Data Structures

CSCI 2270-202: REC 12

Sanskar Katiyar

Logistics

Office Hours (Zoom ID on Course Calendar)

Wednesday: 3 pm – 5 pm

Thursday: 5 pm – 6 pm

Friday: 3 pm – 5 pm

Recitation Materials *(Notes, Slides, Code, etc.)*

sanskarkatiyar.github.io/CSCI2270

Recitation Outline

1. Hashing: Introduction
2. Hash Table, Hash Function
3. Collision Resolution
4. Exercise

Hashing: Introduction

Hashing: Introduction

A technique for *fast* insertion (storage) and search (retrieval)

Time Complexity for Search, Insertion:

- Average-case: **O(1)**, *under certain (reasonable) assumptions*
- Worst-case: **O(N)**

Allows maintaining data as (*key*, *value)* pairs over arbitrary keys

Hashing: Introduction

Key-Value pair

Use a key to identify/organize the data item

jodo0000

Identi(**key**)

Identikey: jodo0000
Name: John Doe
Degree: BS
Major: Computer Science
Minor: Psychology
. .
.

Student Record (**value**)

Hashing: Example (1)

Given a string S, find the first character that repeats (l-to-r).

`In "abcdabcdabcd", 'a'` repeats first

Approach #1: Use an array

```
for(i = 0; i < S.size() - 1; i++)
    for(j= i + 1; j < S.size(); j++)
        if(S[i] == S[j]) return S[i];
```

Time: **O(N²)**

Hashing: Example (1)

Approach #2: Use Hashing

```
Maintain an array A[256]
// 1 place for each ASCII character code

Initialize A[i] = 0, for all i

for(i = 0; i < S.size(); i++)
    A[(int) S[i]]++;
    if(A[(int) S[i]] > 1) return S[i];
```

Time: **O(N)**

Hashing: Example (1)

Approach #2: Use Hashing

Key: character

Value: count of character

By maintaining a table and a map, we were able to reduce the retrieval to O(1) and thus the overall time complexity to O(N)

Hashing: Example (2)

LeetCode (TwoSum)

