

- 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.
- Multiplication by 31 can be replaced by a shift and a subtraction for better performance: $31 * i == 2^5 * i - i = (i << 5) - i$.

"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;

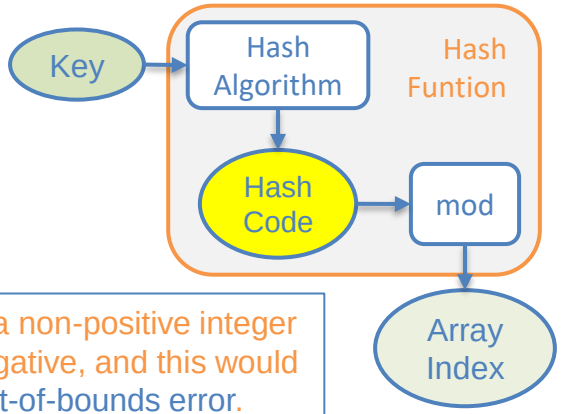
# Modulo Hashing

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 the modulo M operator (% M) to produce an integer between 0 and M-1, which is used as an array index into an array of size M.




```
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 highest sign bit (to turn the 32-bit integer into a 31-bit nonnegative integer) and then computes the remainder when dividing by M. (7 in binary is 0111, and f in binary is 1111.)

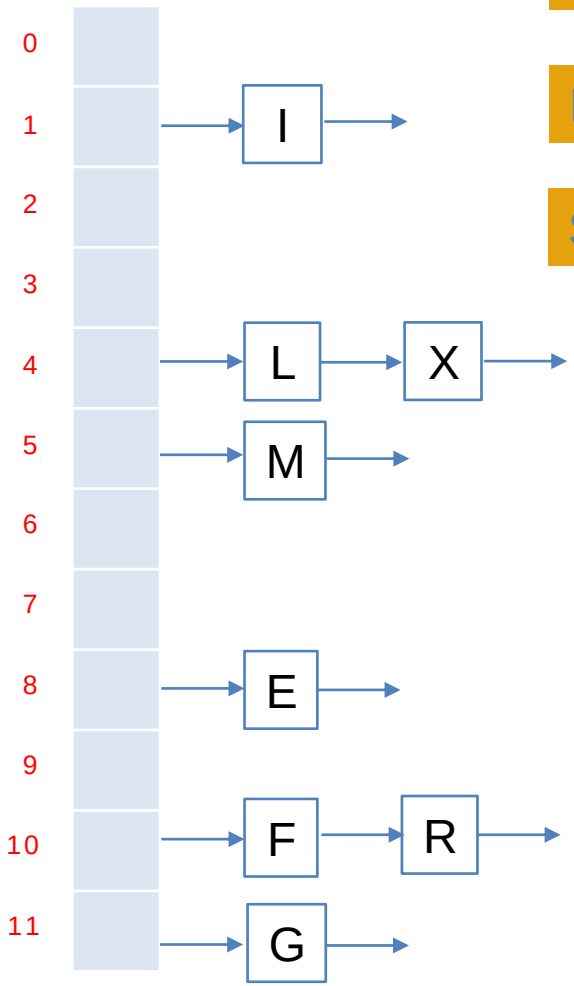
# Absolute value of Integer.MIN_VALUE is itself

- 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

# Handling Hash Collisions

- Two types of techniques for handling hash collisions:
- 1. Separate Chaining (Closed Addressing):
  - Each slot in the hash table holds a reference to a linked list that stores all elements hashing to the same index.
- 2. Open Addressing: resolve collisions by finding another open slot within the hash table itself. It includes:
  - Linear Probing: Sequentially checks the next available slots.
  - Quadratic Probing: Uses a quadratic function to determine the next slot to check, i.e., if the primary hash index is x, probes slots $x+1^2$, $x+2^2$, $x+3^2$, $x+4^2$ and so on, until an empty slot is found.
  - Double Hashing: First apply a primary hash function h1(k). If collision, then apply a second hash function h2(k) to calculate the probe sequence, e.g., Probe(k, i)=(h1(k)+i·h2(k))%M, i=0, 1, 2…that is:
    - primary slot: h1(k), if it is occupied then
    - attempt 1: h1(k) + h2(k), if that is occupied then
    - attempt 2: h1(k) + 2·h2(k), if that is occupied then
    - attempt 3: h1(k) + 3·h2(k), and so on

# Separate Chaining




Use an array of M linked lists.

