CSCI 2270 – CS 2: Data Structures

University of Colorado
Boulder

Topics

- Hashing
- Hash Tables
- Collision Resolution

Hashing

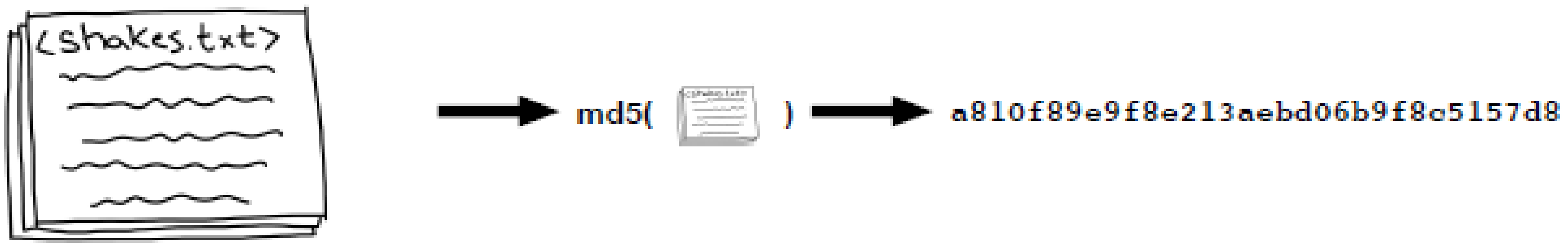
- BST can be found in O(log n) time for a well-balanced search tree.
- Is there a more efficient way to search for an element in a container?
  - Hashing
    - ***Requires data to be specially organized***
    - ***Use of a hash table (array)***
    - ***To determine whether a particular item with a key, say X, is in the table, we apply a hash function to the key.***
      - Compute h(X)
        - » Gives us the address of the item in the hash table
    - ***Items are stored in no particular order***

Hash Table

- aka Hash Map
- A data structure that stores data using a parameter in the data, called a key, to map the data to an index in an array.
- Hash tables use a “hash function” to squish a large range of key-values into a small range. It’s the array where records are stored.
- Instead of storing the key values directly at their index, hash-tables store them at the index of their hashed representation.

Hash Function

- Take something, do some math on it, and get a number
  - hash(my_node) = 7145205621
- The result 7145205621 is known as a hash code, hash function, or digest.



Input: likely large

e.g. collected works of Shakespeare (5.5 M)

Hash Function

Using md5 here, but there are many others as well.

Output: small

e.g. 120 bits

Hash Function Properties

- Same input -> same output
- Different inputs -> same output? (Collisions!)
- Similar inputs -> similar outputs?
- Given output -> reverse engineer input?

Hash Function

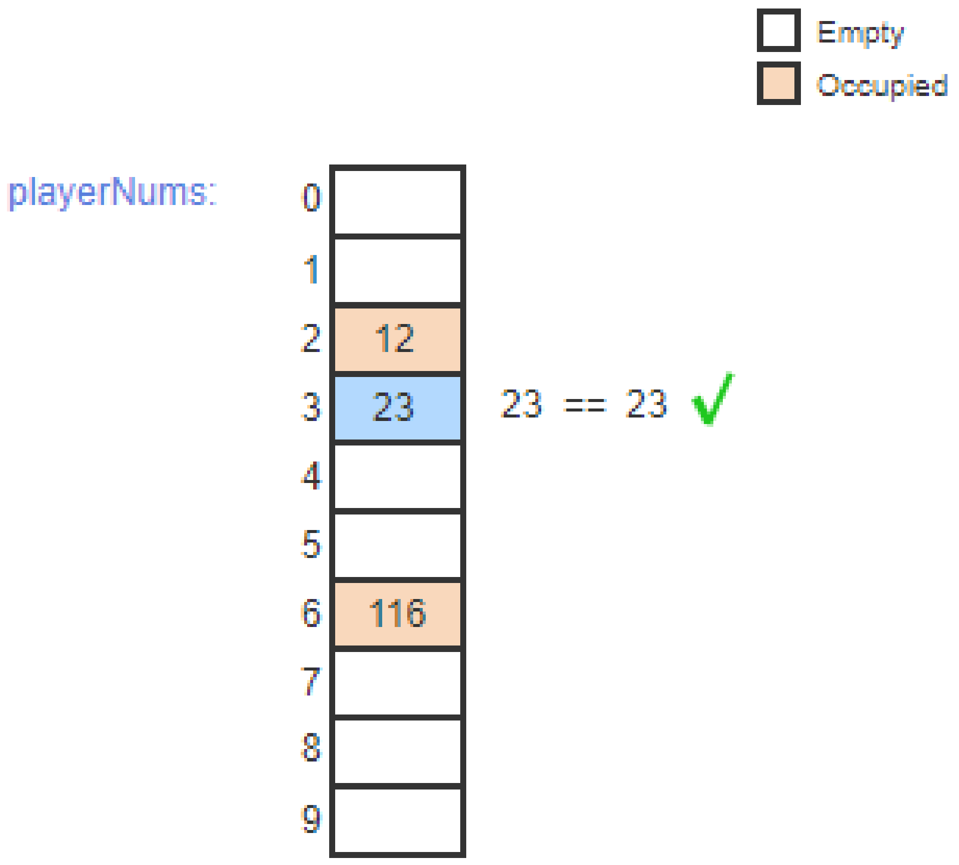
- Same Input, Same Output
- hash(a) always yields x
- Output only depends on input

Hash Function Example 1

```
playerNums = new HashTable(10)
HashInsert(playerNums, 116)
HashInsert(playerNums, 12)
HashInsert(playerNums, 23)
HashSearch(playerNums, 23)
```

