Hashing

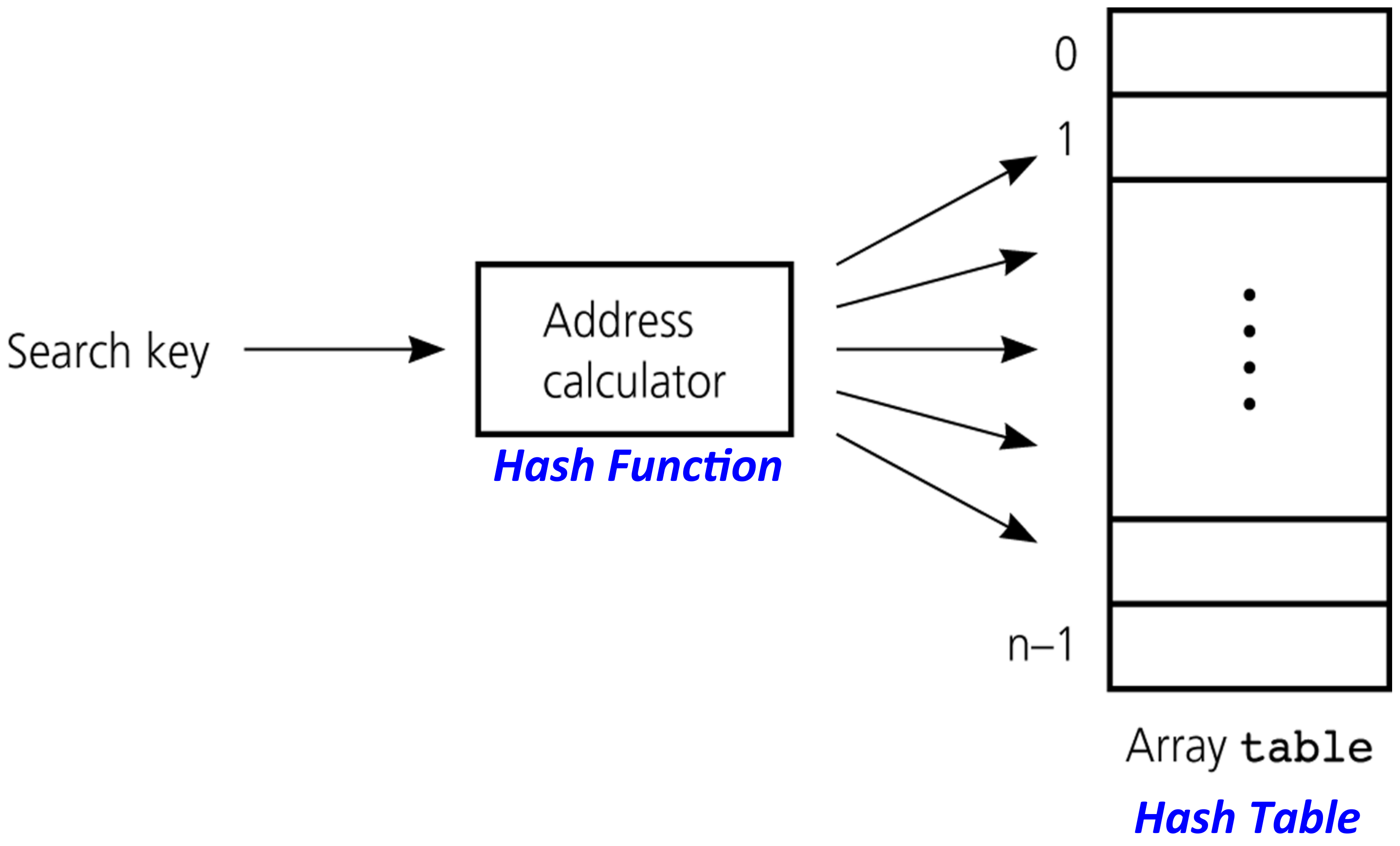
CSCI-2270

Elizabeth Boese

Hashing

- Balanced search trees *(red-black, and AVL trees)*, implement table operations in O (logN) time
  - retrieval, insertion, and deletion
- Can we find a data structure so that we can perform these table operations even faster (e.g., in O(1) time)?
  - **Hash Tables**

Hash Function -- Address Calculator



Hash Tables

- In **Hash Tables**, we have
  - **An array** (index ranges 0 … n – 1)
    - Each array location is called a ***bucket***
  - **Hash Function** (An address calculator) maps a search key into an array index between 0 … n – 1
  - E.g., **h(x) = x mod n**, where x is an integer
  - **Hashing**

Collisions

- **Perfect hash function**
  maps each search key into a unique location of the hash table.
  - Possible if we know all search keys in advance.
  - In practice(we do not know all search keys, so a hash function can <u>map more than one key into the same location.</u>

- **Collisions**
  occur when a hash function maps more than one item into the same array location.
  - We have to resolve the collisions.

Hash Functions

- We can design different hash functions.
- But a **good** hash function should
  - be easy and fast to compute
  - place items uniformly (evenly) throughout the hash table.
- We will consider only integer hash functions
  - On a computer, everything is represented with bits.
  - They can be converted into integers.
    - 1001010101001010000110…. remember?

Everything is an Integer

- If search keys are strings, think of them as integers, and apply a hash function for integers.

- For example, strings can be encoded using ASCII codes of characters.