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

```
Given nums = [2, 7, 11, 15], target = 9,
Since nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

Hashing: Introduction

Common Operations

`Search(key)`: Search for the *(key, value)* record identified by *key*

`Insert(key, value)`: Insert *(key, value)* record identified by *key*

`Remove(key)`: Delete *(key, value)* record identified by *key*

Key Components

Hash Table (or Map)

Hash Function

Collision Resolution

Hash Table, Hash Function

Hashing: Hash Table

Hash Table (or Hash Map)

- Stores the (key, value) record
- Generalization of an array
- Uses a hash function to map the key to the associated value

Hash function ***h*: *X*** → ***Y***, *where Y in [0, table_size) and Z*

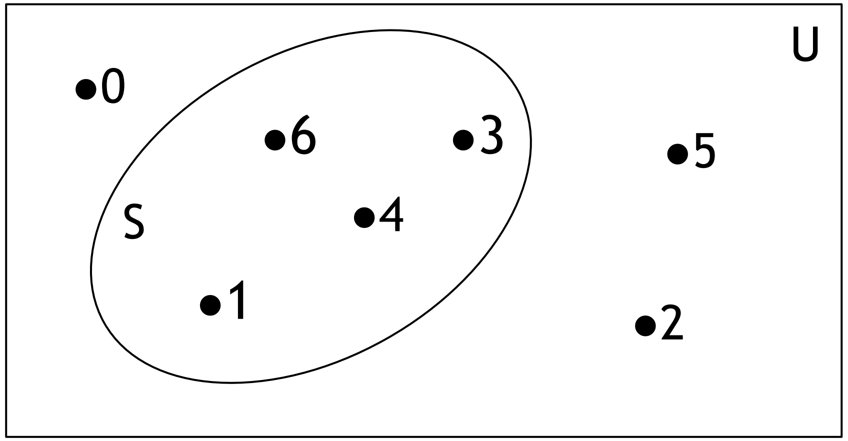
Utilize a hash table when

- *number of keys* << *number of possible keys*

Hashing: Direct Addressing

$$h(x) = x$$

	1		3	4		6
0	1	2	3	4	5	6



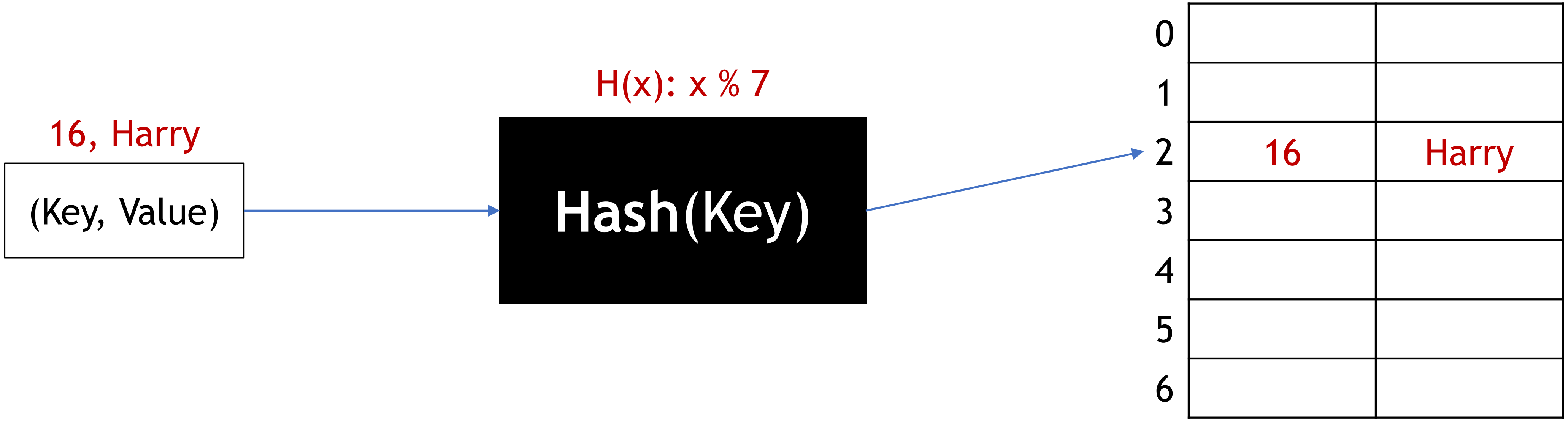
Direct Addressing is useful when **set of all *possible* keys is (finite) and *small***

Each **key identifies a unique record**

Direct Address Tables
(Arrays: O(1) Access)

Hashing: Hash Function

A hash function maps the key to an appropriate integer index in the hash table.



Hashing: Hash Function

Properties of a good Hash function

- Quick Computation, O(1)
- Distribute key values uniformly across the table
- Have a high load factor for a given set of keys
- Minimize collision
- Allow efficient collision resolution