Hash function: key % 10

23 % 10 = 3



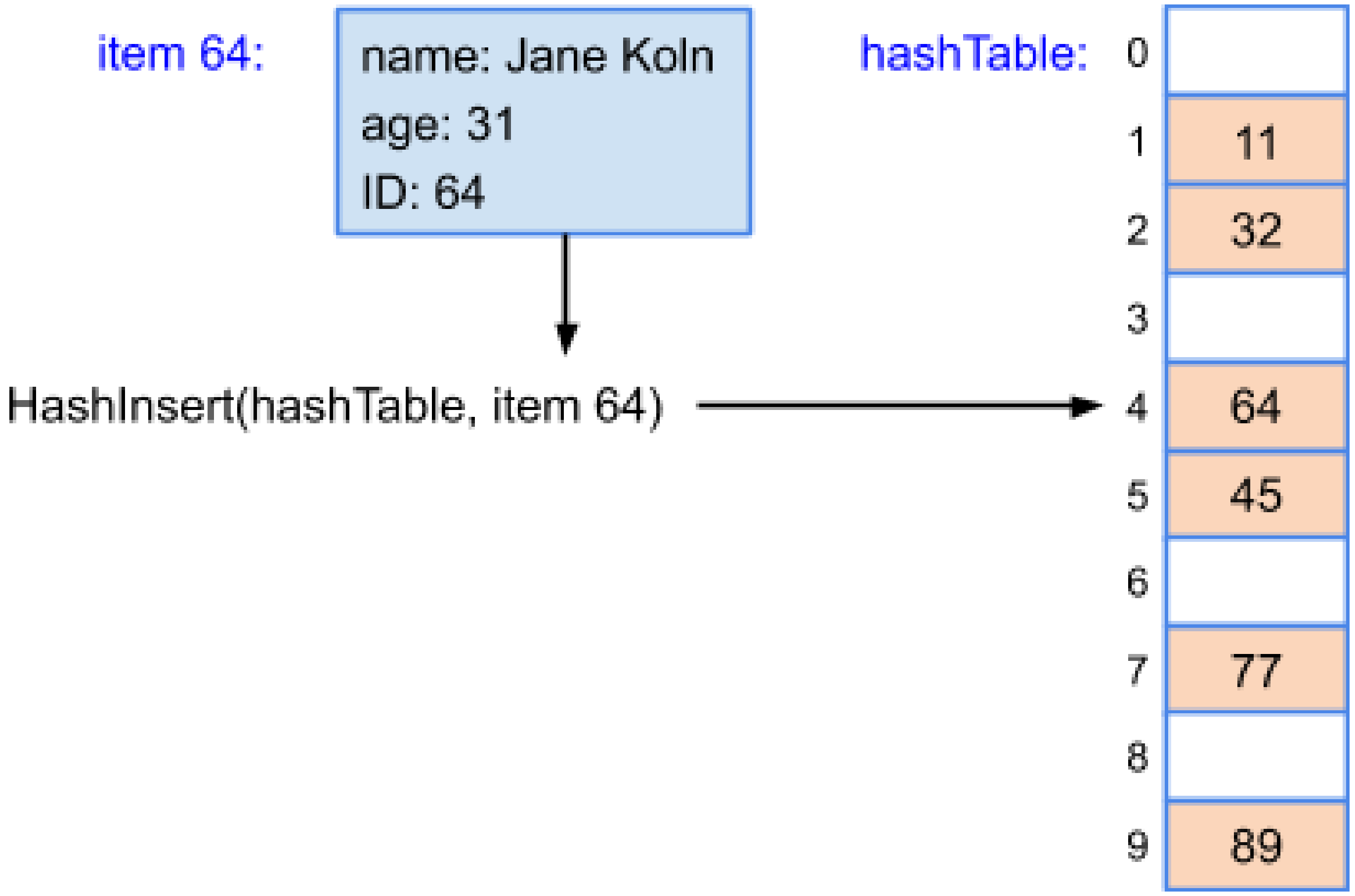
Hash Function Example 2

- Phone Book – store contacts (names of people) along with their phone numbers.
- Create an association (associative array/hash table)

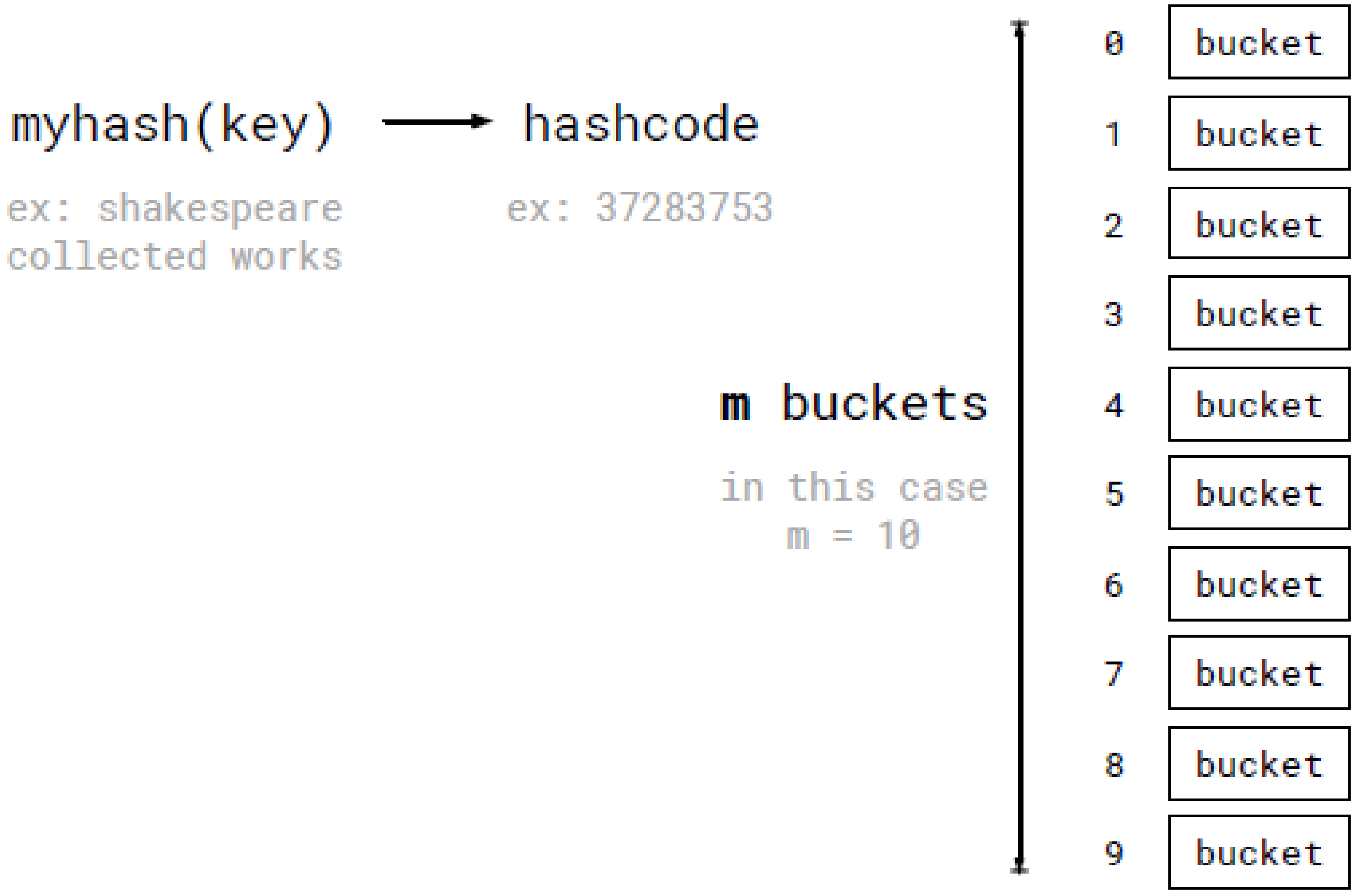
  name->phone number
- Remember, we're using a hash function to compute the index (aka hash code, aka bucket) where the phone numbers go.
- Let's say the name I want to store a number for is Asa and the phone number 123-456-7890. So, Asa is the key and 123-456-7890 is the value.

  Key: "Asa" ----> HF(Asa) ----> index 3 (store 123-456-7890 at [3])
- How is this beneficial? Well, if I want to find Asa's phone number in a 1 million item list, I can call the hash function again, pass the key, and retrieve the index value to retrieve Asa's phone number. This is done in O(1) without having to loop through 1,000,000 records.