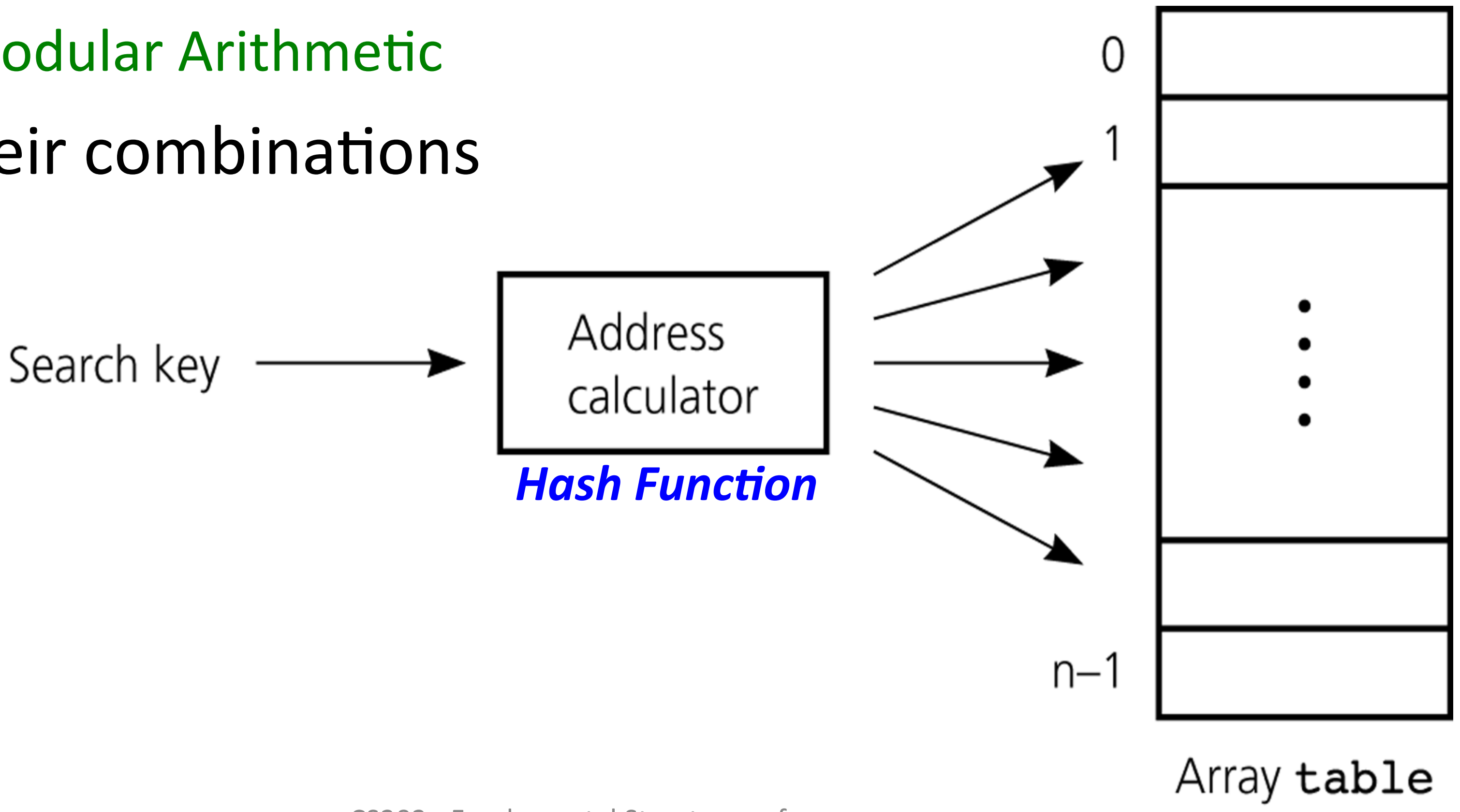
- Consider the string "NOTE"
  - ASCII code of **N** is 4Eh (01001110), **O** is 4Fh (01001111),
    T is 54h(01010100), **E** is 45h (01000101)
  - Concatenate four binary numbers to get a new binary number
    01001110010011110101010001000101= 4E4F5445h =
    1313821765

How to Design a Hash Function?

Three possibilities

1. Selecting digits
2. Folding
3. Modular Arithmetic

- Or, their combinations



Hash Functions -- Selecting Digits

1. **Select certain digits** and combine to create the address.

- For example, suppose that we have 11-digit Turkish nationality ID's
  - Define a hash function that selects the 2nd and 5th most significant digits

    h(0**3**34**7**5678) = **37**

    h(0**2**34**5**5678) = **25**
  - Define the table size as 100

- Is this a good hash function?
  - No, since it does not place items uniformly.

Hash Functions -- Folding

2. **Folding –** selecting all digits and adding them.

- For example, suppose previous nine-digit numbers
  - Define a hash function that selects all digits and adds them

    h(033475678) = **0 + 3 + 3 + 4 + 7 + 5 + 6 + 7 + 8** = 43

    h(023455678) = **0 + 2 + 3 + 4 + 5 + 5 + 6 + 7 + 8** = 40
  - Define the table size as 82

- We can select a group of digits and add the digits in this group as well.

Hash Functions -- Modular Arithmetic

3. **Modular arithmetic** – provides a simple and effective hash function.

$$\mathbf{h(x) = x} \ \ \text{mod} \ \ \mathbf{tableSize}$$

- The table size should be a prime number.
  - ***Why? Think about it.***

Why Primes?

- Assume you hash the following with x mod 8:
  - 64, 100, 128, 200, 300, 400, 500

0	64 128 200 400
1	
2	
3	
4	100 300 500
5	
6	
7	

Why Primes?

- Now try it with x mod 7
  - 64, 100, 128, 200, 300, 400, 500

0	
1	64 128 400
2	100
3	500
4	200
5	
6	300

Rationale

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

- If we are adding numbers $a_1, a_2, a_3 \ldots a_4$ to a table of size m
  - All values will be hashed into multiples of

    $\gcd(a_1, a_2, a_3 \ldots a_4, m)$

  - For example, if we are adding 64, 100, 128, 200, 300, 400, 500 to a table of size 8, all values will be hashed to 0 or 4

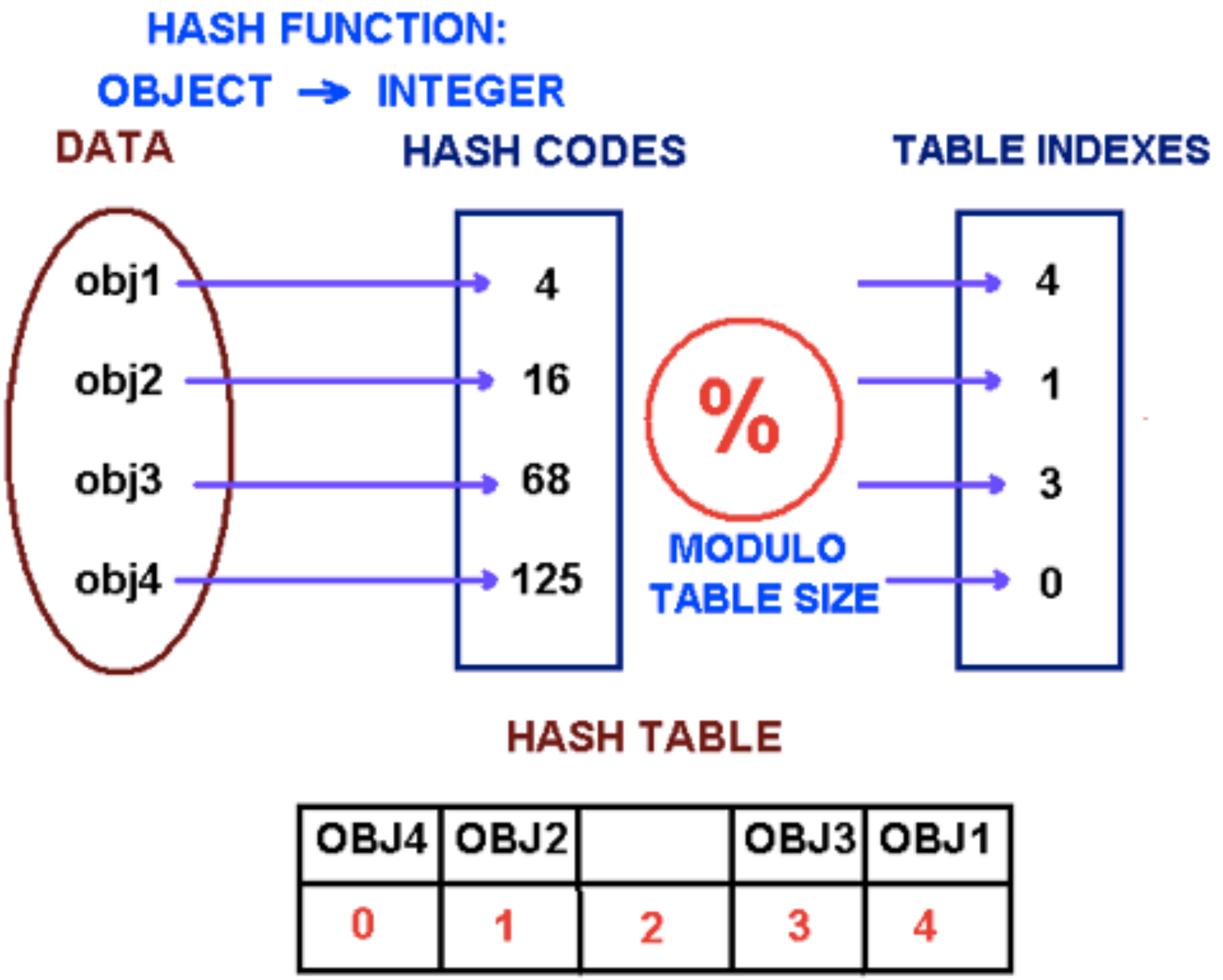
    $\gcd(64,100,128,200,300,400,500, \mathbf{8}) = 4$