Hashing: Hash Function

Example: *String*

h(x) = *(Sum of ASCII values of all characters)* **%** *113*

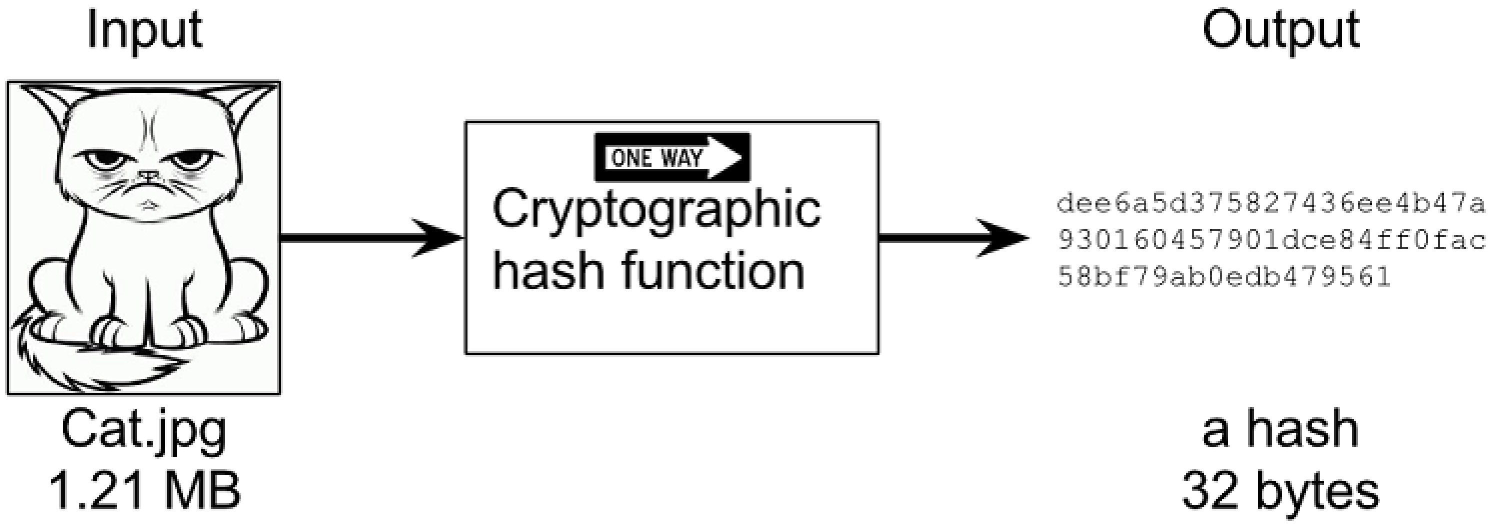
```
h(“Hashing”)  =>   706 % 113   => 28
h(“Table”)    =>   488 % 113   => 36
h(“2270”)     =>   203 % 113   => 90
```

Hashing: Hash Function

Applications in **Cryptography**

Message Digest for Authenticity

Password Storage and Verification



Hashing: Load Factor

$$Load\ Factor = \frac{Number\ of\ Elements\ in\ the\ Hash\ Table}{Size\ of\ Hash\ Table}$$

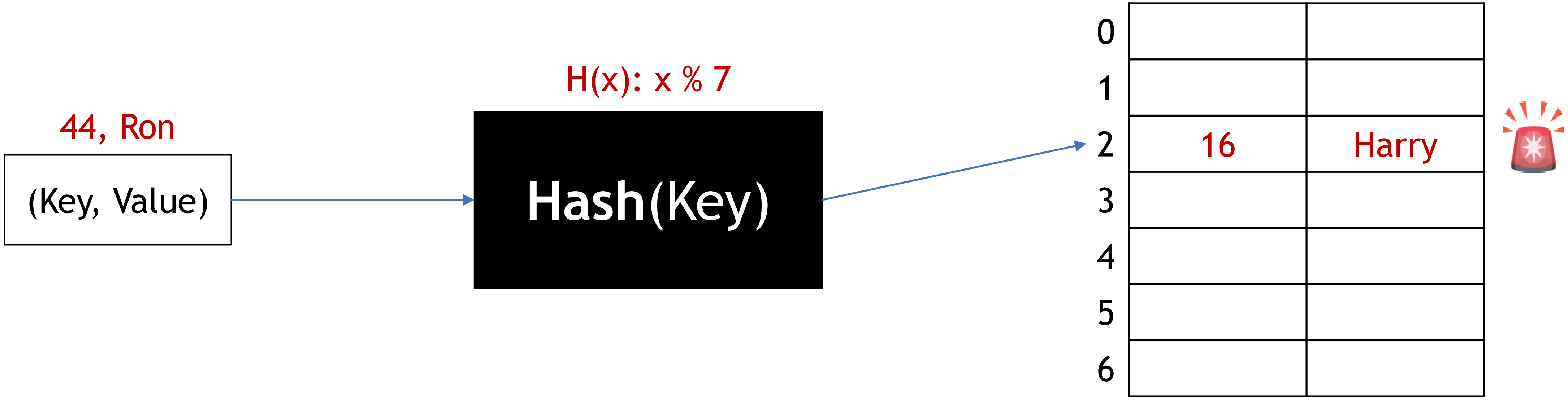
Determines the efficiency of hash functions

Serves as a decision parameter, whether to rehash or expand existing hash table entries

Collision

Hashing: Collision

Collision: When a hash function generates the same hash for two different key inputs



Hashing: Collision Resolution

Collision Resolution: Finding an alternate location for some entry *x*, where *h(x)* causes a collision in the hash table

But then hashing is **no longer O(1)** search/insert/delete
Yes! **But still much better than O(N)**, on average

We will discuss 2 of N techniques here:

- **Linear Probing** (Open Addressing)
- **Separate Chaining** (Direct Chaining)

Collision Resolution: Linear Probing

Open Addressing

All elements are stored in the hash table

`table.size()` >= Number of keys

Linear Probing

If index `[h(x) % S]` is full, then attempt insertion at `[(h(x) + 1) % S]`

If index `[(h(x) + 1) % S]` is full, then try `[(h(x) + 2) % S]...`