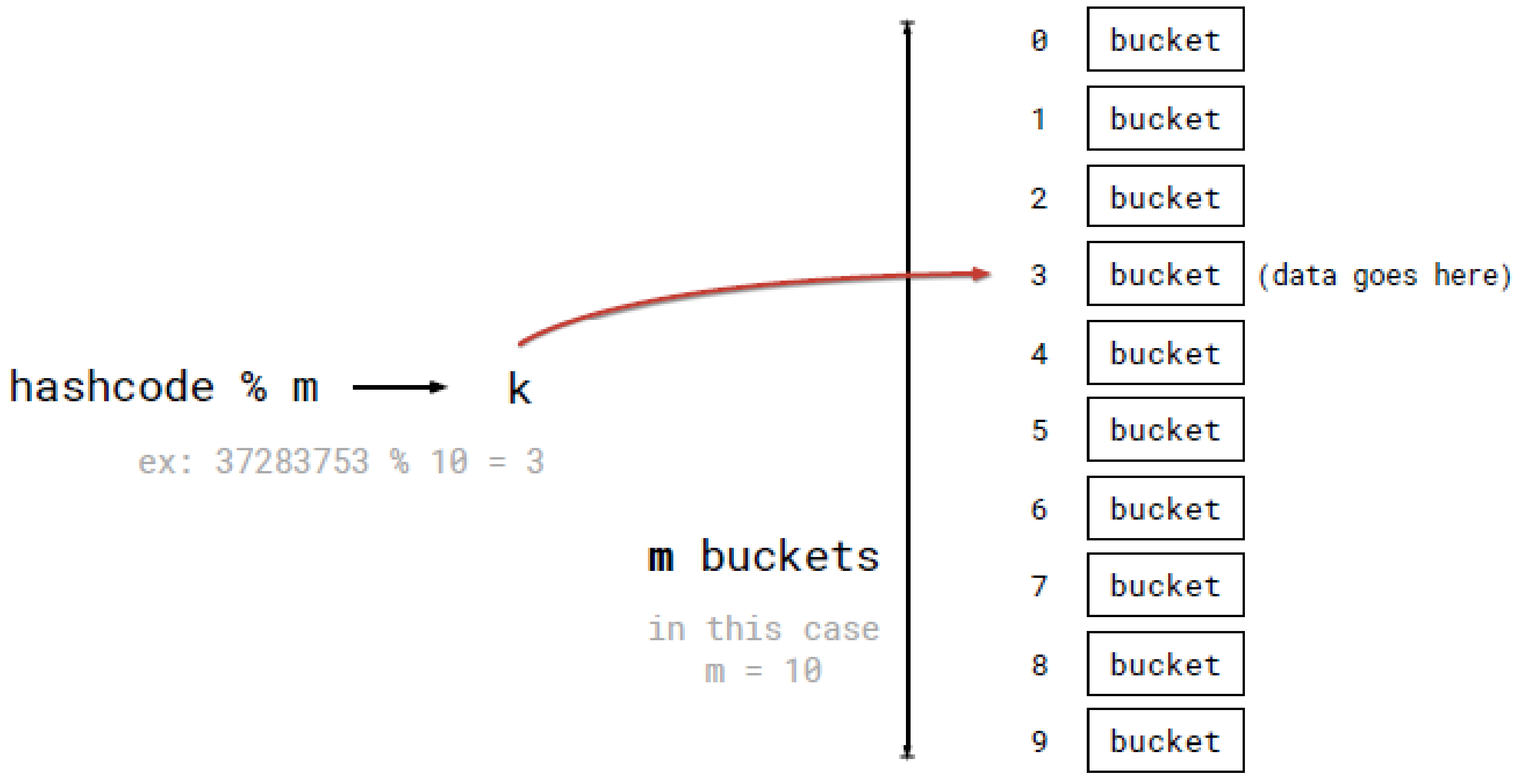
Hash Function Example 3



Hash Function Example 4

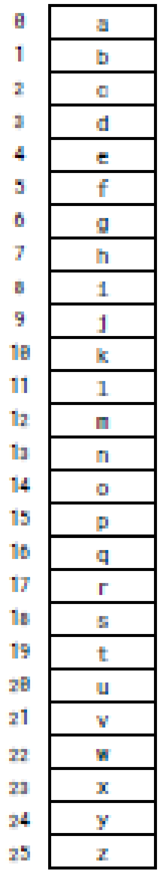


Hash Function Example 4



Hash Function Example 5

a = 97	h = 104	o = 111	v = 118
b = 98	i = 105	p = 112	w = 119
c = 99	j = 106	q = 113	x = 120
d = 100	k = 107	r = 114	y = 121
e = 101	l = 108	s = 115	z = 122
f = 102	m = 109	t = 116	
g = 103	n = 110	u = 117	



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p
16	q
17	r
18	s
19	t
20	u
21	v
22	w
23	x
24	y
25	z

Hash Function Example

word = xander

a = 97	h = 104	o = 111	v = 118
b = 98	i = 105	p = 112	w = 119
c = 99	j = 106	q = 113	x = 120
d = 100	k = 107	r = 114	y = 121
e = 101	l = 108	s = 115	z = 122
f = 102	m = 109	t = 116	
g = 103	n = 110	u = 117	

$$\begin{tikzcd}[row sep=large]
\texttt{hash\_func(word)} \arrow[d] \\
\texttt{hashcode} \arrow[d] \\
\texttt{bucket\_func(hashcode)} \arrow[d] \\
\texttt{bucket index}
\end{tikzcd}$$

hash_func(word): 32 bit code: first 4 letters next to each other in binary.

hashcode:

```
x: 120 = 01111000 binary
a: 97  = 01100001 binary
n: 110 = 01101110 binary
d: 100 = 01100111 binary
    ->
01111000011000010110000101100111
    ==
2019647847 in decimal
```

bucket_func(hashcode): value of first 8 bits minus 97

bucket index:

```
01111000 binary = 120 decimal
minus 97 gives us…
23
```

Hash Function Example

0	a	
1	b	
2	c	
3	d	
4	e	
5	f	
6	g	
7	h	
8	i	
9	j	
10	k	
11	l	
12	m	
13	n	
14	o	
15	p	
16	q	
17	r	
18	s	
19	t	
20	u	
21	v	
22	w	
23	x	X-Bucket
24	y	
25	z	

Applications of Hashing

- Data Structures
  - Applications we showed on the previous slides
- Message Digest
  - Cryptographic Hash Functions
- Password Verification

Applications of Hashing

- Cryptographic Hash Functions

MD5("The quick brown fox jumps over the lazy dog")

= 9e107d9d372bb6826bd81d3542a419d6

MD5("The quick brown fox jumps over the lazy dog.")

= e4d909c290d0fb1ca068ffaddf22cbd0

Applications of Hashing

- Password verification
  - Hash the password entered on a web site and send it to the server.
  - Server-side will compare the hashed password to the hashed password value on the server side. If they match, log in, else reject.