Insert: put in the end of hashed chain (or in sorted order)

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(k) = 4

insert F hash(F) = 10

insert G hash(G) = 11

insert R hash(R) = 10

Search E hash(E) = 8

Search X hash(k) = 4

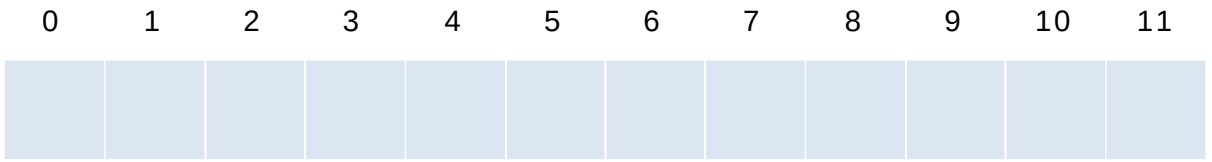
Search Y hash(Y) = 5

# Open Addressing Example 1 : Linear Probing

Collision: two distinct keys hashing to same index.

Solution: Linear probing

Idea: Just put it in the next open slot




insert L hash(L) = 4

insert I hash(I) = 1

insert E hash(E) = 8

insert M hash(M) = 5

insert X hash(k) = 4

insert F hash(F) = 10

insert G hash(G) = 11

insert R hash(R) = 10

Collision!

Search hit

(return corresponding value)

Search E hash(E) = 8

Search X hash(k) = 4

Search Y hash(Y) = 5

**Linear probing**: Check the hash table index, and if occupied but no match, try the next slot until the next slot is empty or is the hashed table index (i.e., wrap around when the table is full).

# Open Addressing Example 2: Linear Probing

- Consider the "mod 5" hash function, and a sequence of keys (hashcodes) that are to be inserted in order: 50, 70, 76, 85, 93.
- Step 1. Start with an empty hash table of size 5.
- Step 2. The first key to be inserted is 50, which is mapped to slot 0 (50%5=0).
- Step 3. The next key is 70, which is mapped to slot 0 (70%5=0) but 50 is already at slot 0, so, search for the next empty slot and insert it into slot 1.
- Step 4. The next key is 76 which is mapped to slot 1 (76%5=1) but 70 is already at slot 1, so search for the next empty slot and insert it into slot 2.
- Step 5. The next key is 85 which is mapped to slot 0 (85%5=0), but 50 is already at slot number 0, so search for the next empty slot and insert it into slot 3.
- Step 6. The next key is 93, which is mapped to slot 3 (93%5=3), but 85 is already at slot 3, so search for the next empty slot and insert it into slot 4.

Step 1

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | | | | |

Step 2

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 50 | | | | |

Step 3

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 50 | 70 | | | |

Step 4

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 50 | 70 | 76 | | |

Step 5

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 50 | 70 | 76 | 85 | |

Step 6

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 50 | 70 | 76 | 85 | 93 |

# Open Addressing Example 3: Quadratic Probing

- Consider the "mod 7" hash function, and a sequence of keys (hashcodes) that are to be inserted in order: 22, 30, 50, 57.
- Step 1. Start with an empty hash table of size 7.
- Step 2. The first key to be inserted is 22 which is mapped to slot 1 (22%7=1)
- Step 3. The next key is 30 which is mapped to slot 2 (30%7=2)
- Step 4. The next key is 50, which is mapped to slot 1 (50%7=1), but slot 1 is already occupied. So, we will search slot 1+1^2=1+1 = 2. Again, slot 2 is occupied, so we will search slot 1+2^2=1+4 = 5, which is empty, so insert it into slot 5.
- Step 5. The next key is 57, which is mapped to slot 1 (57%7=1), but slot 1 is already occupied. So, we will search slot 1+1^2=1+1 = 2. Again, slot 2 is occupied, so we will search slot 1+2^2=1+4 = 5, but slot 5 is occupied. So, we will search slot 1+3^2=1+9 =10. But there are only 7 slots, so we wrap around and search slot (1+3^2)%7=3, which is empty, so insert it into slot 3.