  - When m is a prime $\gcd(a_1, a_2, a_3 \ldots a_4, m) = 1$, all values will be hashed to anywhere

    $\gcd(64,100,128,200,300,400,500,\mathbf{7}) = 1$

    unless $\gcd(a_1, a_2, a_3 \ldots a_4) = m$, which is rare.

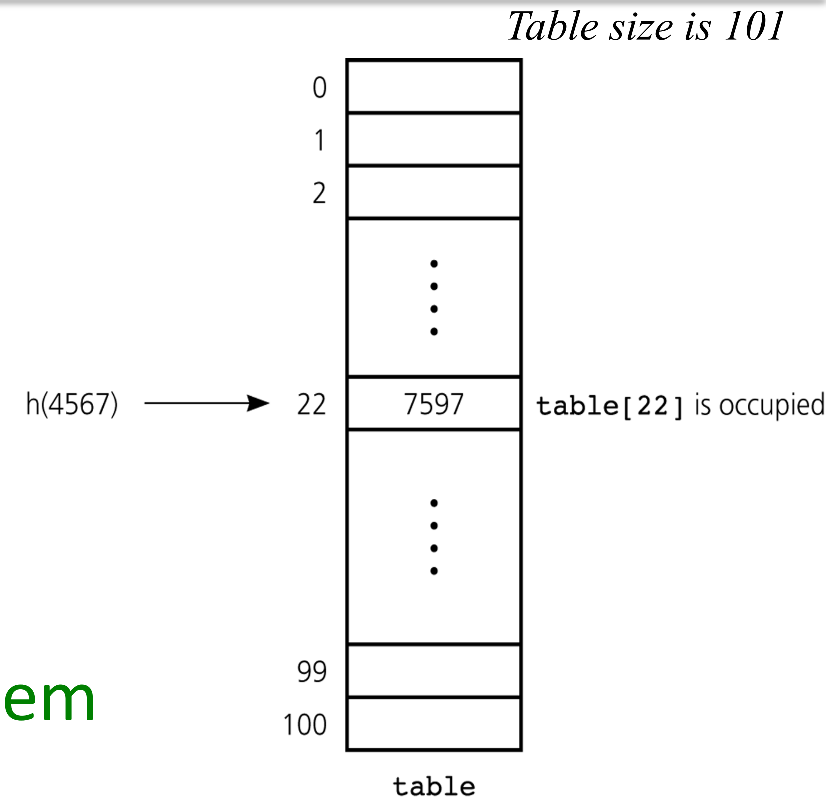
Hashing



OBJ4	OBJ2		OBJ3	OBJ1
0	1	2	3	4

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Hashing/hashing.html

Collision Resolution



Collision resolution

two general approaches

- **Open Addressing**

  Each entry holds one item
- **Chaining**

  Each entry can hold more than 1 item

  (**Buckets** – hold certain number of items)

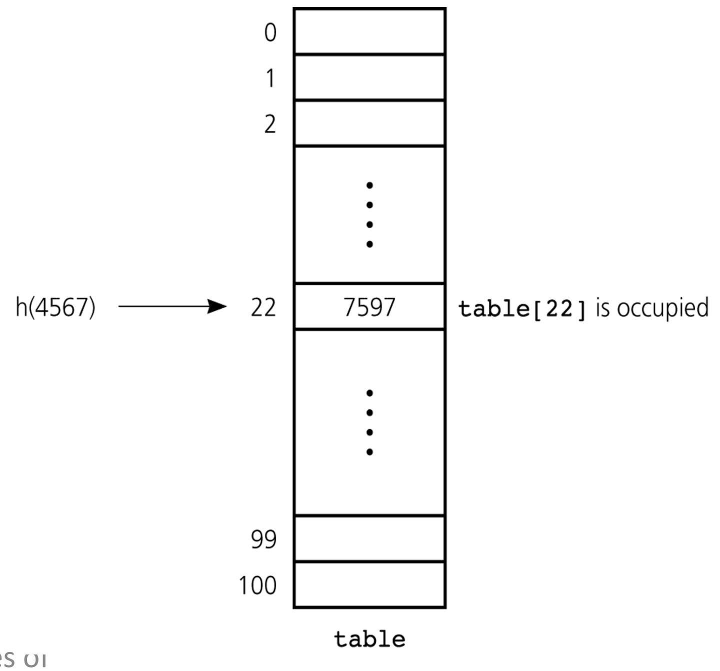
Open Addressing

- **Open addressing** – probes for some other empty location when a collision occurs.

- **Probe sequence** - sequence of examined locations.
  Different open-addressing schemes:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Open Addressing -- Linear Probing

- **Linear probing**: search table sequentially starting from the original hash location.
  - Check next location, if location is occupied.
  - Wrap around from last to first table location



Linear Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 → 2+1=3
    - 25 mod 11 = 3 → 3+1=4
    - 24 mod 11 = 2 → 2+1, 2+2, 2+3=5
    - 10 mod 11 = 10
    - 9 mod 11 = 9 → 9+1, 9+2 mod 11 =0

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Linear Probing -- Clustering Problem

Linear Probing Problem

- **Primary clustering**
  - table items tend to **cluster** together in the hash table.

    (i.e. table contains groups of consecutively occupied locations.)

  - Clusters can get close to one another, and merge into a larger cluster.
  - Thus, the one part of the table might be quite dense, even though another part has relatively few items.
  - Causes long probe searches → decreases the overall efficiency.

Open Addressing -- Quadratic Probing

- **Quadratic probing:** almost eliminates clustering problem

- Approach:
  - Start from the original hash location `i`
  - If location is occupied, check locations $i+1^2$, $i+2^2$, $i+3^2$, $i+4^2$ ...
  - Wrap around table, if necessary.

Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 ➔ $2+1^2=3$
    - 25 mod 11 = 3 ➔ $3+1^2=4$
    - 24 mod 11 = 2 ➔ $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 ➔ $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

Open Addressing -- Double Hashing

- **Double hashing** also reduces clustering.
- Increment using a **second hash function h_2**.
  Should satisfy:

$$h_2(\text{key}) \neq 0$$

$$h_2 \neq h_1$$

- Probes following locations until it finds an unoccupied place

$$h_1(\text{key})$$

$$h_1(\text{key}) + h_2(\text{key})$$

$$h_1(\text{key}) + 2*h_2(\text{key}),$$

...