Encryption vs Hashing

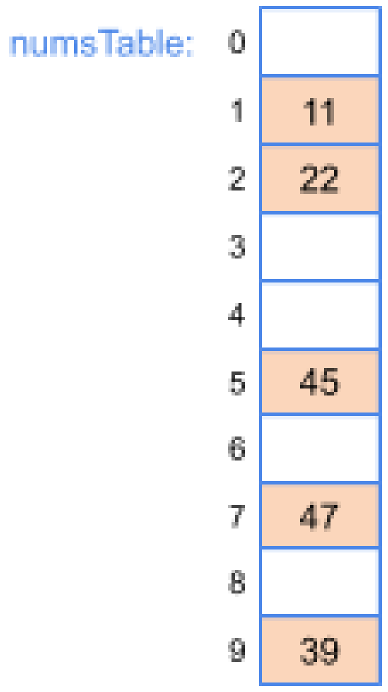
- Encryption is not the same thing as hashing
  - Encryption
    - ***Scrambling information using a cipher***
    - ***Two-way function***
    - ***Intention is to decrypt it later***
  - Hashing
    - ***Map data of any size to a fixed length***
    - ***Serves as a check-sum***
    - ***One-way function***

Hash Table Quiz

- Q1 - A 100 element hash table has 100 _____. Items or buckets?
- Q2 - A hash function computes a bucket index from an item's _____. Integer value or key?
- Q3 - For a well-designed hash table, searching requires _____ on average. O(1), O(N), O(log N)
- Q4 - A company will store all employees in a hash table. Each employee item consists of a name, department, and employee ID number. Which is the most appropriate key? Name, Department, Employee ID number

Hash Table Search Efficiency Quiz

- Q1 - How many buckets will be checked for HashSearch(numsTable, 45)?
- Q2 - If item keys range from 0 to 49, how many keys may map to the same bucket?
- Q3 - If a linear search were applied to the array, how many array elements would be checked to find item 45?



Collisions

- Hash(a) always yields x
- Hash(b) always yields z

- Usually x != z
- Sometimes x = z (Collision!)

- **Collision** - When two or more hash keys result in the same hash value

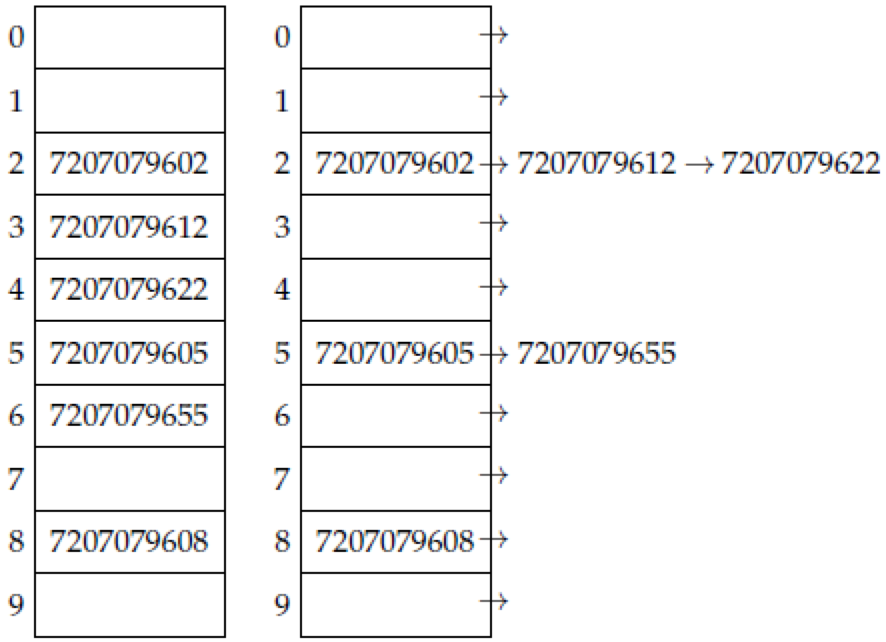
- A perfect hash would encounter no collisions and perfectly fill a hash table.

Collisions

- Perfect hash functions are unrealistic, so we need collision minimization and collision resolution.
- Collision Types
  - Hash code collisions
  - Bucket collision (we will focus on this)

Collisions

- Open addressing: If a collision occurs, add a record at some other available location.
  - ***Linear Probing***
  - ***Quadratic Probing***
  - ***Double hashing***
- Chaining
  - ***Contains a data structure (probably a linked list)***



Collisions – Open Addressing

- Linear probing finds the next available location and stores the record there (in linear order)
  - i.e. h(x), h(x) + 1, h(x) + 2
  - Assume that the array is circular so that if the lower portion of the array is full, we can continue the search in the top portion of the array (accomplished via mod operator).
  - Starting at *t*, check array locations using probe sequence
    - *t, (t + 1) % HTSize, (t + 2) % HTSize, . . ., (t + j) % HTSize*

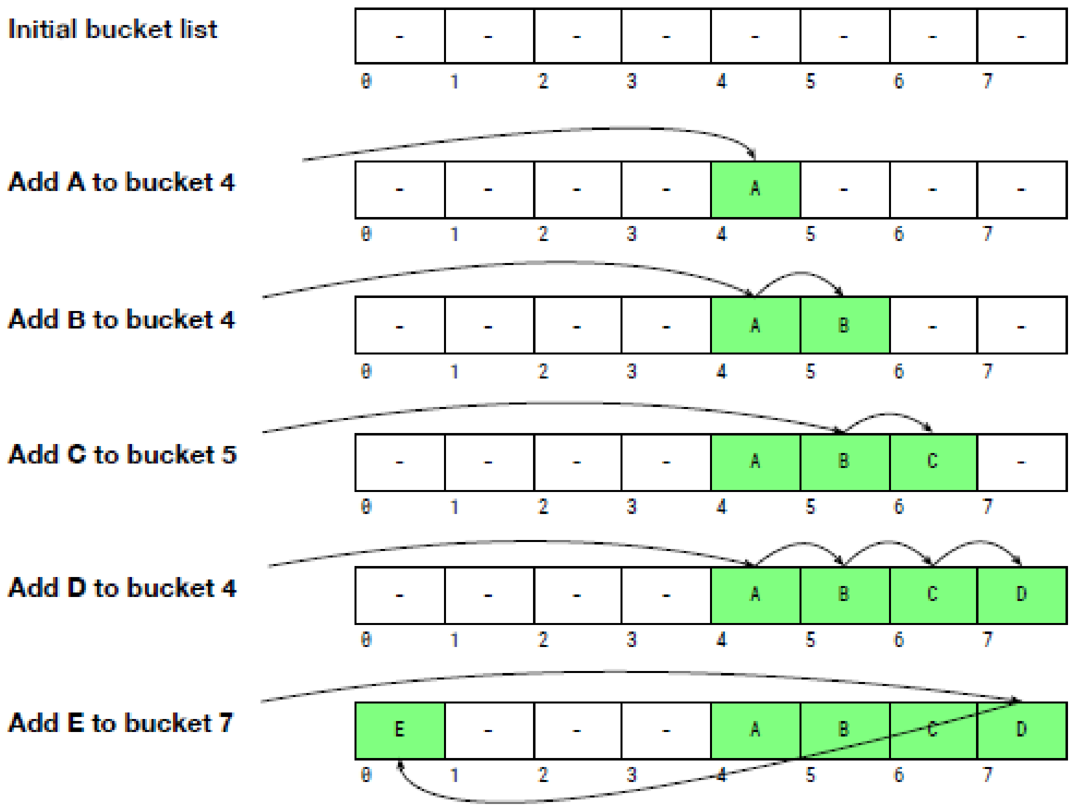
Collisions – Open Addressing

- Linear probing example

ID	h(ID)	(h(ID) + 1) % 13	(h(ID) + 2) % 13
197354864	5		
933185952	10		
132489973	5	6	
134152056	12		
216500306	9		
106500306	3		
216510306	12	0	
197354865	6	7	

