CSE 220

Data Structures

Lecture 07:
Hashing and Hash Tables

Anwarul Bashir Shuaib [AWBS]
Lecturer
Department of Computer Science and Engineering
BRAC University

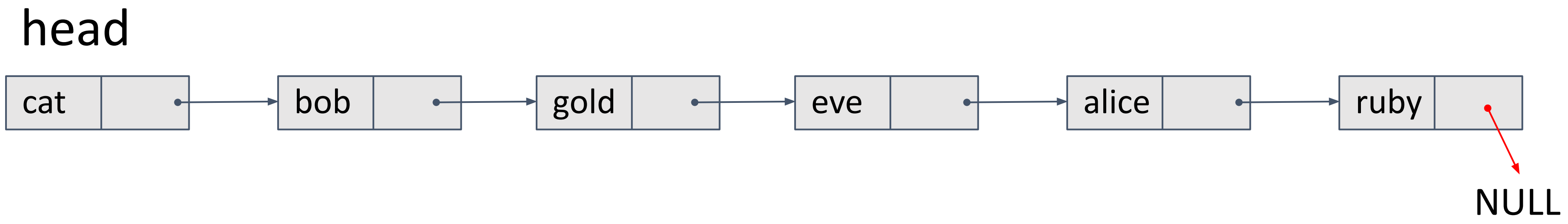
Need for Efficient Searching

Arrays: Fast access by index (`arr[0], arr[1], …`),
but slow to search by value (e.g., Is 'Alice' in this array?).

cat	**bob**	**alice**	**eve**	**gold**	**brown**
0	**1**	**2**	**3**	**4**	**5**

Need for Efficient Searching

- Same goes for linked lists.
- Find “alice” in this list ⇒ O(n)



Need for Efficient Storing

Add a new name “russel” at position 2 ⇒ O(n)

cat	bob	alice	eve	gold	brown
0	1	2	3	4	5

Same for linked lists ⇒ O(n)

Fast Searching and Storing

- What if the array is sorted?
  - For searching, best we can do is O(logn)
  - Insertion / deletion of an element is still O(n), as we need to shift elements to make room / fill up empty space

Problem with Arrays/Lists

- Arrays: Fast access by index, but slow to search by value (e.g., "Is 'Alice' in this array?").
- Lists: Easy to add/remove, but slow to search (check every item).

Solution: What if we could turn a **key** (like "Alice") into an **index** for instant access?

Hash tables are extremely efficient with both storing and retrieving
Both operations are O(1)!

Hash functions

Definition: A function that converts any key (text, number, etc.) into a number (hash code).
Example:

- Hash("Alice") → 140 → Index = 140 % 6 → **Index 2**
  Key Point: Same key → same index every time!

cat	bob	alice	eve	gold	brown
0	1	2	3	4	5

1. Key → Hash Function → Hash Code (e.g., 140).
2. Hash Code % Table Size → Index (e.g., 140 % 6 = 2).

More on hash functions

- Hash functions should be as different as possible for different keys
  - We call this property collision resistance
  - We try to come up with such hash functions so that it is extremely difficult to find two different inputs, say x and y, that produces same hash code, e.g: hash(x) == hash(y)
- Hash functions is a one-way function (you cannot decode the key from the hash code)
- For a given key (say, “alice”), hash functions will always produce the same hash code (for example, hash(“alice”) = 140)

Key-Value Mapping

- You have a list of user names, and their phone numbers.
- How would you store this information?

Username ⇒ Key
Phone number ⇒ Value

5513	2231	5123	6712	9224	5215
cat	bob	alice	eve	gold	brown

Key-Value Mapping

- You have a list of user names, and their phone numbers.
- How would you store this information?