Double Hashing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

    $h_1(x) = x \bmod 11$

    $h_2(x) = 7 - (x \bmod 7)$
  - Insert keys: 58, 14, 91
    - 58 mod 11 = 3
    - 14 mod 11 = 3

      $\rightarrow h_2 = (7 - (14 \bmod 7)) = 7$

      $\rightarrow h_1 + h_2 \rightarrow 3 + 7 = 10$
    - 91 mod 11 = 3

      $\rightarrow$h2 = (7 – (91 mod 7)) = 7

      $\rightarrow h_1 + h_2 \rightarrow 3 + 7 = 10$ TAKEN

      so 3 + 2*7 = 17

      fit in table so $\rightarrow$ 17 mod 11 = 6

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Open Addressing -- Retrieval & Deletion

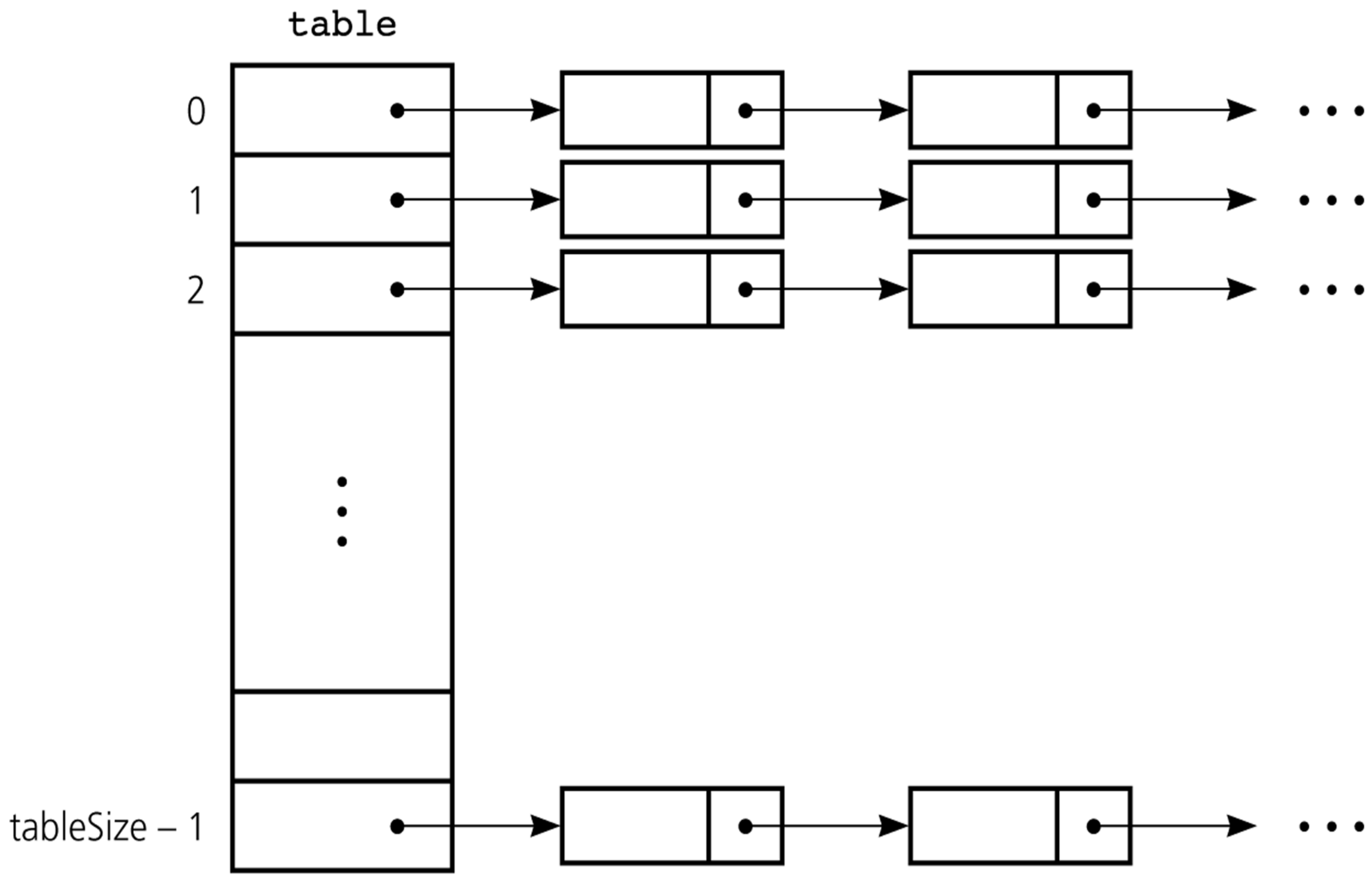
- Retrieving an item with a given key:
  - (same as insertion): probe the locations until we find the desired item or we reach to an empty location.

- Deletions in open addressing cause complications
  - We CANNOT simply delete an item from the hash table because this new empty (a deleted) location causes to stop prematurely (incorrectly) indicating a failure during a retrieval.
  - Solution: We have to have three kinds of locations in a hash table: ***Occupied, Empty, Deleted***.
  - A deleted location will be treated as an occupied location during retrieval.

Separate Chaining

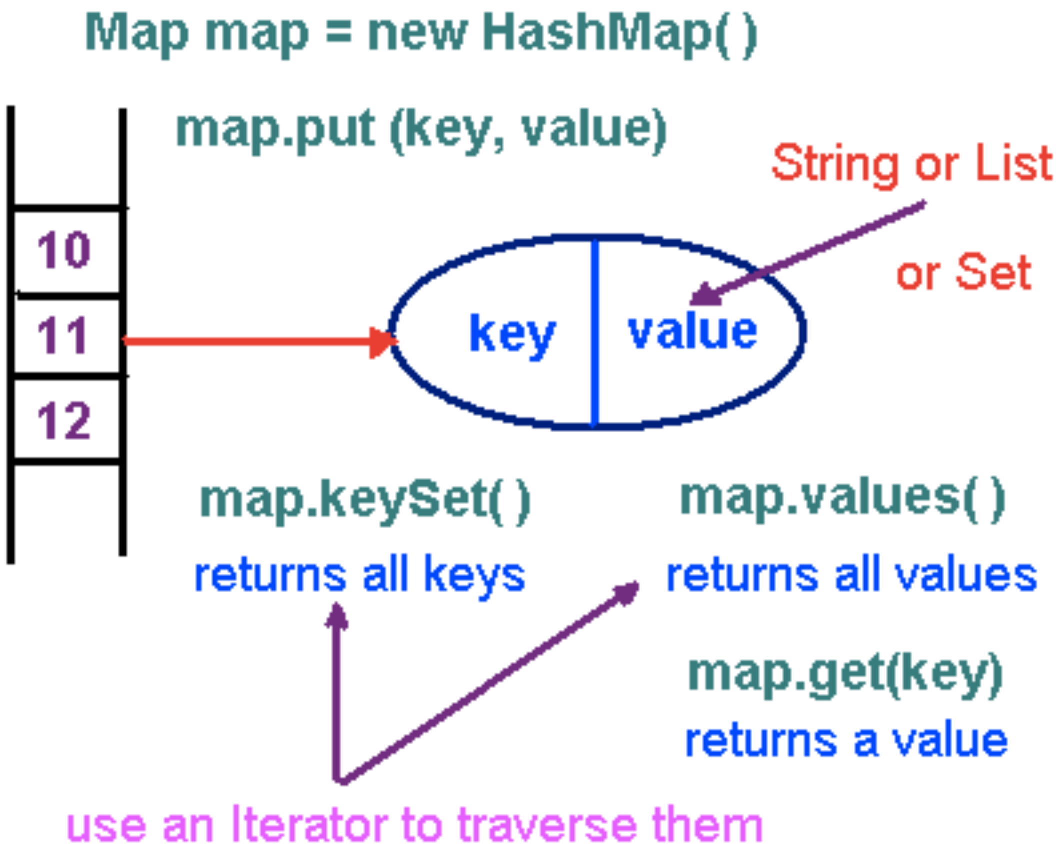
- Another way to resolve collisions is to change the structure of the hash table.
  - In open-addressing, each location holds only one item.
- **Idea 1**: each location is itself an array called bucket
  - Store items that are hashed into same location in this array.
  - Problem: What will be the size of the bucket?
- **Idea 2:** each location is itself a linked list.
  Known as **separate-chaining**.
  - Each entry (of the hash table) is a pointer to a linked list (the chain) of the items that the hash function has mapped into that location.