Step 1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |

Step 2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 22 | | | | | |

Step 3

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 22 | 30 | | | | |

Step 4

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 22 | 30 | | | 50 | |

Step 5

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 22 | 30 | 57 | | 50 | |

# Open Addressing Example 4: Double Hashing

- Double hashing with two hash functions:
  - h1(k)=x%7 (primary hash)
  - h2(k)=1+(x%5) (secondary hash, ensuring steps ≠ 0)
- Probing formula:
  - Probe(k, i)=(h1(k)+i·h2(k))%7
- Consider a sequence of keys (hashcodes) that are to be inserted in order: 16, 23, 30, 9, 2, 37.
- Step 1. Start with an empty hash table of size 7.
- Step 2. The first key is 16. h1(16)=16%7=2. Slot 2 is empty → insert 16.
- Step 3. The next key is 23. h1(23)=23%7=2 (collision at slot 2). h2(23)=1+(23%5)=1+3=4. Probe 1: (2+1·4)%7=6. Slot 6 is empty → insert 23.
- Step 4. The next key is 30. h1(30)=30%7=2 (collision at slot 2). h2(30)=1+(30%5)=1+0=1. Probe 1: (2+1·1)%7=3. Slot 3 is empty → insert 30.
- Step 5. The next key is 9. h1(9)=9%7=2 (collision at slot 2). h2(9)=1+(9%5)=1+4=5. Probe 1: (2+1·5)%7=0. Slot 0 is empty → insert 9.
- Step 6. The next key is 2. h1(2)=2%7=2 (collision at slot 2). h2(2)=1+(2%5)=1+2=3. Probe 1: (2+1·3)%7=5. Slot 5 is empty → insert 2.
- Step 6. The next key is 37. h1(2)=37%7=2 (collision at slot 2). h2(37)=1+(37%5)=1+2=3. Probe 1: (2+1·3)%7=5. Slot 5 is occupied. Probe 2: (2+2·3)%7=1. Slot 1 is empty → insert 37.
- Why No Clustering? Even with collisions at slot 2, keys spread uniformly due to unique step sizes from the 2nd hash function h2. Different probe sequences for collisions:
  - 23 used step size 4 → slots 2, 6.
  - 30 used step size 1 → slots 2, 3.
  - 9 used step size 5 → slots 2, 0.
  - 2 used step size 3 → slots 2, 5.
  - 37 used step size 3 → slots 2, 5, 1.

Step 1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |

Step 2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | 16 | | | | |

Step 3

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | 16 | | | | 23 |

Step 4

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | 16 | 30 | | | 23 |

Step 5

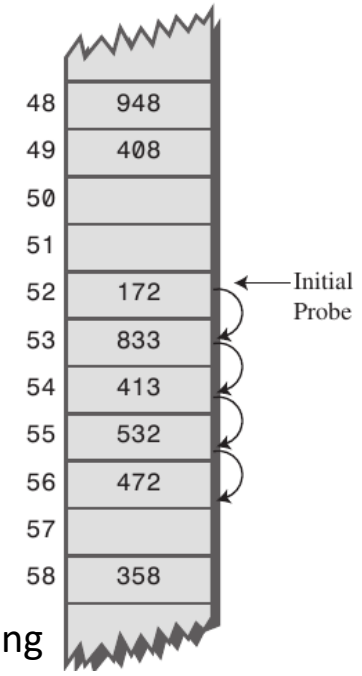
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 9 | | 16 | 30 | | | 23 |

Step 6

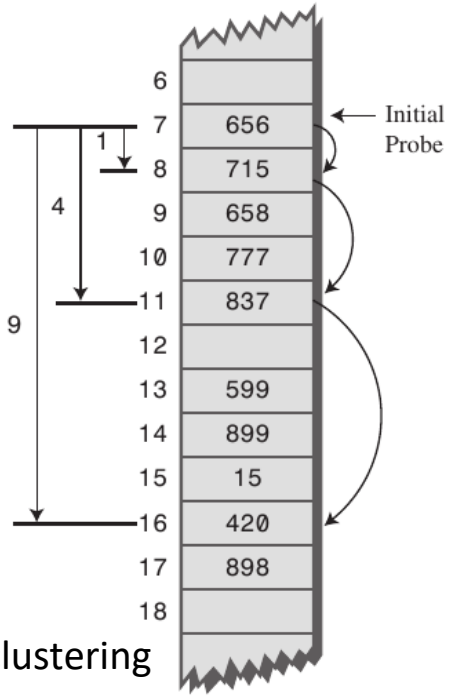
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 9 | 37 | 16 | 30 | | 2 | 23 |

# Primary Clustering and Secondary Clustering

- Primary clustering is the tendency for a collision resolution scheme such as linear probing to create long runs of filled slots clustered together.