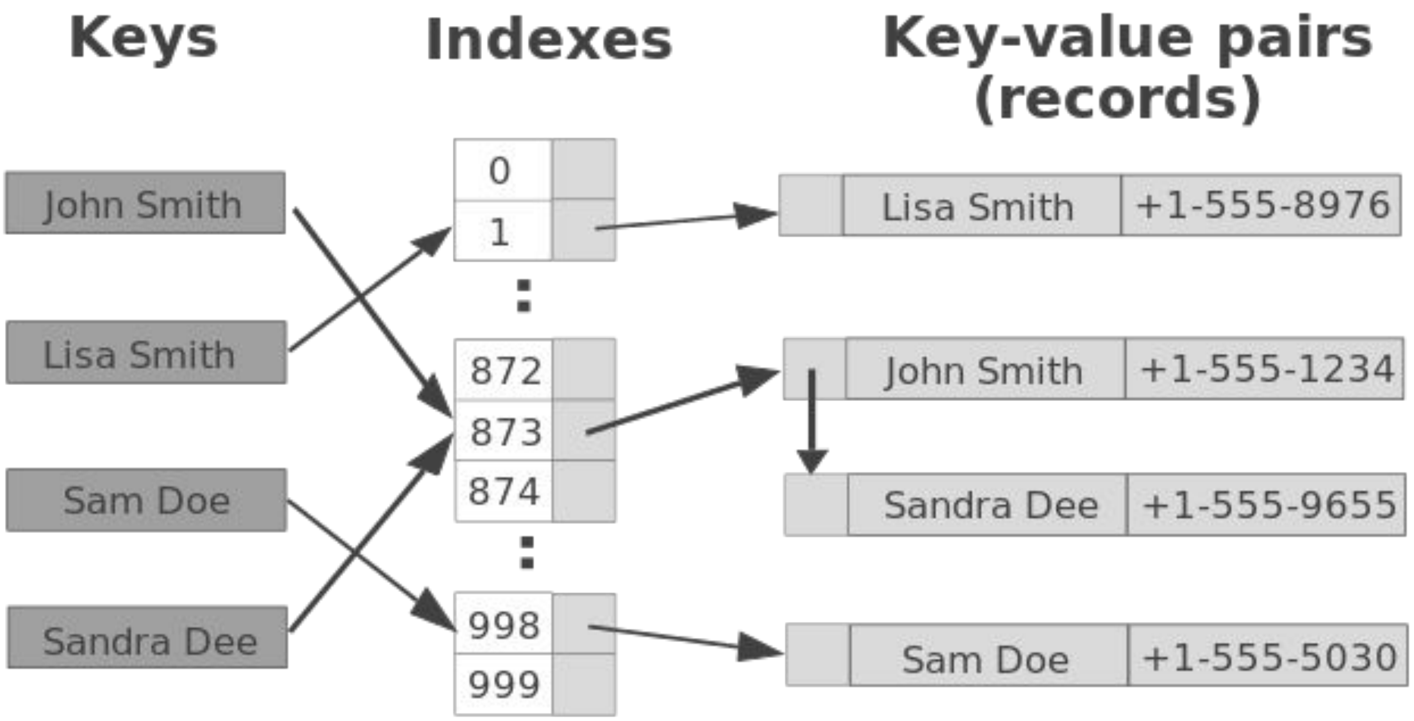
Username ⇒ Key
Phone number ⇒ Value

Values	5513	2231	5123	6712	9224	5215
Keys	cat	bob	alice	eve	gold	brown

Hash(key) % Table size = Index

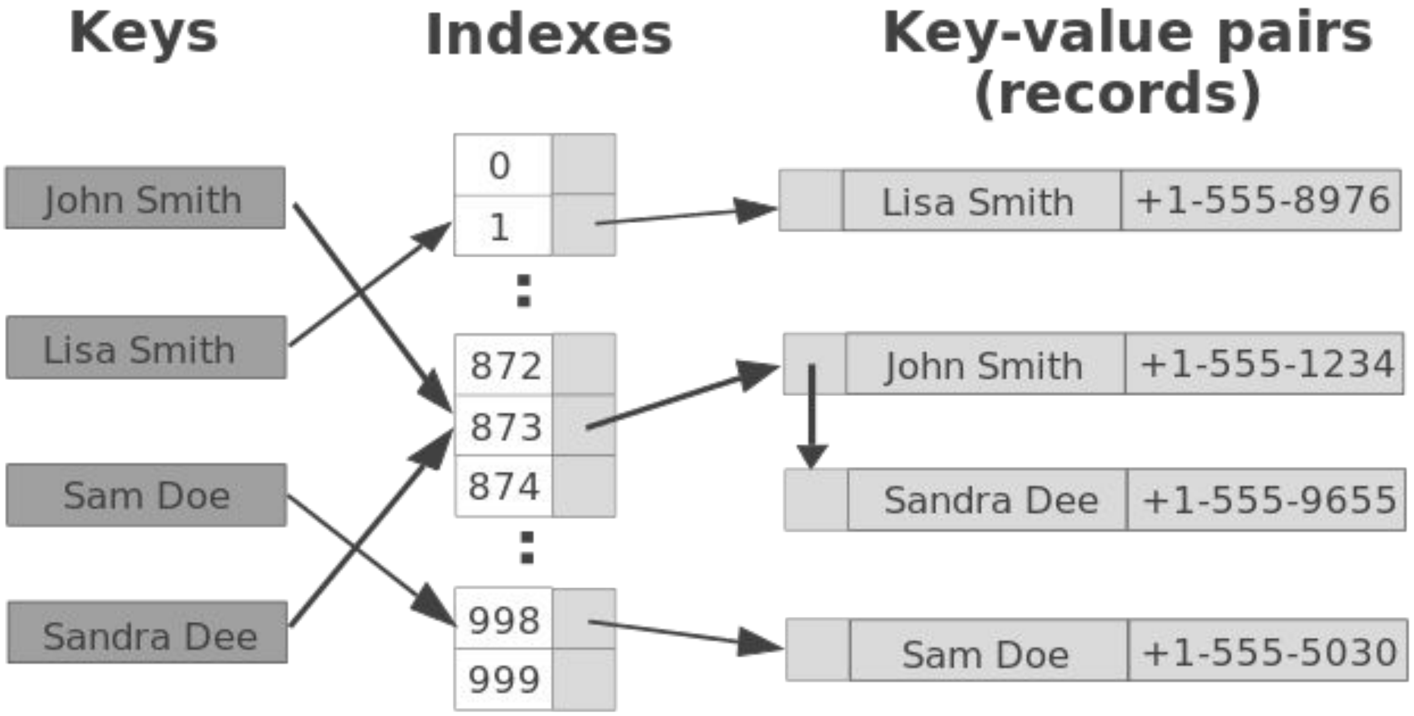
Hash Table

- We have infinitely possible keys, but limited memory
- We create a fixed sized array (say 1000), and map all possible (key, value) pairs in that array



Hash Table

- We store each (k,v) pair in a linked list. Each cell in the array is a linked list head.
- Whenever multiple keys map to the same position (collision occurs), we append the (k,v) pair in the linked list at that position.



Practice

- You have the keys 41, 18, 28, 36, 10, 90, 12, 54, 38, 25
- Hash table with size 13
- h(k) = k % 13
