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
- Identify other challenges associated with Hash Tables
- Explain the difference between a Hash Set, Hash Map and Hash Table

Motivation

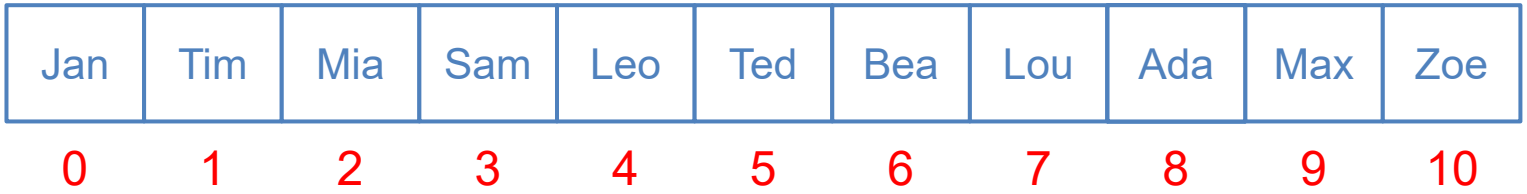
Find Ada

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]



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 tables in 4 minutes
https://www.youtube.com/watch?v=knV86FlSXJ8
Hashing | Set 1 (Introduction) | GeeksforGeeks
https://www.youtube.com/watch?v=wWgIAphfn2U

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

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

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

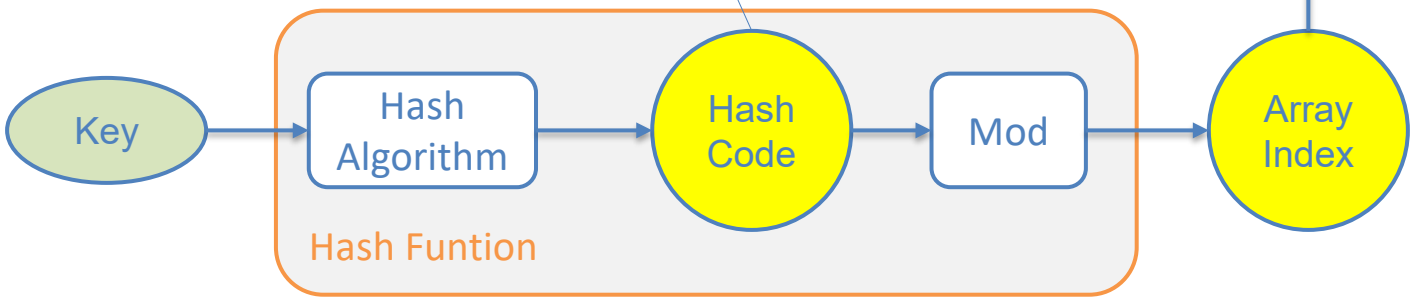
Hash tables are often used to store <key, value> pairs, which can be the objects in java. Key is just one of the object's property

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

myData = Array[9]

Ada
03/27/1969
Inventor

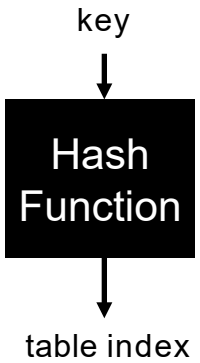
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Key Value	Ted
10/12/1955 Astronomer	11/02/1986 Inventor	05/12/1966 Philosopher	10/12/1951 Biologist	04/09/1977 Physicist	10/12/1965 Inventor	10/12/1990 Philosopher	02/13/1943 Biologist	06/24/1971 Scientist		12/06/1998 Actress
0	1	2	3	4	5	6	7	8	9	10



Hash Function

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

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.

$$address = key \ Mod \ n$$

- 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

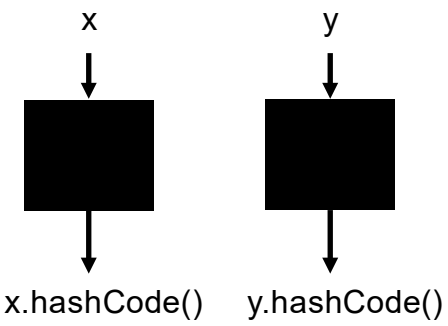
All Java classes inherit a method hashCode(), which returns a 32-bit int.

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

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

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.