Separate Chaining



Each location of the hash table contains a pointer to a linked list

HashMap



```
Iterator itr = map.keySet().iterator();
while( itr.has.Next())
{
   Object key = itr.next();
   System.out.println(map.get(key));
}
```

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Hashing/hashing.html

Hashing Analysis

Hashing -- Analysis

- An analysis of the average-case efficiency of hashing involves the **load factor** α**:**

  α= (current number of items) / tableSize

- α measures how full a hash table is.
  - Hash table should not be too loaded if we want to get better performance from hashing.

- In average case analyses, we assume that the hash function uniformly distributes keys in the hash table.

- Unsuccessful searches generally require more time than successful searches.

Separate Chaining -- Analysis

- **Separate Chaining –** approximate average number of comparisons (probes) that a search requires :

$$1+\frac{\alpha}{2}$$ *for a successful search*

$$\alpha$$ *for an unsuccessful search*

- It is the most efficient collision resolution scheme.
- But requires more storage *(needs storage for pointers)*.
- It easily performs the deletion operation.
  (Deletion is more difficult in open-addressing.)

Linear Probing -- Analysis

Example: Find the average number of probes for a successful search and an unsuccessful search for this hash table? Use the following hash function: **h(x) = x mod 11**

Successful Search: ***Try 20, 30, 2, 13, 25, 24, 10, 9***

20: 9 **30:** 8 **2:** 2 **13:** 2,3
25: 3, 4 **24:** 2, 3, 4, 5 **10:** 10 **9:** 9, 10, 0

Avg. no of probes = (1+1+1+2+2+4+1+3)/8 = 1.9

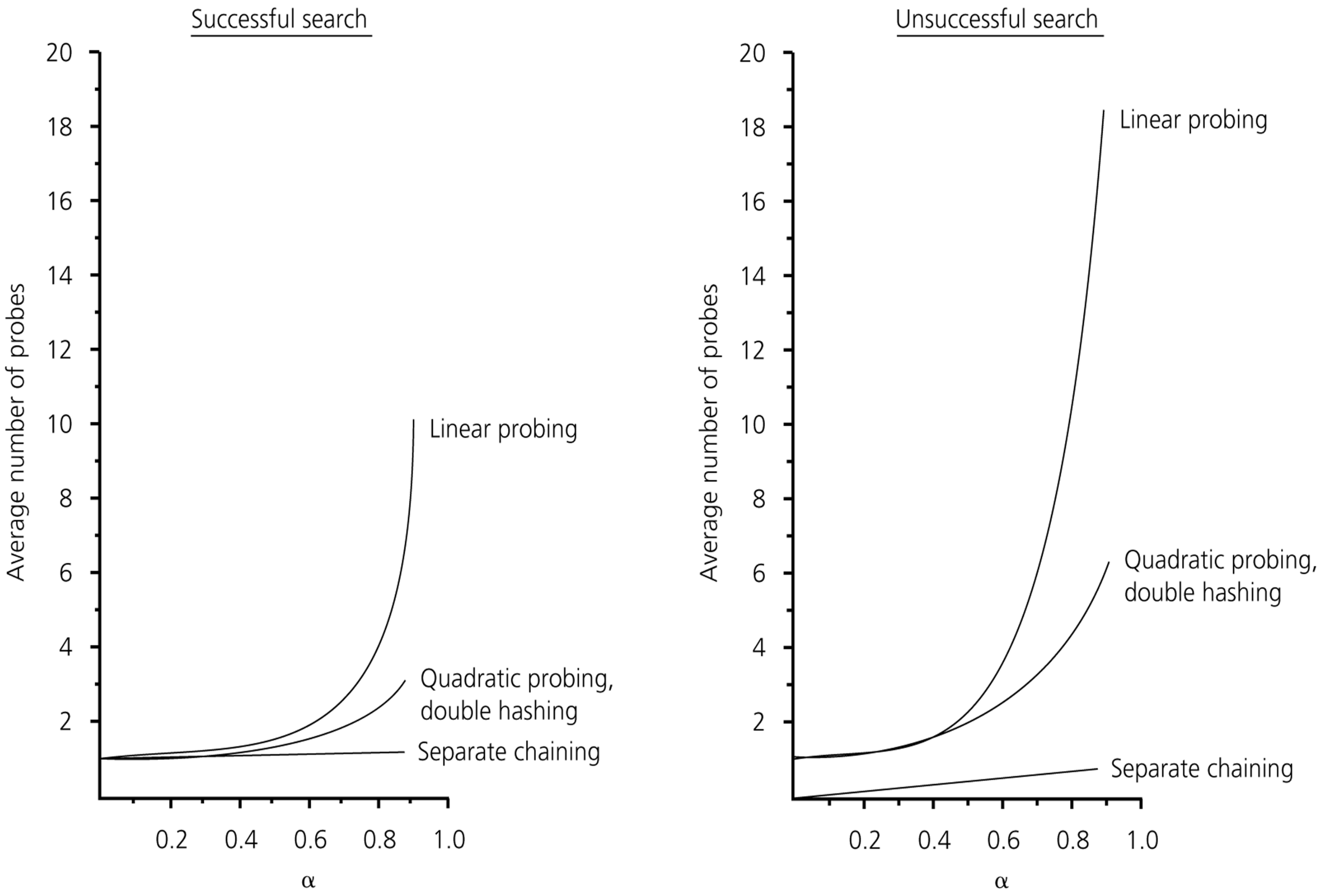
Unsuccessful Search: ***Try 0, 1, 35, 3, 4, 5, 6, 7, 8, 9, 32***

0: 0,1 1:1 35: 2, 3, 4, 5, 6
3: 3, 4, 5, 6 4: 4, 5, 6 5: 5, 6
6:6 7: 7 8: 8, 9, 10, 0, 1
9: 9, 10, 0, 13 2: 10,0,1

Avg. no of probes =(2+1+5+4+3+2+1+1+5+4+3)/11 = 2.8

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

The relative efficiency of four collision-resolution methods



What Constitutes a Good Hash Function

- A hash function should be <u>easy</u> and <u>fast</u> to compute.
- A hash function should <u>scatter the data evenly</u> throughout the hash table.
  - How well does the hash function scatter random data?
  - How well does the hash function scatter non-random data?
- Two general principles :
  1. The hash function should use entire key in the calculation.
  2. If a hash function uses modulo arithmetic, the table size should be prime.