Collisions – Open Addressing

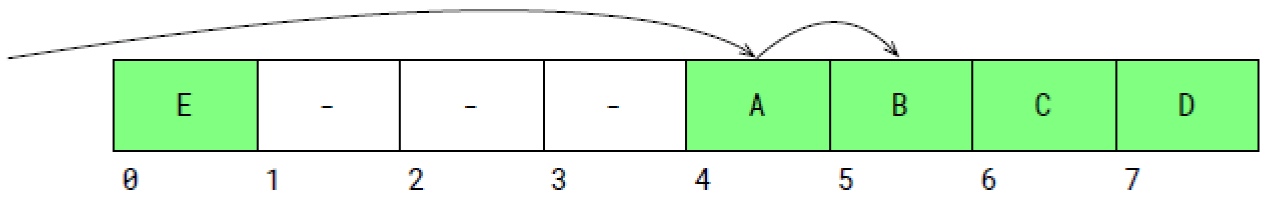
- Linear probing - another example of insertion



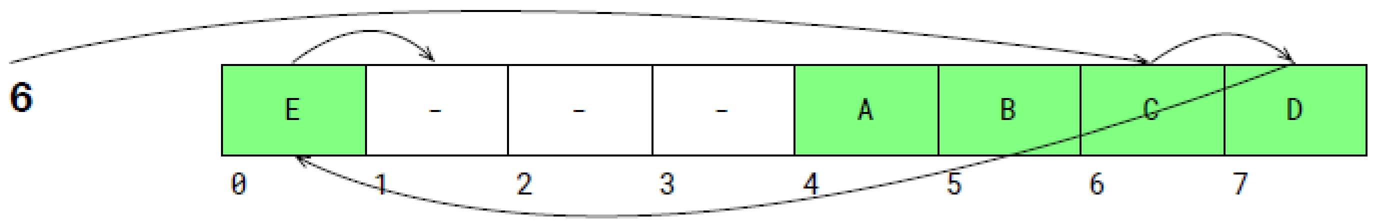
Collisions – Open Addressing

- Linear probing – example of finding a value

Find B in bucket 4



Find Q in bucket 6

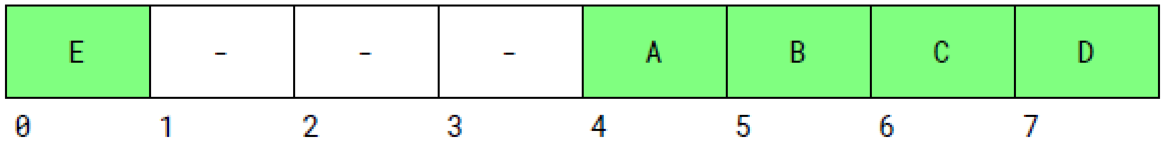


University of Colorado **Boulder**

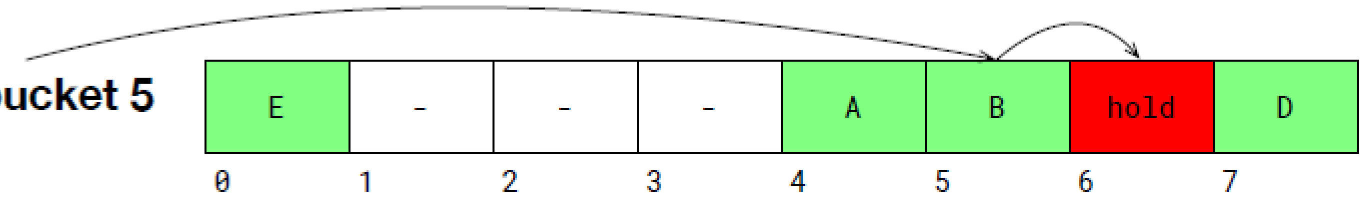
Collisions – Open Addressing

- Linear probing – example of removing a value

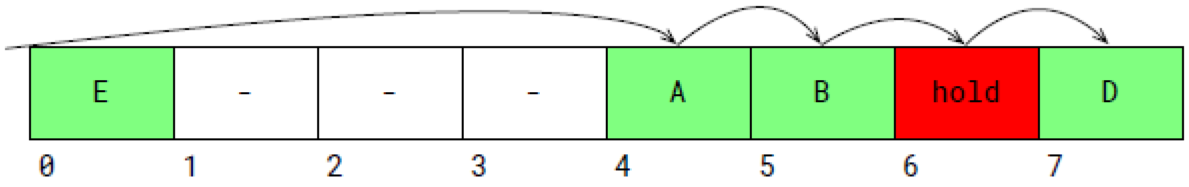
Current state



Remove C from bucket 5



Find D in bucket 4



Collisions – Open Addressing

- Linear probing disadvantage
  - Clustering: this is where groups of consecutive cells in the hash table are occupied. Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry. As clusters grow in size, they may merge into even larger clusters, further slowing down the search time.

Collisions – Open Addressing

- Example of clustering

- The probe sequence of this linear probing is:

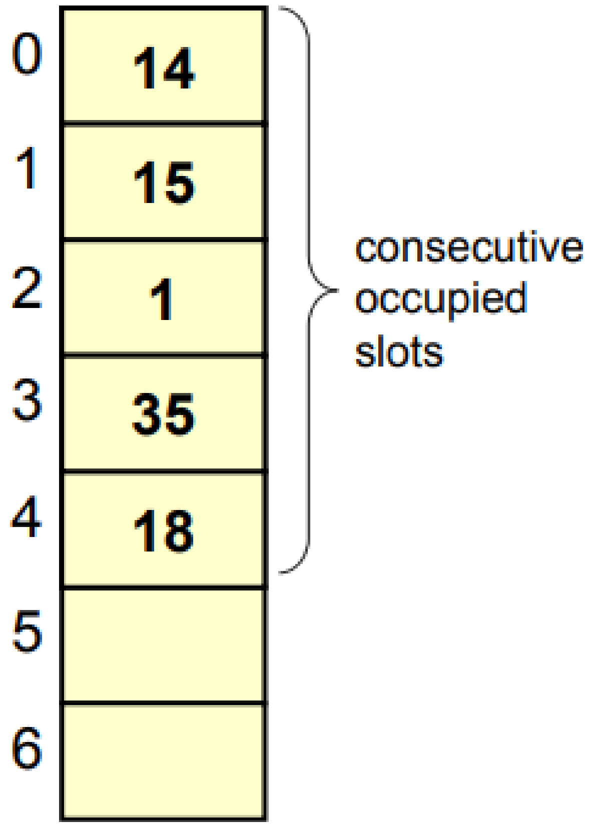
hash(key)

(hash(key) + 1) % m

(hash(key) + 2) % m

(hash(key) + 3) % m

- O(1) -> O(n)