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

01000000 00100100 01111010 11100001

XOR **01000111 10101110 00010100 01111011**

00000111 10001010 01101110 10011010

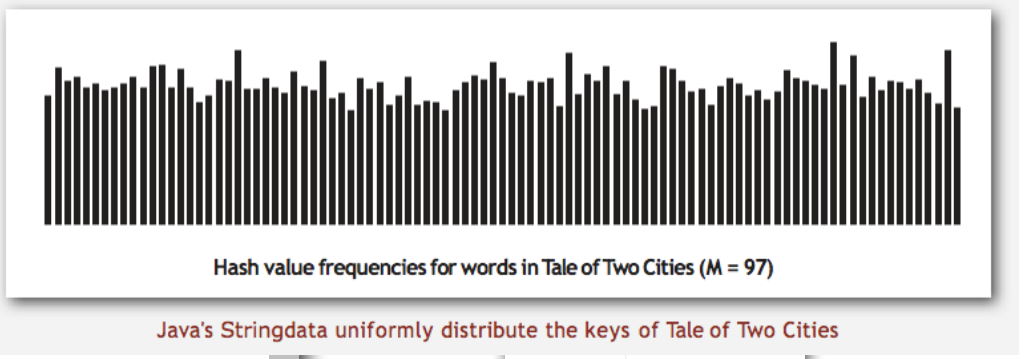
- return 126512794
- all the digits contribute to the hash code

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;
    }
}
```

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 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;
    ...
    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

store cache of hash code

- 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;
    }
}
```

- 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 << 5) - i.
- Using primes is an old technique.

nonzero constant

typically a small prime

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

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

- Combine each significant field using the 31*x* + *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;

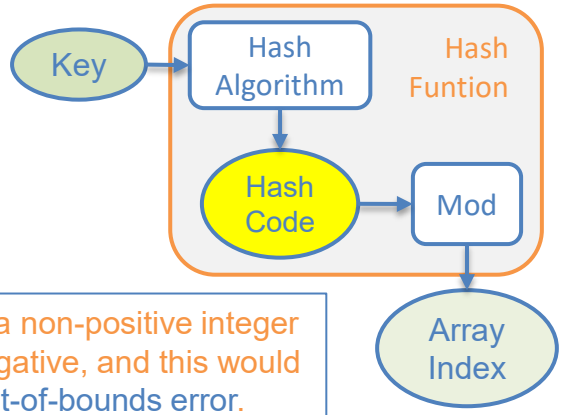
Modular Hashing

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.

1-in-a-billion bug

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

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

correct

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

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

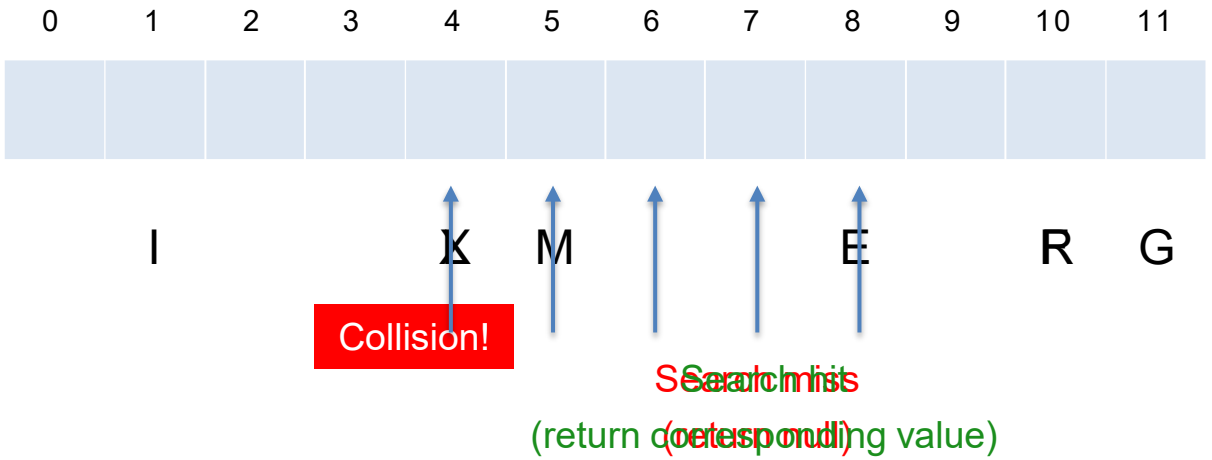
- Integer representation: In Java, integers are stored using 32 bits in two's complement format.
- Range of int: The range of int in Java is from -2^31 to 2^31 - 1, which is -2,147,483,648 to 2,147,483,647.
- Integer.MIN_VALUE: This constant represents the minimum value an int can hold, which is -2,147,483,648.
- 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