```
rehash(x) = [h(x) + (const) step_size] % table.size()
```

`h(x)` `S`

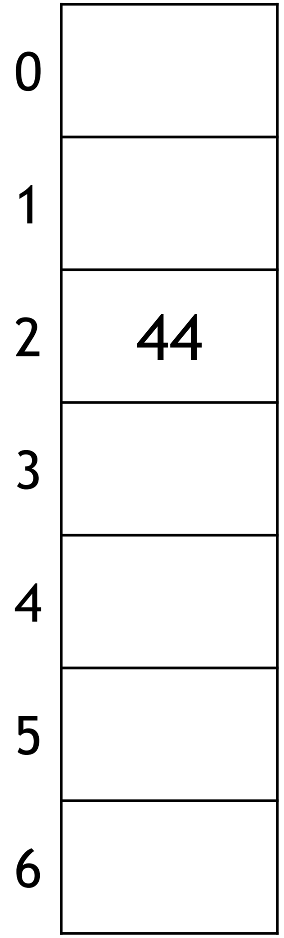
Collision Resolution: Linear Probing

Key	Key % 7	Hash

0	
1	
2	
3	
4	
5	
6	

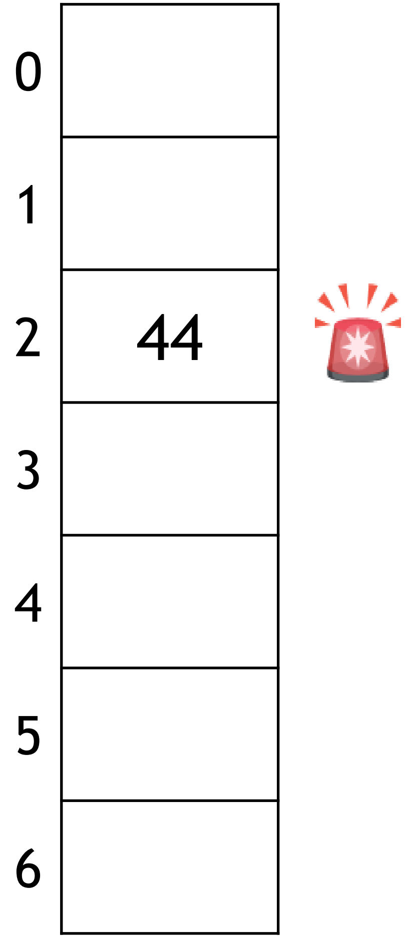
Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2



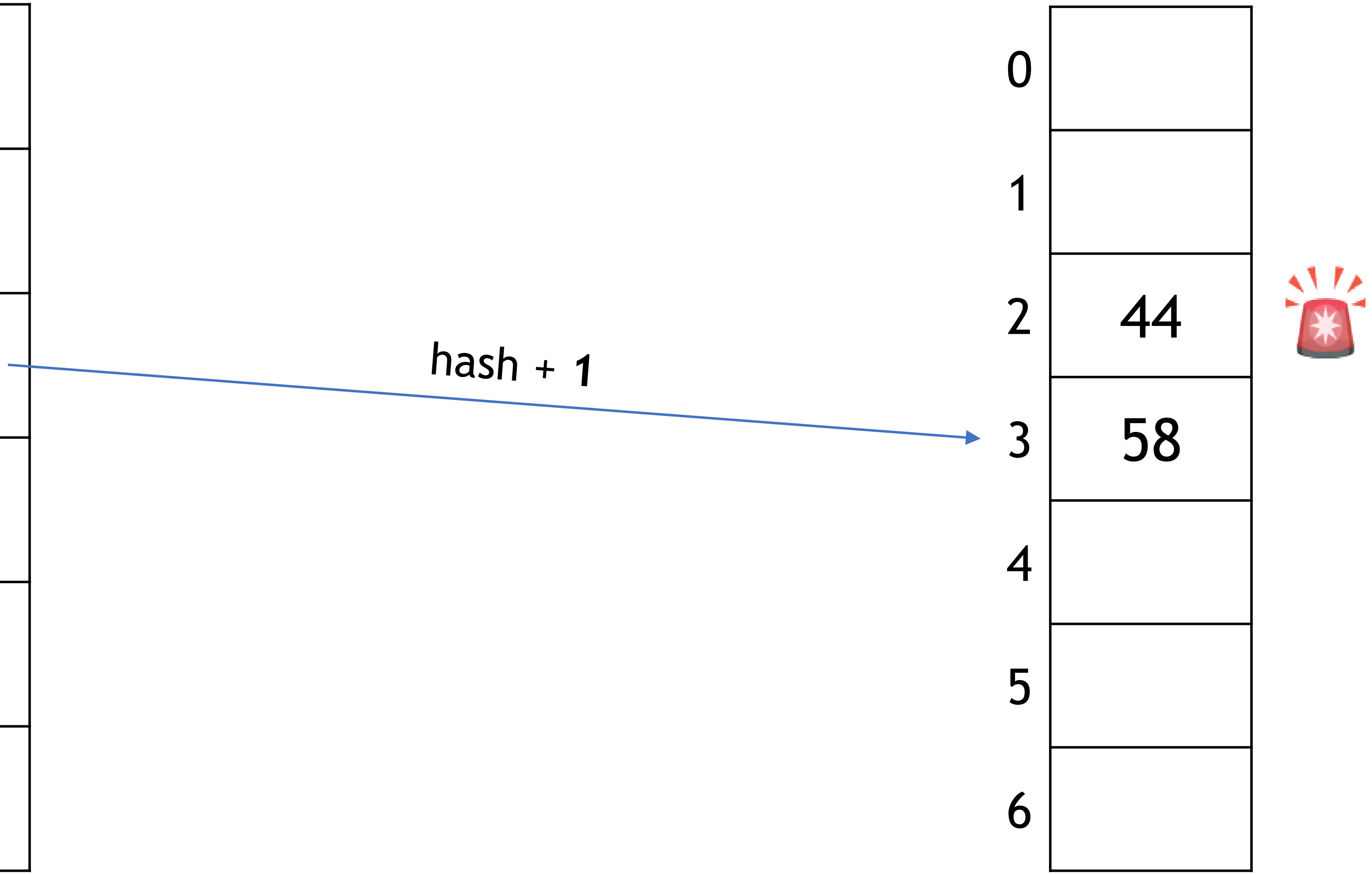
Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2



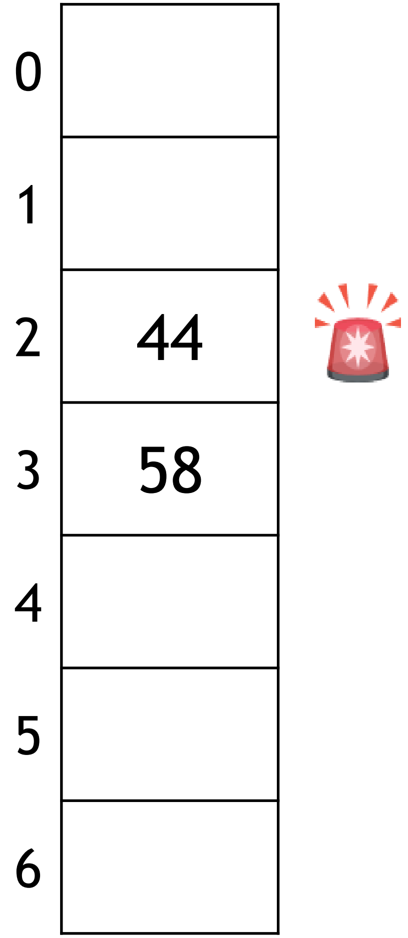
Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2



Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2



Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2

hash + 1

0	
1	
2	44
3	58
4	
5	
6	

Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2

hash + 2

0	
1	
2	44
3	58
4	16
5	
6	

Collision Resolution: Linear Probing

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2
7	7 % 7	0

0	7
1	
2	44
3	58
4	16
5	
6	

`step_size = 1`

Collision Resolution: Linear Probing

Insertion