Example: Hash Functions for Strings

Hash Function 1

- Add up the ASCII values of all characters of the key.

```
int hash(const string &key, int tableSize)
{
    int hasVal = 0;

    for (int i = 0; i < key.length(); i++)
        hashVal += key[i];
    return hashVal % tableSize;
}
```

- Simple to implement and fast.
- However, if the table size is large, the function does not distribute the keys well.
  - e.g. Table size =10,000 key length <= 8, the hash function can assume values only between 0 and 1016

Hash Function 2

- Examine only the first 3 characters of the key.

```
int hash (const string &key, int tableSize)
{
    return (key[0]+27 * key[1] + 729*key[2]) % tableSize;
}
```

- In theory, **26** * **26** * **26** = **17,576** different words can be generated. However, English is not random, only **2,851** different combinations are possible.

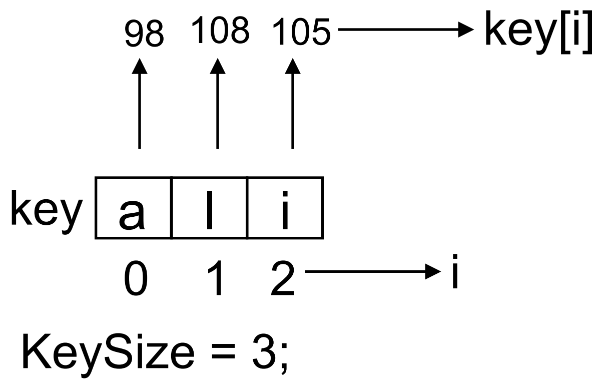
- Thus, this function although easily computable, is also not appropriate if the hash table is reasonably large.

Hash Function 3

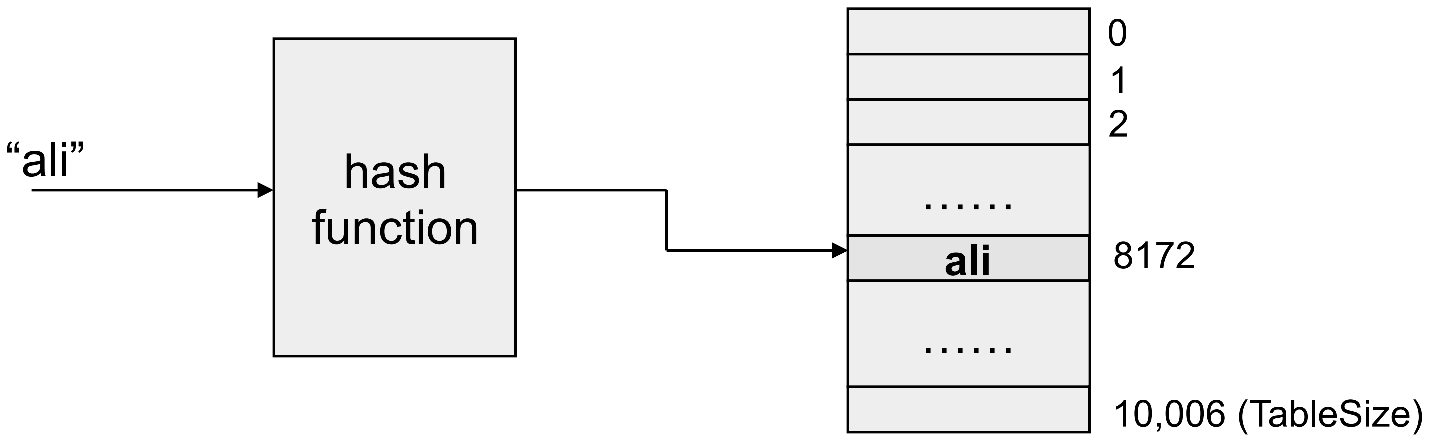
$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^{i}$$

```
int hash (const string &key, int tableSize)
{
   int hashVal = 0;

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

   hashVal %= tableSize;
   if (hashVal < 0)    /* in case overflows occurs */
       hashVal += tableSize;

   return hashVal;
};
```

Hash function for strings:



hash(“ali”) = (105 * 1 + 108*37 + 98*37^2) % 10,007 = 8172



Hash Table versus Search Trees

- In most of the operations, the hash table performs better than search trees.

- However, traversing the data in the hash table in a sorted order is very difficult.
  - For similar operations, the hash table will not be good choice (e.g., finding all the items in a certain range).

Performance

- With either chaining or open addressing:
  - **Search** - O(1) expected, O(n) worst case
  - **Insert** - O(1) expected, O(n) worst case
  - **Delete** - O(1) expected, O(n) worst case
  - **Min**, **Max** and **Predecessor**, **Successor** - O(n+m) expected and worst case

- Pragmatically, a hash table is often the best data structure to maintain a dictionary/table. However, the worst-case time is unpredictable.
- The best worst-case bounds come from balanced binary trees.

Other applications of hash tables

- To implement Table ADT, Dictionary ADT
- Compilers
  - *symbol table* (a data structure to keep track of declared variables)
- Spelling checkers
- Games
  - *transposition table (*track of positions it has encountered)
- Substring Pattern Matching
- Searching
- Document comparison

Substring Pattern Matching

Substring Pattern Matching

- Input: A text string **t** and a pattern string **p**.
- Problem: Does **t** contain the pattern **p** as a substring, and if so where?
- E.g: Is *Bilkent* in the news?
- **Brute Force:** search for the presence of pattern string **p** in text **t** overlays the pattern string at every position in the text. → O(mn) (m: size of pattern, n: size of text)
- **Via Hashing:** compute a given hash function on both the pattern string **p** and the m-character substring starting from the ith position of **t**. → O(n)

Slide by Steven Skiena

Hashing, Hashing, and Hashing

Search

- Udi Manber says that the three most important algorithms at Yahoo are hashing, hashing, and hashing.
- Hashing has a variety of clever applications beyond just speeding up search, by giving you a short but distinctive representation of a larger document.

Slide by Steven Skiena

Document Comparison

Document Comparison

- *Is this new document different from the rest in a large database? – Hash the new document, and compare it to* the hash codes of database.

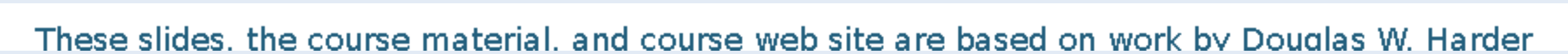
- *How can I convince you that a file isn't changed? – Check* if the cryptographic hash code of the file you give me today is the same as that of the original. Any changes to the file will change the hash code.

THE AWESOME NITTY-GRITTY

The Nitty-Gritty

These nitty-gritty slides from:

- Carlos Moreno
- cmoreno @ uwaterloo.ca
- EIT-4103
- https://ece.uwaterloo.ca/~cmoreno/ece250
- who cites the following:



These slides, the course material, and course web site are based on work by Douglas W. Harder

Nitty-Gritty

Linear probing using a jump size **p**

- if there is a collision, instead of skipping to the next bin to probe it, skip **p** bins forward and probe there.