  - If the primary hash index is x, subsequent probes go to x+1, x+2, x+3 and so on, this results in Primary Clustering.
  - Once the primary cluster forms, the bigger the cluster gets, the faster it grows. And it reduces the performance.
- Secondary clustering is the tendency for a collision resolution scheme such as quadratic probing to create long runs of filled slots away from the hash slot of keys.
  - Less severe than primary clustering.




Primary clustering



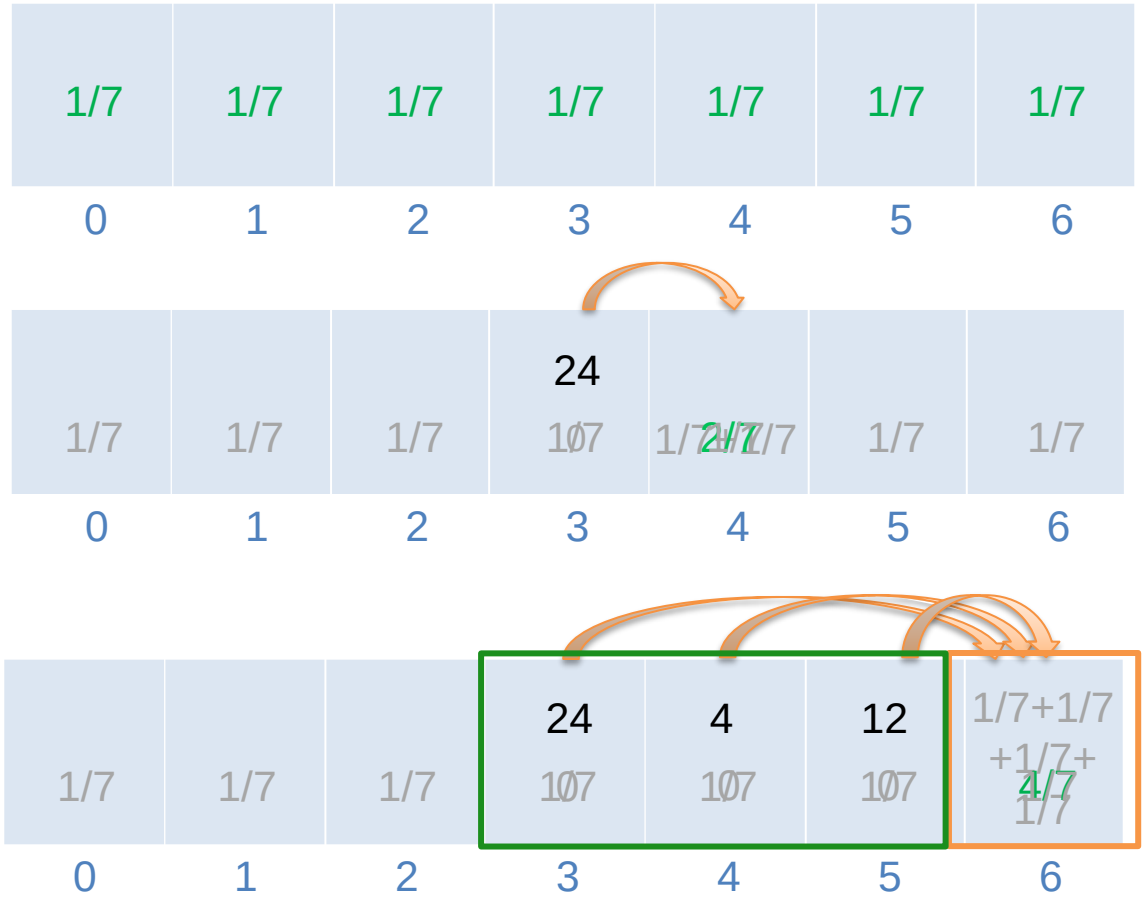

Secondary clustering

# Linear Probing: Primary Clustering

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

Hash(k) = k mod 7

All keys equally likely



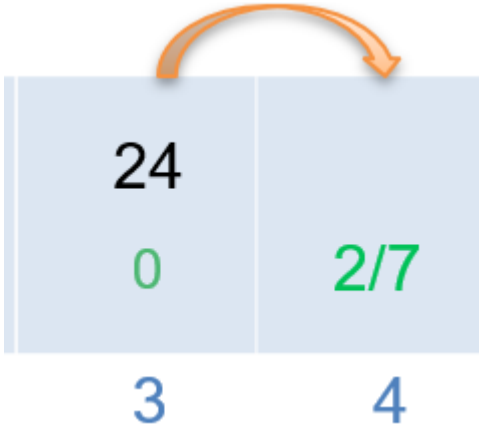

Cluster is a contiguous block of items.