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

Search E hash(E) = 8

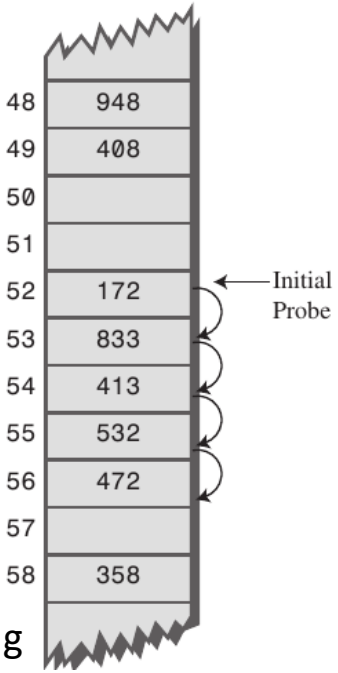
Search X hash(X) = 4

Search Y hash(Y) = 5

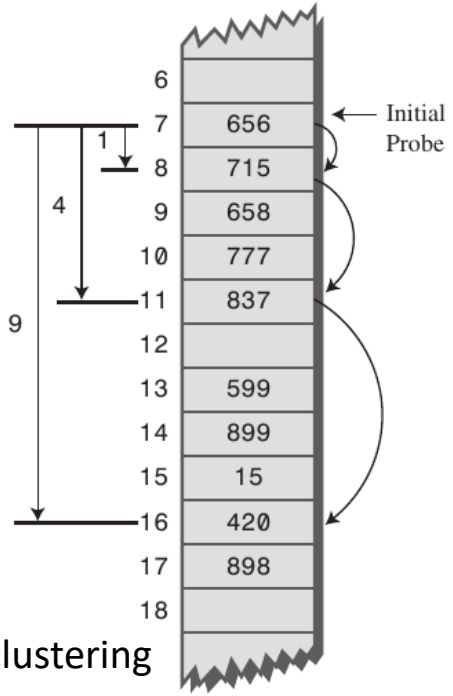
Idea for search: Search the hashed table index, and if occupied but no match, try the next spot until the next spot is empty or is the hashed table index (i.e., loops back and the table is full).

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 near the hash position of keys.
  - 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 position of keys.
  - If the primary hash index is x, probes go to x+1, x+4, x+9, x+16, x+25 and so on, this results in Secondary Clustering.