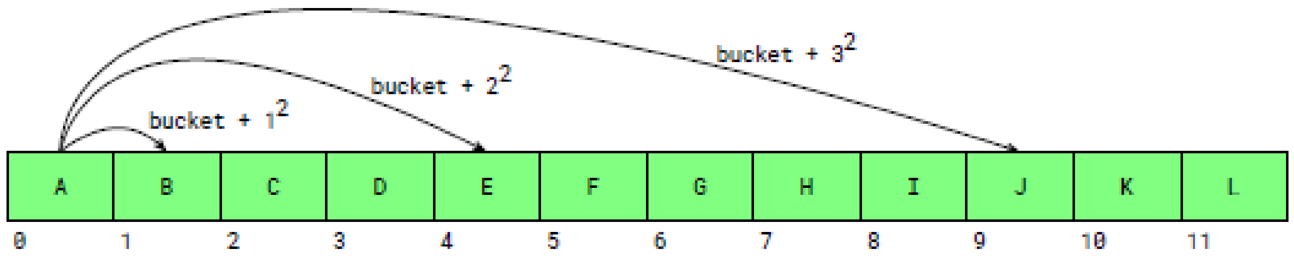
Collisions – Open Addressing

- Quadratic probing: Instead of looking at the next adjacent slot, skip over by i^2 indices. In other words, find the next free position in the array in quadratic order.
  - i.e. $h(x)$, $h(x) + 1^2$, $h(x) + 2^2$
  - Avoids clustering that existed in linear probing

Collisions – Open Addressing

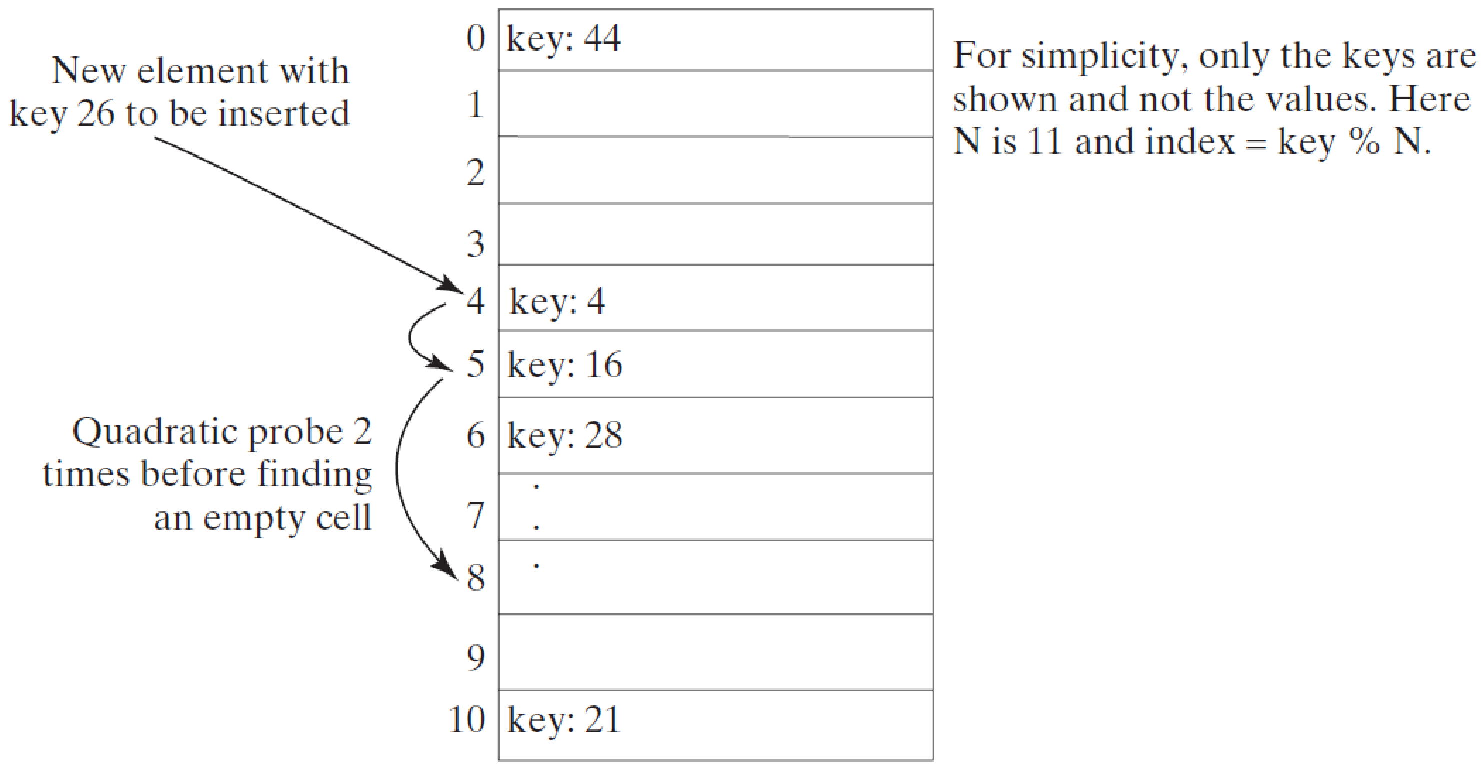
- Quadratic probing example

Find J in bucket 0



Collisions – Open Addressing

- Quadratic probing – another example



New element with key 26 to be inserted
Quadratic probe 2 times before finding an empty cell
0 key: 44
1
2
3
4 key: 4
5 key: 16
6 key: 28
7
8
9
10 key: 21
For simplicity, only the keys are shown and not the values. Here N is 11 and index = key % N.

Collisions – Open Addressing

- Quadratic probing disadvantage
  - Secondary clustering
    - ***Although quadratic probing avoids linear probing's clustering problem, it has its own clustering problem, called secondary clustering. In secondary clustering, the entries that collide with an occupied entry use the same probe sequence.***
    - ***Note: secondary clustering is not as bad as primary clustering.***

Collisions – Open Addressing

- We want to avoid clustering of any kind. Let's look at double hashing.
- Double hashing: Find the next free position in the array using 2 different hash functions.
  - i.e. $h^1(x) + i * h^2(x)$

- hash(key)
- (hash(key) + 1 * hash_2(key)) % m
- (hash(key) + 2 * hash_2(key)) % m
- (hash(key) + 3 * hash_2(key)) % m

- hash_2 is called the secondary hash function. The result of the second hash function will be the number of positions form the point of collision to insert.

Collisions – Open Addressing

- A couple of rules first…
  - The second function must:
    - ***Never evaluate to 0 (can't increment 0)***
    - ***Make sure that all cells can be probed***

- A popular second hash function is:
  $hash_2$(key) = R - (key % R)
  where R is a prime number that is smaller than the size of the table.

- So, we will pick an R. Assume the array size (number of buckets is 11). Let's pick R = 7. Thus, **$hash_2$(key) = 7 – (key % 7)**. That means for hash1, we will use **$hash_1$(key) = key % 11**.