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

Higher insert and search costs - O(n)

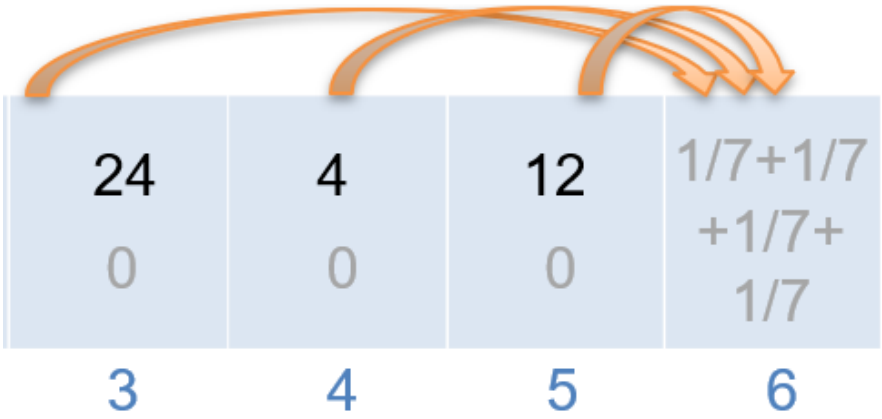
# Linear Probing: Primary Clustering Explanations

- Case 1: Probability of placing into slot 4 = prob(hashing into 3) + prob(hashing into 4) = 1/7+1/7 = 2/7




Case 1

- Case 2: Probability of placing into slot 6 = prob(hashing into 3) + prob(hashing into 4) + prob(hashing into 5) + prob(hashing into 6) = 1/7+1/7+1/7+1/7=4/7




Case 2

# Linear Probing: Primary Clustering (Contd.)

Three methods to mitigate the problem

1. Better-designed hash function

2. Alternative probing methods

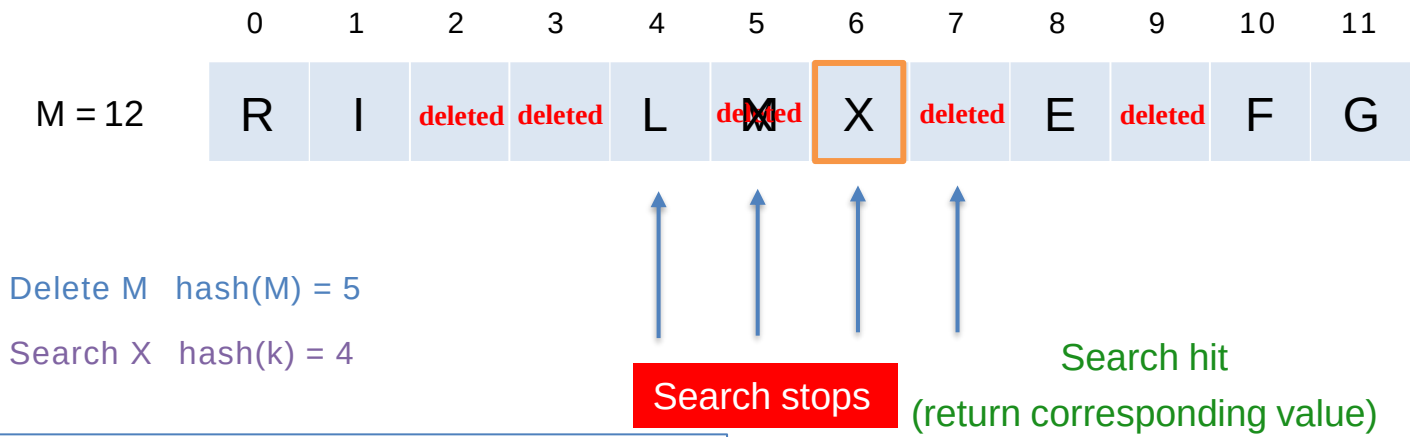
3. Resize the hash table when it's "full"

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

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 slot as with a tombstone to indicate "empty but deleted". Probing is continued when encountering such slot. An add operation can store data in such slot.