```
insert(x, T=table) {
for(i = 0; i < T.size(); i++) {
  if T[(h(x) + i) % T.size()].isEmpty()
  {
    T[(h(x) + i) % T.size()] = x;
    return SUCCESS_INSERTION;
  }
}

return ERROR_TABLE_FULL;
}
```

Search (Item Exists)

```
search(x, T=table) {
for(i = 0; i < T.size(); i++) {
  if T[(h(x) + i) % T.size()] == x
  {
    return TRUE; // or x.data
  }
}

return FALSE;
}
```

Collision Resolution: Linear Probing

Clustering

Items tend to cluster together in the table (***retrieval time***)

If clusters merge, certain portions are densely occupied while others are relatively empty ***(wasted space)***

Role of step_size

Cannot resolve clustering by simply increasing step_size

Choose step_size such that **step_size** and **table.size() are co-primes**

But **table.size() can't change**

Collision Resolution: Quadratic Probing

Linear Probing

`rehash(x) = [h(x) + (const) step_size] % table.size()`

Quadratic Probing

If index `[h(x) % S]` is full, then try $[h(x) + 1^2] \% S$

If index $[(h(x) + 1^2) \% S]$ is full, then try $[(h(x) + 2^2) \% S]$...

$$\boxed{\texttt{rehash}(x) = [h(x) + (\text{const})\ \texttt{step\_size}^2] \ \% \ \texttt{table.size()}}$$

h(x)

Collision Resolution: Double Hashing

Use 2 separate hash functions: h_1, h_2

If index $[h_1(x) \% S]$ is full, then try $[h_1(x) + 1*h_2(x)] \% S$

If index $[(h_1(x) + k*h_2(x)) \% S]$ is full, then try $[h_1(x) + (k+1)*h_2(x)] \% S$

. . .