Primary clustering



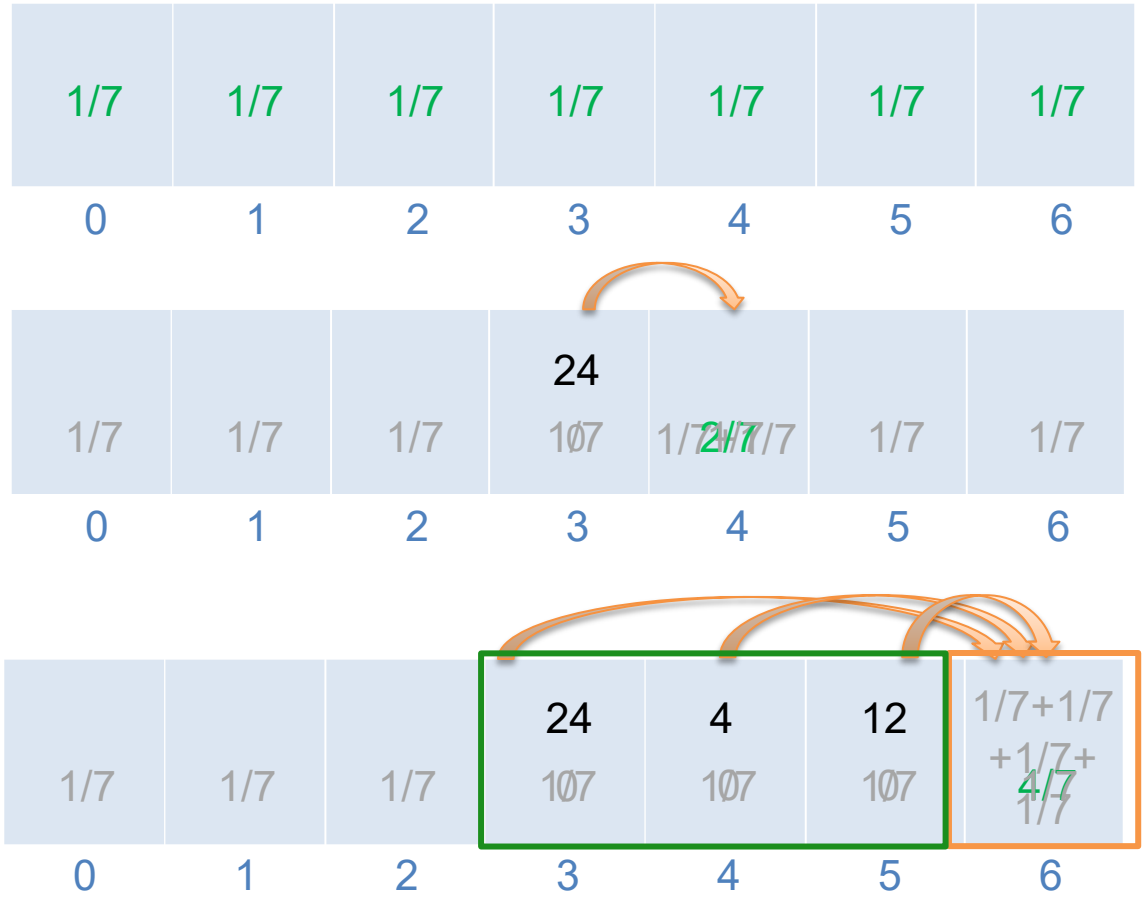
Secondary clustering

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.

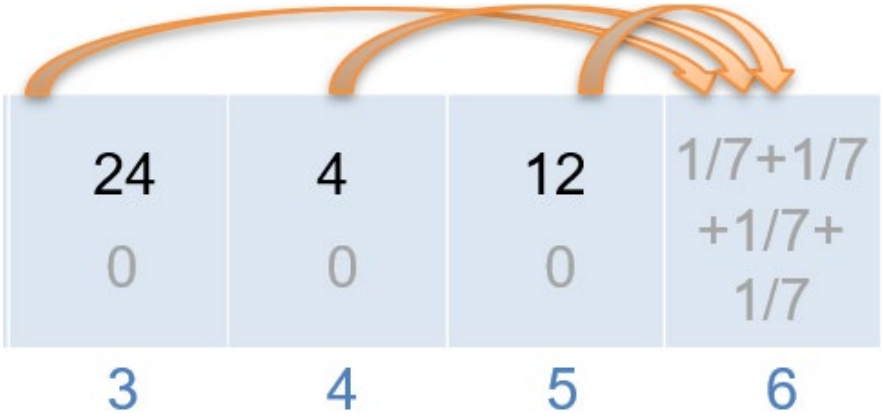
Higher insert and search costs - O(n)

Linear Probing: Primary Clustering Explanations

- Case 1: Probabil<ity of placing into position 4 = prob(hashing into 3) + prob(hashing into 4) = 1/7+1/7 = 2/7

Case 1

- Case 2: Probability of placing into position 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"