Table pollution issue: These tombstones 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 (M at slot 5)
2.Go to the next slot (6)
3.If the slot is empty, quit
4.If the slot is full, delete the element (X) in that slot (6) and re-add it to the hash table using the normal means (in slot 5). (The item must be removed before re-adding, as it is likely that the item is added back into its original slot.)
5.Repeat at step 2.

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

# Separate Chaining vs Linear Probing

- Separate Chaining
- Pros:
- 1. Simplicity of Deletion: Deleting an element involves simply removing it from the linked list, without affecting other elements.
- 2. Less Sensitive to Hash Function Quality and Load Factor: Poor hash functions cause fewer performance issues compared to linear probing, as clustering is not a concern.
- Cons:
- 1. Cache Performance: Poorer cache locality compared to linear probing because linked list nodes often occupy non-contiguous memory locations.
- 2. Memory Overhead: Requires additional memory for pointers in linked lists, which can be significant if many collisions occur.
- Linear probing
- Pros:
- 1. Cache Efficiency: Traverses the hash table array linearly, leading to better cache performance due to spatial locality, since an array occupies contiguous memory locations.
- 2. Memory Efficiency: Does not require extra memory for pointers or external structures; all data resides within the array itself.
- Cons
- 1. Deletion Complexity: Deletion requires marking slots as "tombstones" or rehashing, which complicates management and may degrade performance over time.
- 2. Clustering Issues: Suffers from primary clustering, where groups of occupied slots grow, increasing probe lengths and degrading performance. Performance worsens significantly as the load factor approaches 1.
- 3. Load Factor Sensitivity: Performance is highly sensitive to load factor; tables must be resized when the load factor exceeds a certain threshold (commonly 0.7).

# Comparison of Techniques for Handling Hash Collisions

| **Method** | Key Distribution | Clustering Risk | Space Efficiency |
|---|---|---|---|
| **Separate Chaining** | Keys stored in linked lists | None | Lower (uses pointers) |
| **Linear Probing** | Sequential placement | High (primary clustering) | Higher |
| **Quadratic Probing** | Spreads keys quadratically | Reduced (secondary clustering) | Higher |
| **Double Hashing** | Spreads using a step size | Minimal | Higher |

# Hashing Tutorial Videos

- Introduction to Hash Maps, Tech With Nikola
  - https://www.youtube.com/watch?v=t-vM3LJDfug&list=PL60uk12YwbCYekcB_pvsT3EVdS-tJcQju&index=4
- Hash tables in 4 minutes
  - https://www.youtube.com/watch?v=knV86FlSXJ8
- Hashing | Set 1 (Introduction) | GeeksforGeeks
  - https://www.youtube.com/watch?v=wWgIAphfn2U
- Hashing | Set 2 (Separate Chaining) | GeeksforGeeks
  - https://www.youtube.com/watch?v=_xA8UvfOGgU
- Hashing | Set 3 (Open Addressing) | GeeksforGeeks
  - https://www.youtube.com/watch?v=Dk57JonwKNk
- Hashing Animations | Data Structure | Visual How
  - https://www.youtube.com/watch?v=VeYKEMY2F9k
- Linear Probing in Hashing Animations | Data Structure | Visual How
  - https://www.youtube.com/watch?v=98Y0UDZ9vvs
- Quadratic Probing Hashing Animations | Data Structure | Visual How
  - https://www.youtube.com/watch?v=0CFJAkpnhBg
- Separate Chaining in Hashing Animations | Data Structure | Visual How
  - https://www.youtube.com/watch?v=LRtKQdsJC3o

# Full-Length Lectures

- [CSE 373 WI24] Lecture 08: Hash Collision Resolutions
  - https://www.youtube.com/watch?v=yuyTVhOL8B8&list=PLEcoVsAaONjd5n69K84sSmAuvTrTQT_Nl&index=7