```
rehash(x) = [h1(x) + step_size * h2(x)] % table.size()
```

Collision Resolution: Direct Chaining

Separate Chaining

Each index of hash table stores a Linked List

The corresponding Linked List stores the items

Direct Chaining

Insert item *x* in the Linked List at index *h(x)* of the table

Insert item at head of Linked List ***(Why?)***

Search is O(N), *N*: length of the Linked List

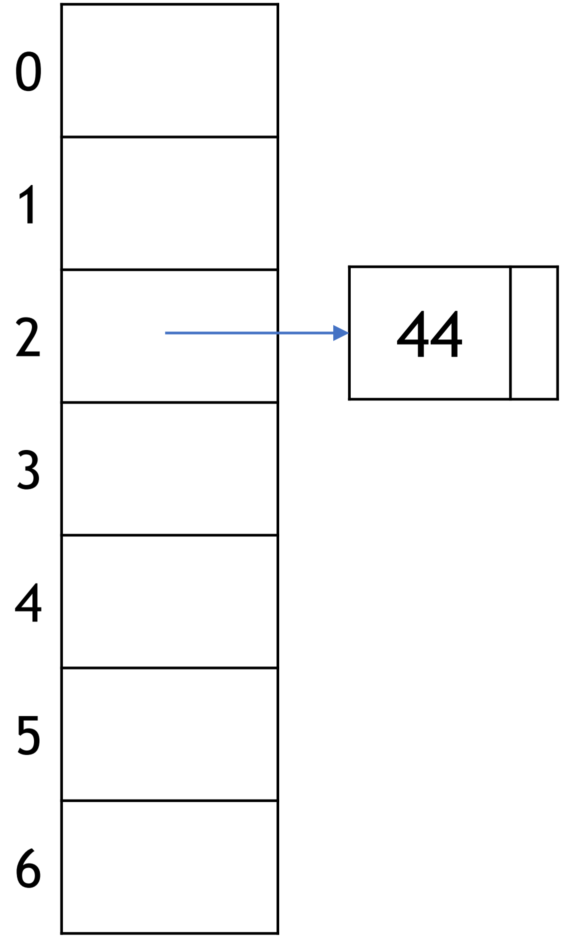
Collision Resolution: Direct Chaining

Key	Key % 7	Hash

0	
1	
2	
3	
4	
5	
6	

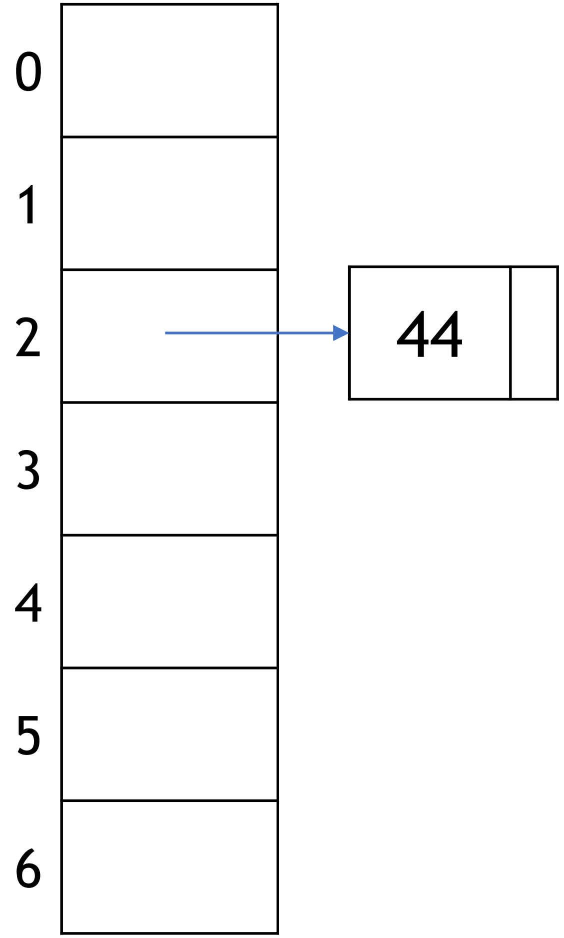
Collision Resolution: Direct Chaining

Key	Key % 7	Hash
44	44 % 7	2



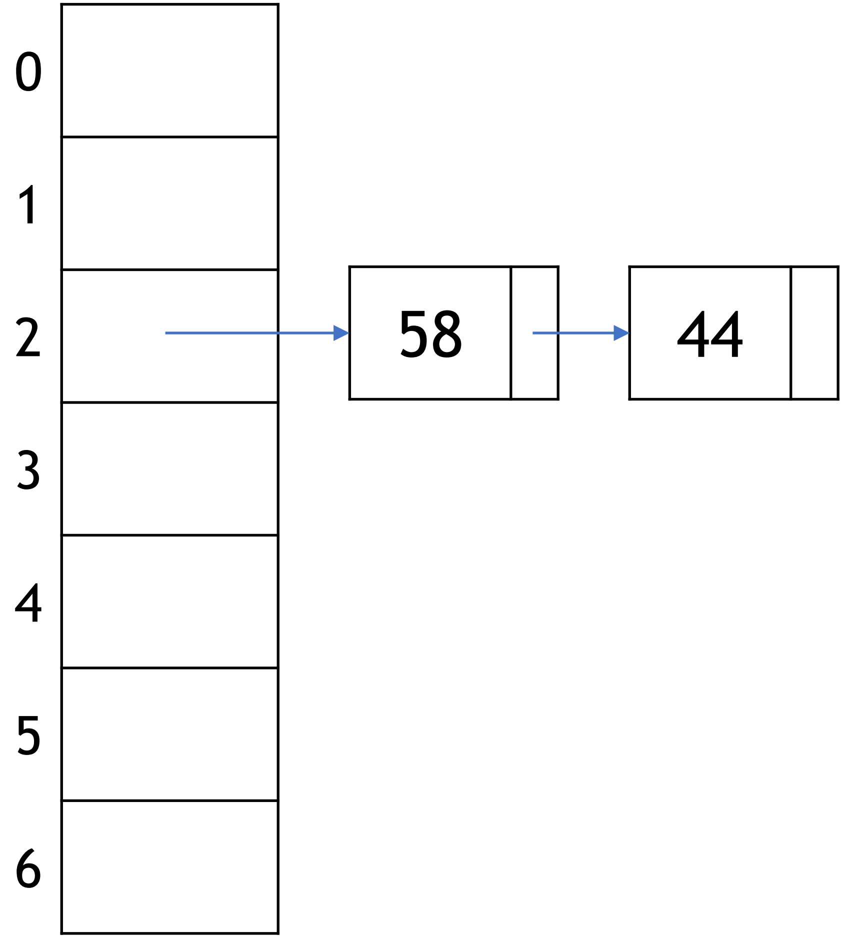
Collision Resolution: Direct Chaining

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2



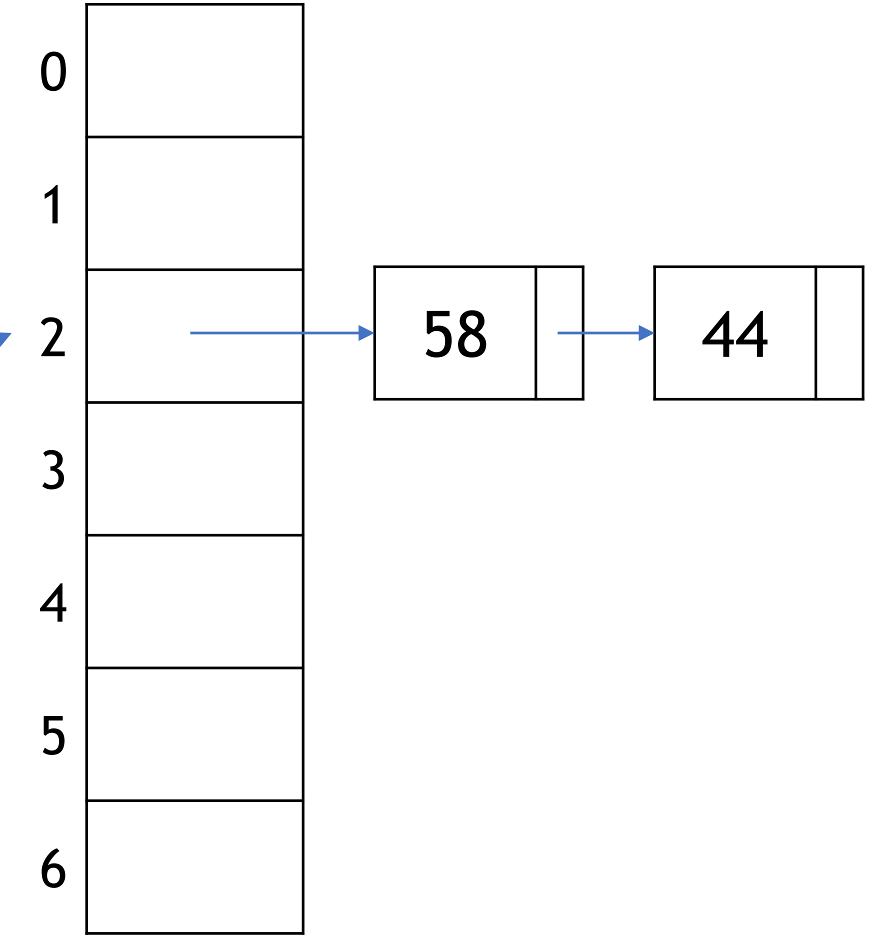
Collision Resolution: Direct Chaining