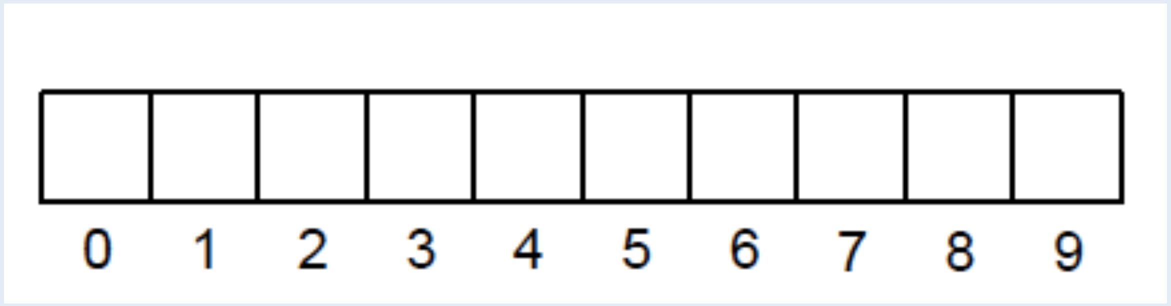
Nitty-Gritty

Example

- Hash table uses size 10
- First hash function, multiply the value times 117 and keep the right-most digit
- Second hash function (jump size), just use the same result, and take the second digit

Nitty-Gritty

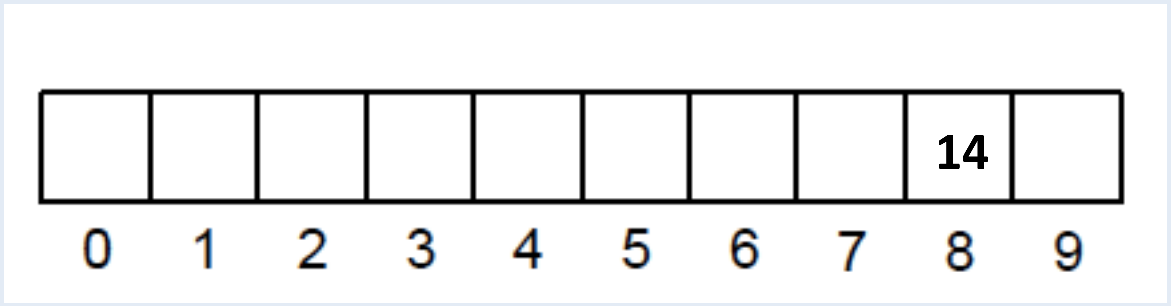
Insert values 14, 29, 43, 19, and 5 into the initially empty hash table:



Nitty-Gritty

Insert values **14**, 29, 43, 19, and 5 into the initially empty hash table:

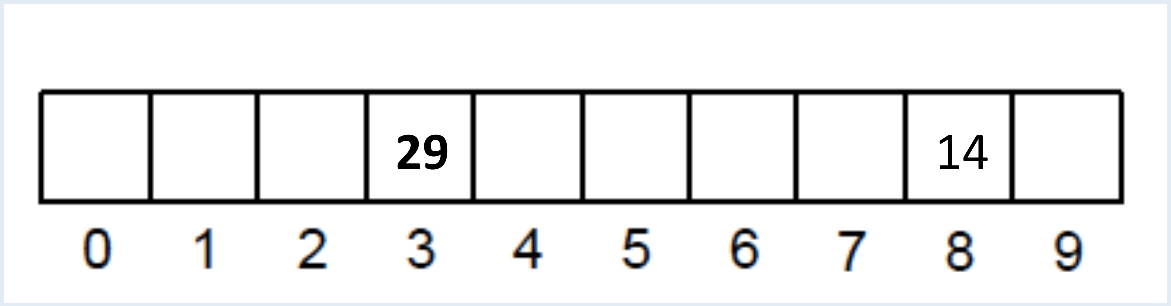
- **14***117 = 16**38** bin **8** ⇒
(and jump size **3** — not relevant now, since this insertion causes no collision)



Nitty-Gritty

Insert values 14, **29**, 43, 19, and 5 into the initially empty hash table:

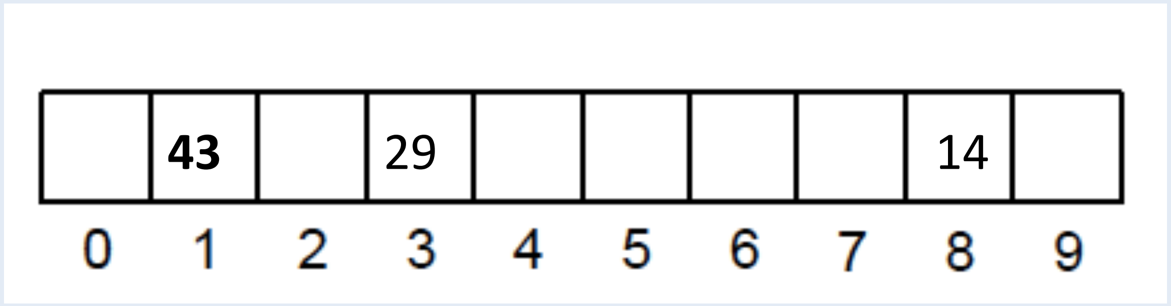
- **29***117 = 339**3** bin **3** ⇒ (jump size not relevant)



Nitty-Gritty

Insert values 14, 29, **43**, 19, and 5 into the initially empty hash table:

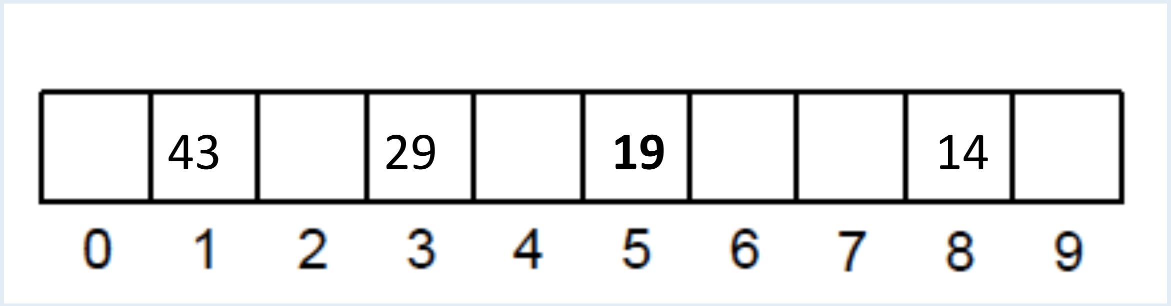
- **43***117 = 5031 bin **1** ⇒ (jump size not relevant)



Nitty-Gritty

Insert values 14, 29, 43, **19**, and 5 into the initially empty hash table:

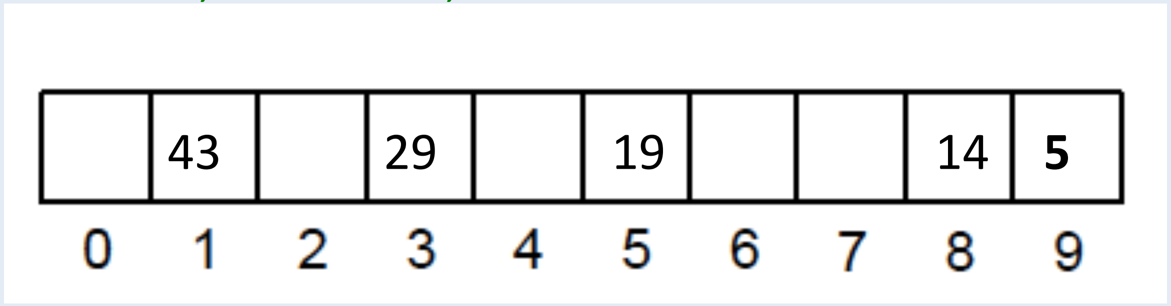
- **19***117 = 2223 bin ⇒ **3**, causing a collision (jump size given by the second digit, **2**)
- Probe bin 3 + 2 — available, so we're done



Nitty-Gritty

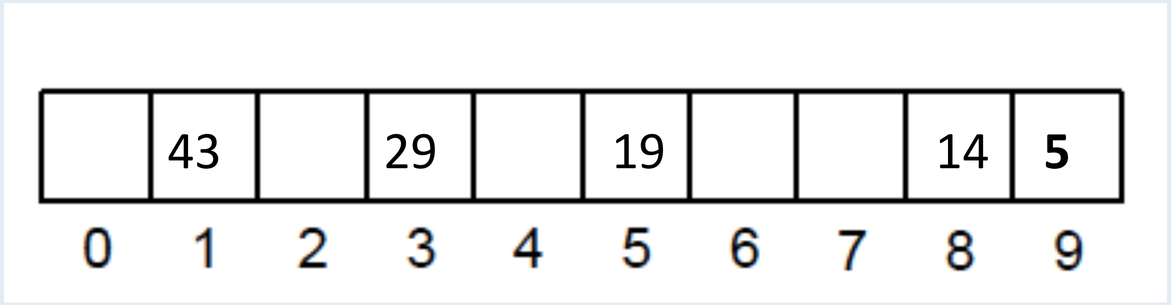
Insert values 14, 29, 43, 19, and **5** into the initially empty hash table:

- **5***117 = 5**8**5 bin ⇒ **5**, causing a collision (jump size given by the second digit, **8**)
- Where would this one end up?
  – 5 + 8 (modulo 10, of course) is 3, which is already taken, so we check 3 + 8, which is 1, also taken, then bin 9!



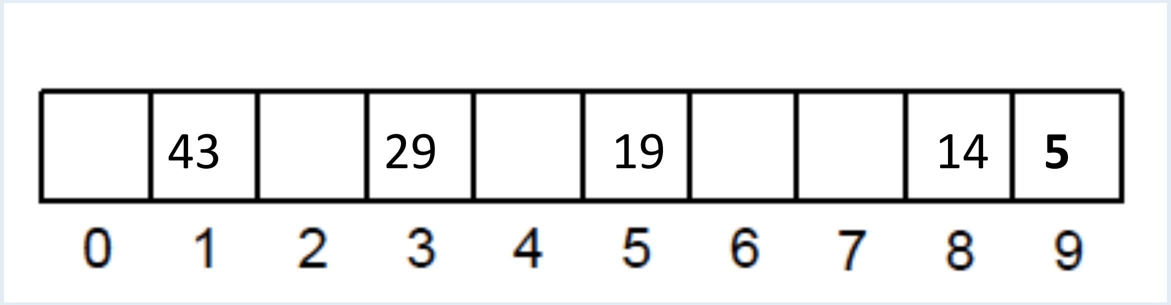
Nitty-Gritty

- There's a big (read: BIG!) problem with this.
- Let's try inserting **59**:
- **59***117 = 69**03** bin ⇒ **3**, causing a collision, so we choose jump size **0** ... Oops!
  - Fix this by not allowing the second hash function to take value 0? (could we do that with just an if? If the value is 0, just make it 1 — why not?)



Nitty-Gritty

- There's a big (read: BIG!) problem with this.
- But that's not all — let's try inserting **74**:
- **74***117 = 86**58** bin ⇒ **8**, causing a collision, so we get jump size = **5**, so we probe .... and... oops!
  - Why? How do we fix it?



Nitty-Gritty

- The problem is that the cycle is given by the least-common-multiple of the two values.
- We'd like it to be the product of the two numbers, but if the numbers are not relatively prime (i.e., share common factors), then the LCM is lower.

Nitty-Gritty

- The idea is that the jump size and the table size should be relatively prime (that is, they should have no common factors!)
  - This guarantees that every bin will be visited before cycling and start repeating bins.
  - And this is easy if one of the numbers (e.g., the table size) is prime (every number is relatively prime to a prime number, of course!)

Nitty-Gritty

- But a prime table size has several problems:
  - Modulo operations become expensive (can't take advantage of bit shifts and bit and operations)
  - Also, dynamically growing the size gets complicated and expensive (have to find prime numbers for the new size).

QUESTIONS TO PONDER

Questions to Ponder

1. What causes a collision?
   a) The program you are running crashes
   b) There are too many hash keys in the array
   c) Two hash keys are the same
   d) The program is out of memory
   e) None of the above

2. What are the three types of collision solutions?
   a) Overflow, underflow and undertow
   b) Chaining, overflow and probing
   c) Probing, underflow and chaining
   d) Noflow, fastflow and chaining
   e) None of the above

3. What does hashing improve?
   a) Speed
   b) Hard drive space
   c) All of the above

1. What does hashing do?
   a) Delete old files
   b) Create new files
   c) None of the above

2. How is a hash key computed?
   a) Long division
   b) Subtraction
   c) Random number generation
   d) Modulo division
   e) All of the above

Questions to Ponder

Consider inserting the keys 59, 10, 31, 88, 22, 4, 68, 28, 15, 34, 17 into a hash table of size *m* = 11 using open addressing with linear probing with the hash function *h(k)* = *k* mod *m*

Which filling in of the hash table below (indexing from 0 through 10) will result?

1. 88 22 68 17 59   4 28 34 15 31 10
2. 88 22 68 34 59   4 28 15 17 31 10
3. 88 34 68 15 59 17 28 22   4 31 10
4. 88 15 68   4 59 34 28 22 17 31 10

http://cse.unl.edu/~sscott/teach/Classes/cse156F06/Notes/Hash-In-Class-Quiz.ppt

Questions to Ponder

1. Circle all the open addressing approaches to hash table collisions:
   - a. Linear probing
   - b. Quadratic probing
   - c. Exponential hashing
   - d. Separate chaining

2. Circle one: What is the big-O complexity to retrieve from a hash table if there are no collisions?
   - a. O(c)
   - b. O(n)
   - c. O(n^2)
   - d. O(log n)

3. Circle one: What is the big-O complexity to insert n new elements into a hash table if there are no collisions?
   - a. O(c)
   - b. O(n)
   - c. O(n^2)
   - d. O(log n)

Questions to Ponder

4. Circle one: What is the disadvantage of using an array of even size as the basis for a hash table?

   a. Takes more space.
   b. Produces more collisions.
   c. Quadratic probing takes more time.
   d. Complicates item removal.

5. Circle all that are true: A hash function should have which properties?

   a. Uniform distribution.
   b. Speedy computation.
   c. Range is a subset of the integers.
   d. Equivalent objects produce equal hash codes