- Let's look at an example of how to apply $hash_2$…

Collisions – Open Addressing

- Remember though for a couple of slides ago…this is the probe sequence we're going to use:

hash(key)

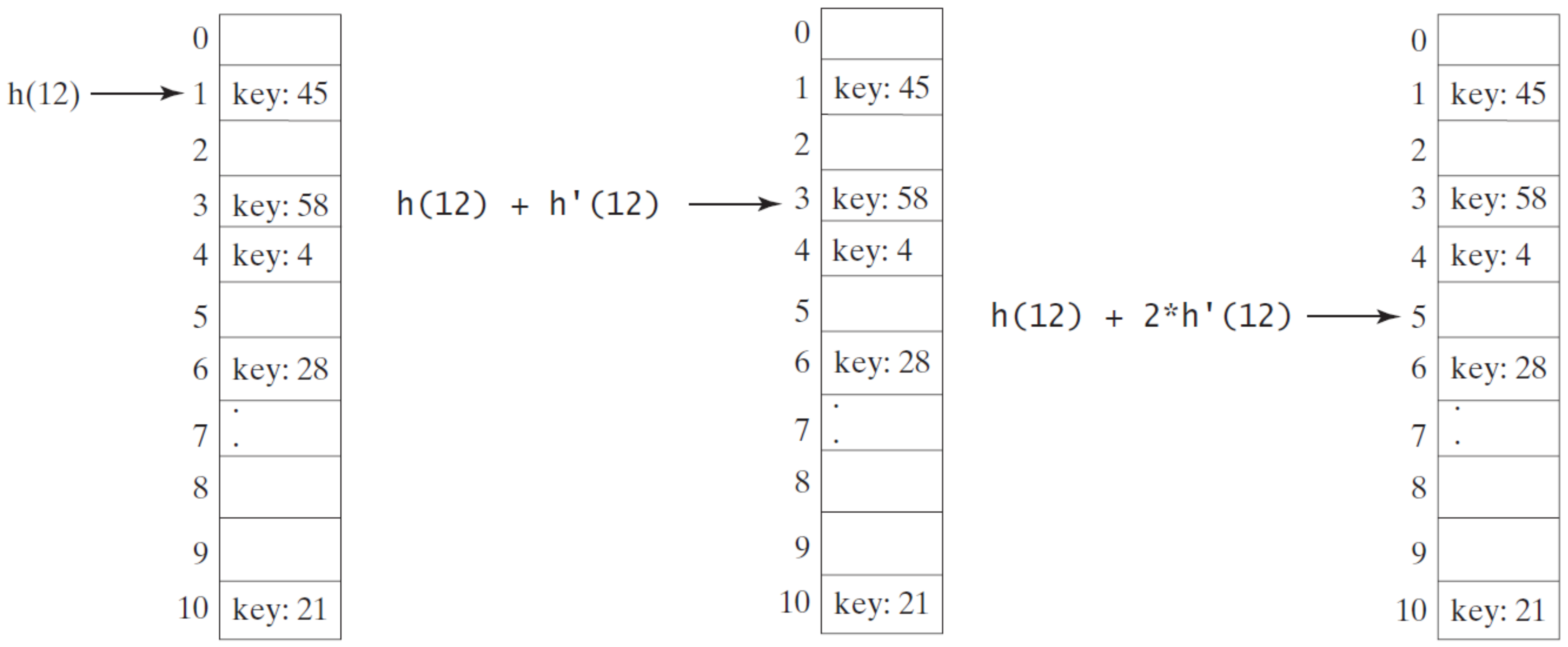
(hash(key) + 1 * $hash_2$(key)) % m

(hash(key) + 2 * $hash_2$(key)) % m

(hash(key) + 3 * $hash_2$(key)) % m

Collisions – Open Addressing

- Double Hashing Example

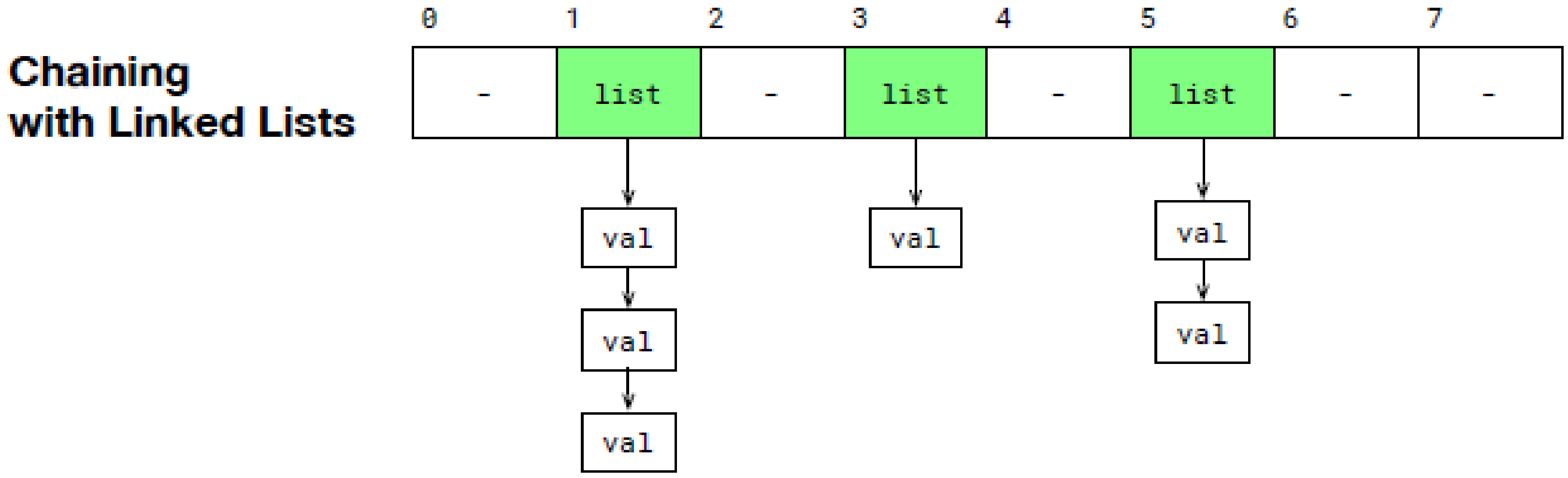


Collisions – Open Addressing

- **Advantages and Disadvantages**
  - Linear probing
    - ***best cache performance***
    - ***suffers from clustering***
    - ***easy to compute***
  - Quadratic probing
    - ***lies between the two in terms of cache performance and clustering***
  - Double hashing
    - ***poor cache performance but no clustering***
    - ***requires more computation time as two hash functions need to be computed***