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

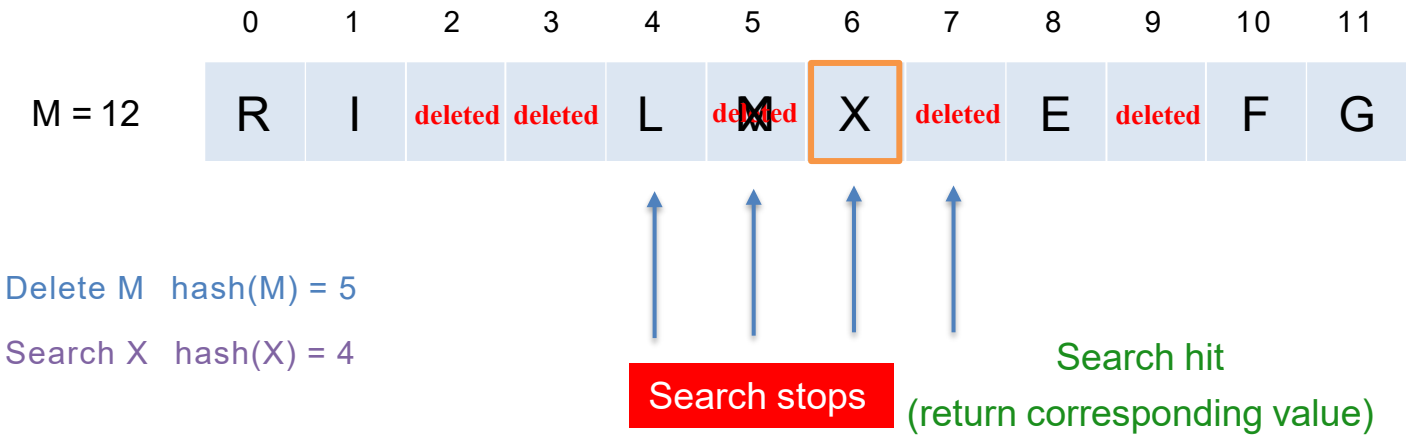
Repopulate the items into a larger array, when load factor > 70%

Open addressing

- linear probing
  - try next open spot
- Plus 3 rehash
  - skip 3 spots, not just next open spot
- Quadratic probing
  - skip (failed attempts)^2 spots, i.e, 1^2, 2^2, 3^2
- Double hashing
  - apply a second hash function to key when a collision occurs

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.

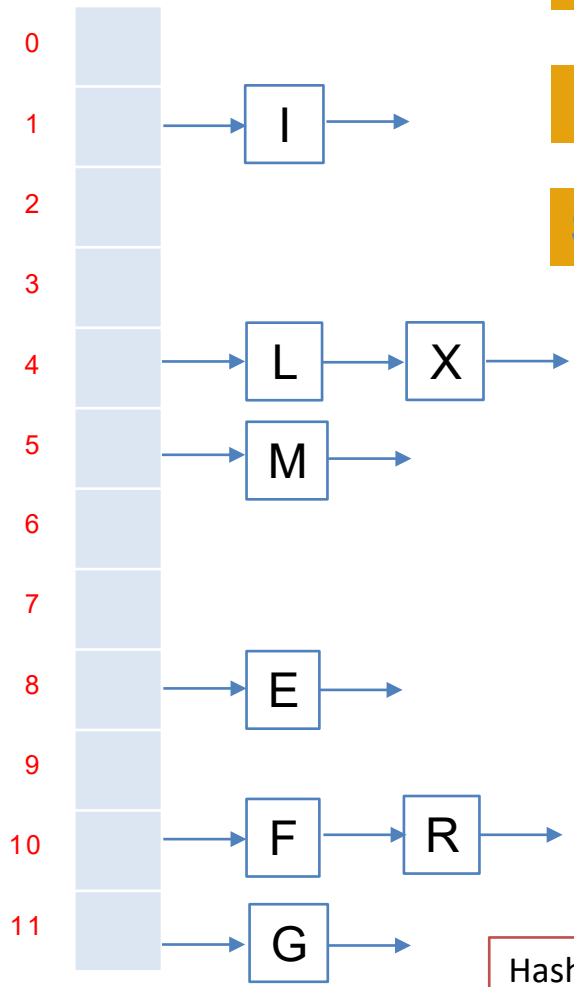
Table pollution issue: These deleted flags may bridge together otherwise unrelated data (different hashcodes). In the worst case, search is linear time.

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

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

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

Search E hash(E) = 8

Search X hash(X) = 4

Search Y hash(Y) = 5

Hashing | Set 2 (Separate Chaining) | GeeksforGeeks
https://www.youtube.com/watch?v=_xA8UvfOGgU

Separate Chaining vs Linear Probing

- Separate chaining.
  - Easier to implement delete.
  - Performance degrades gracefully.
  - less sensitive to poorly-designed hash function
- Linear probing.
  - Less wasted space.
  - Better cache performance

"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."

With linear probing, an array occupies contiguous memory locations; with separate chaining, a linked list occupies non-contiguous memory locations.

"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."

Hashing Tutorial Videos

- 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