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2



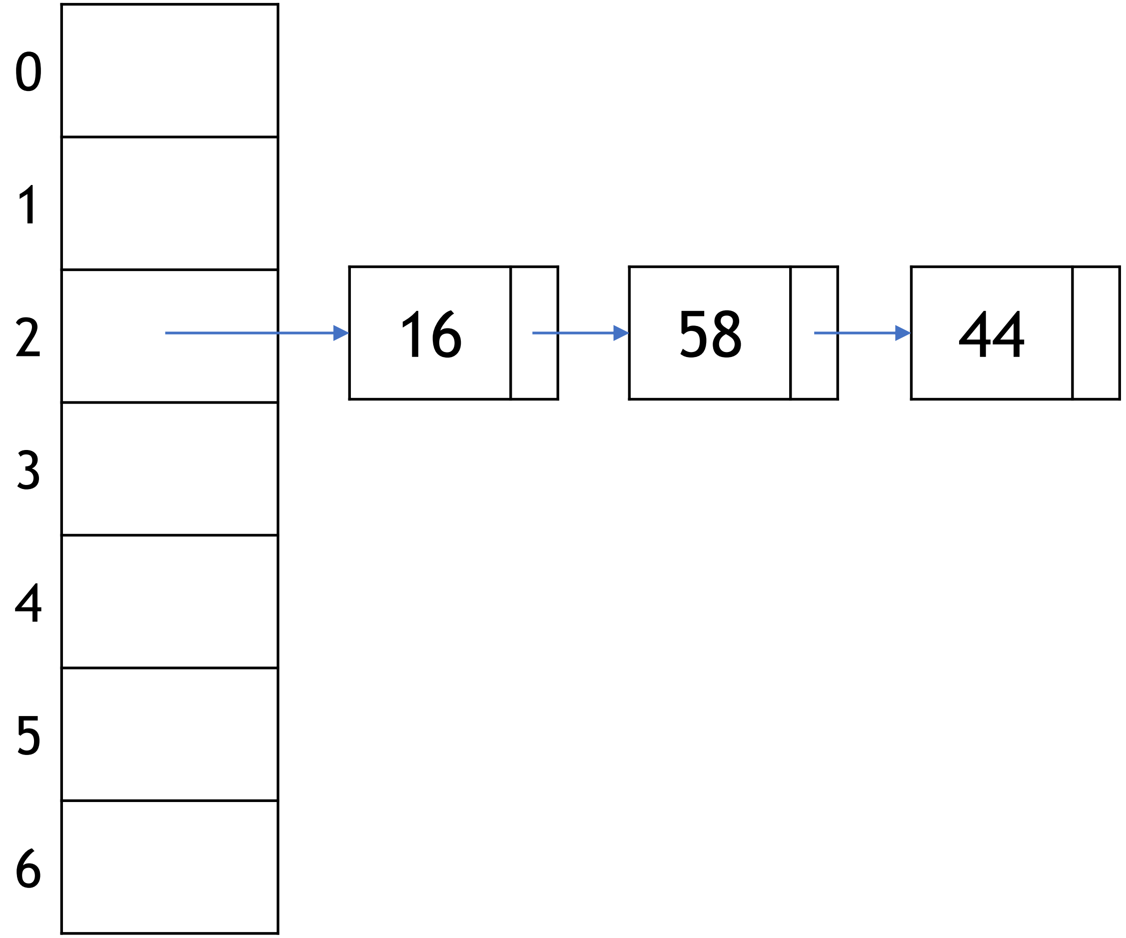
Collision Resolution: Direct Chaining

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2



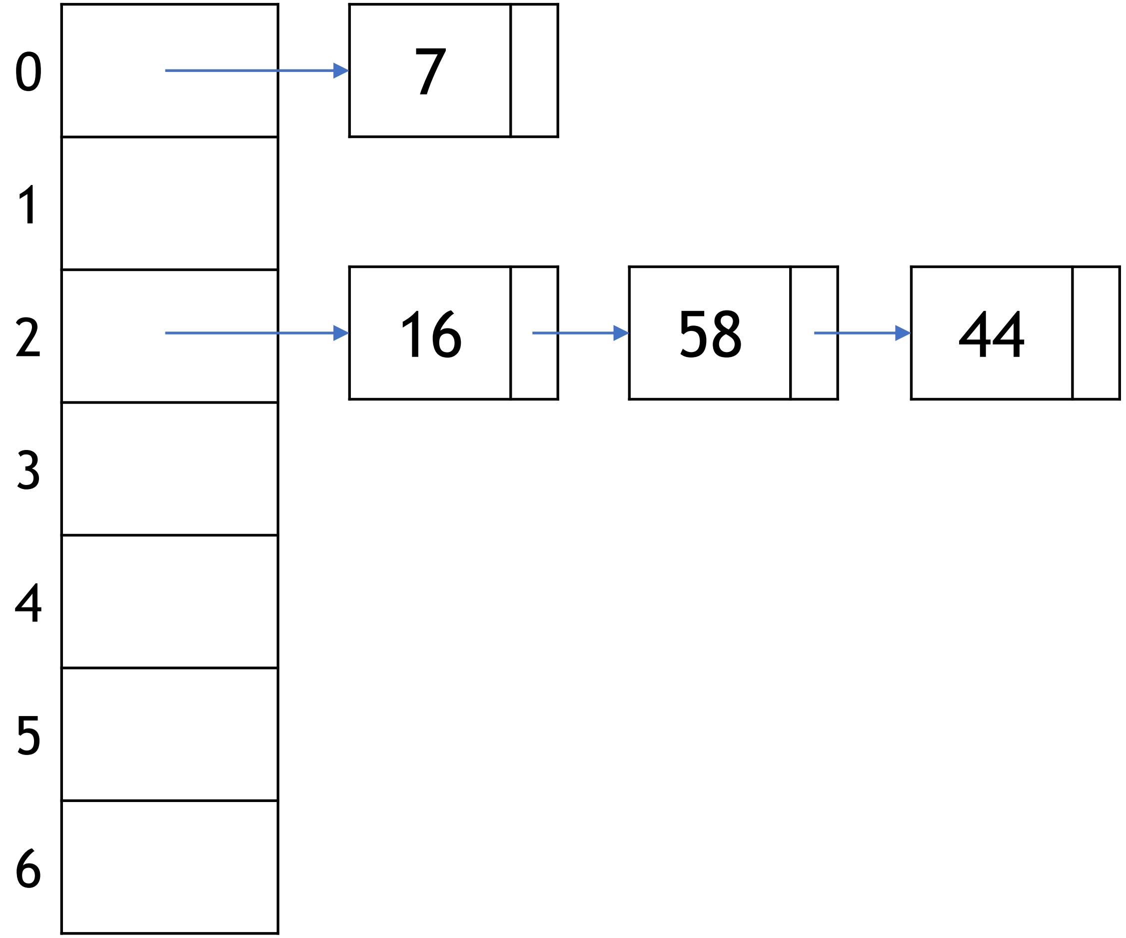
Collision Resolution: Direct Chaining

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2



Collision Resolution: Direct Chaining

Key	Key % 7	Hash
44	44 % 7	2
58	58 % 7	2
16	16 % 7	2
7	7 % 7	0



Collision Resolution: Direct Chaining

Insertion

```
insert(x, T=table) {
  temp = createNode(x);
  temp->next = T[h(x)]->head;
  T[h(x)]->head = temp;

return true;
}
```

Search

```
search(x, T=table) {
  temp = T[h(x)]->head;
  while(temp != NULL) {
    if(temp->data == x) {
      return temp;
    }
  }
return temp;
}
```

Collision Resolution: Direct Chaining

Chaining is Expensive

- Storing Linked List links requires extra memory
- Cache locality is not available

If Linked List becomes very large

- Operation's time complexity is worsened

Certain portions of the table may never be used

Open Addressing vs. Separate Chaining

	Open Addressing	Separate Chaining
Hash Function	Requires extra care to avoid clustering	Less sensitive to hash function or load factor
Usability Scenario	If frequency of operations and number of keys are known	If number of keys, operation frequency are unknown
Cache Performance	Great. Everything is maintained in the table (contiguous)	Poor. Linked List allocations are spread all over memory
Memory Usage	Slots can be used even if hash function doesn't map to it	Certain portions of the table may go unused; Also requires extra space for LL links

References

https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/

https://www.geeksforgeeks.org/hashing-set-3-open-addressing/