Collisions – Chaining

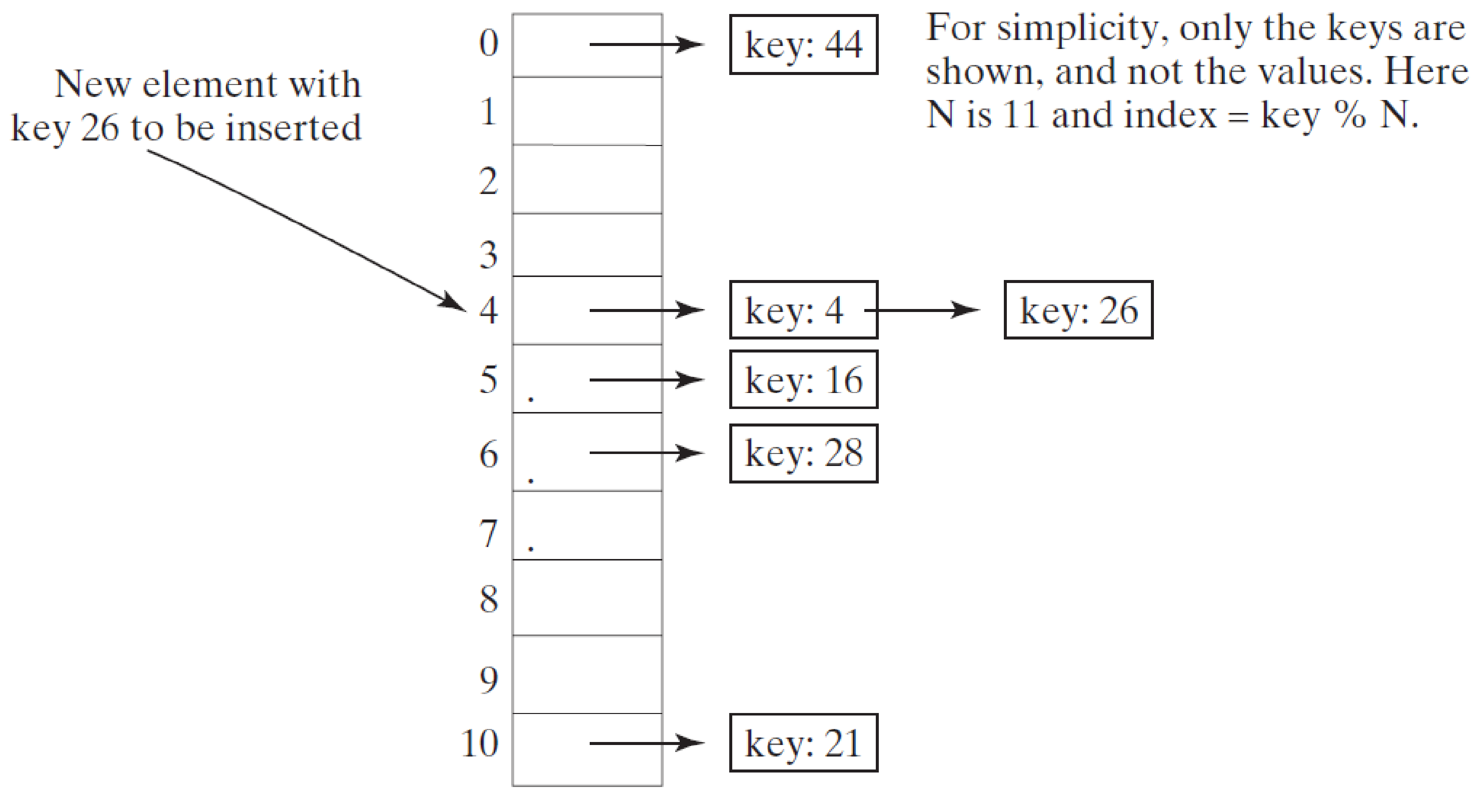
- Every element is a pointer to a linked-list head.



- The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

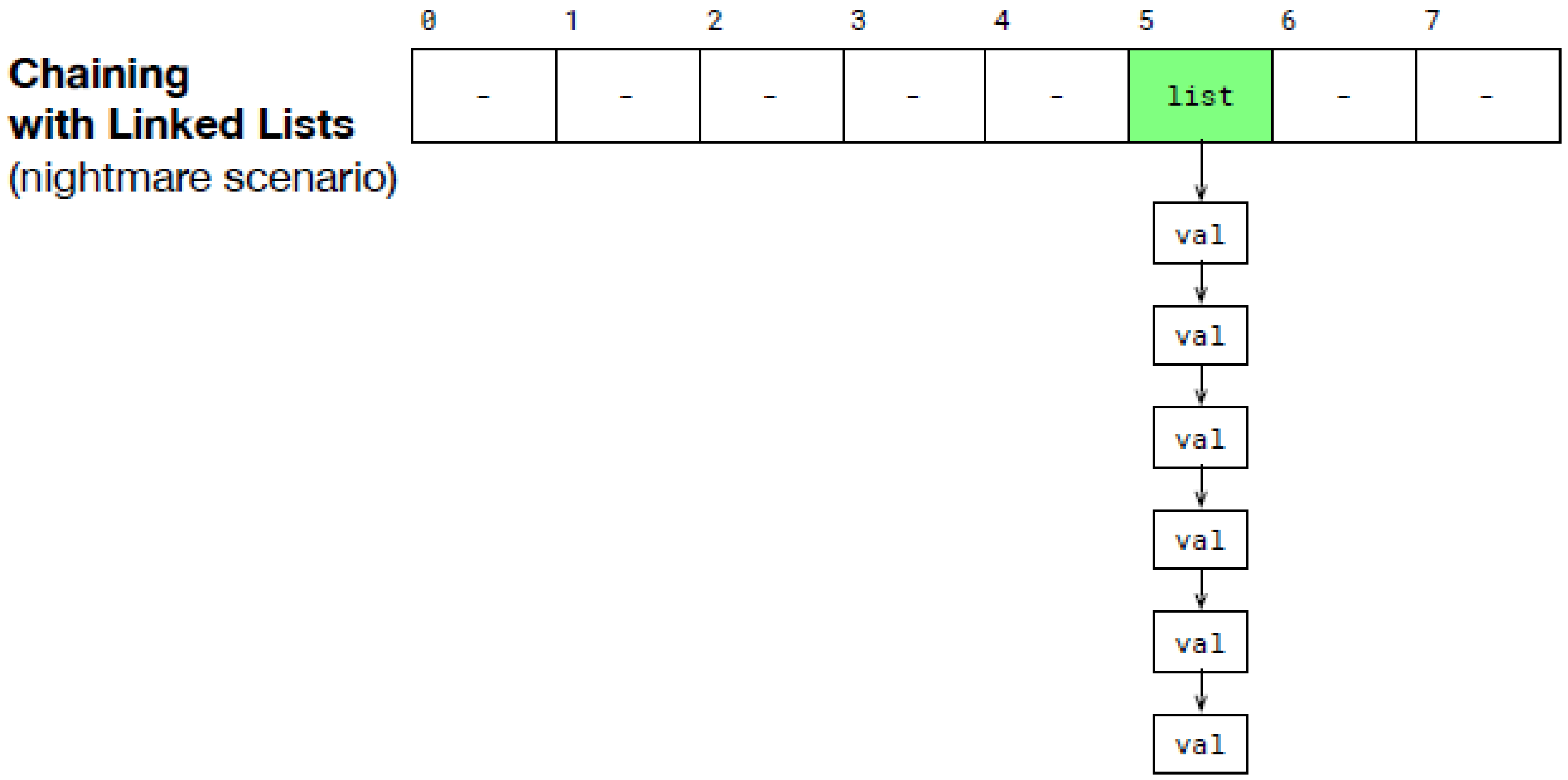
Collisions – Chaining

- Another visual


New element with key 26 to be inserted
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 26
key: 16
key: 28
key: 21
For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

Collisions – Chaining

- Bad scenario



Questions?