- Draw the final hash table after inserting the keys

Practice

- Step 1: Find indices for each keys

You can perform modulo on your calculator

k	h(k) = k % 13
41	2
18	5
28	2
36	10
10	10
90	12
12	12
54	2
38	12
25	12

- Step 2: Draw the table

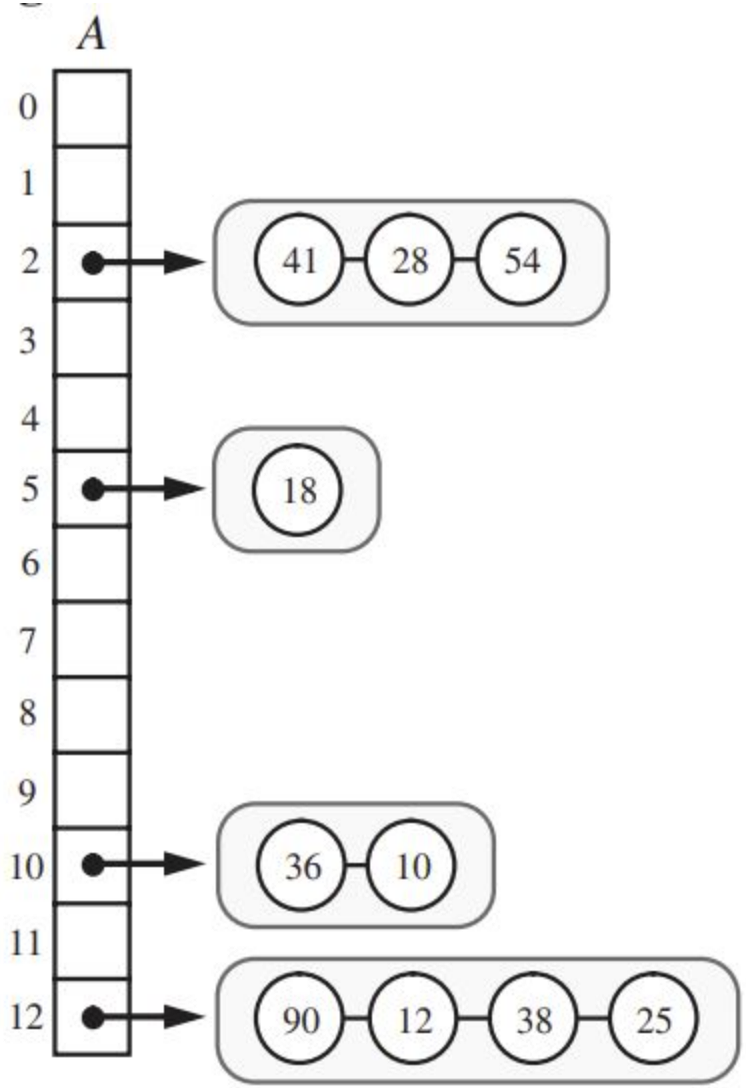


Figure 6.4: Example of a hash table of size 13, storing 10 integer keys, with collisions resolved by the chaining method. The hash function in this case is $h(k) = k \bmod 13$.

Operations on Hash Tables

- Denote the hash table by "M"

get(k): If M contains an item with key equal to k, then return the value of such an item; else return a special element NULL.

put(k, v): Insert an item with key k and value v; if there is already an item with key k, then replace its value with v.

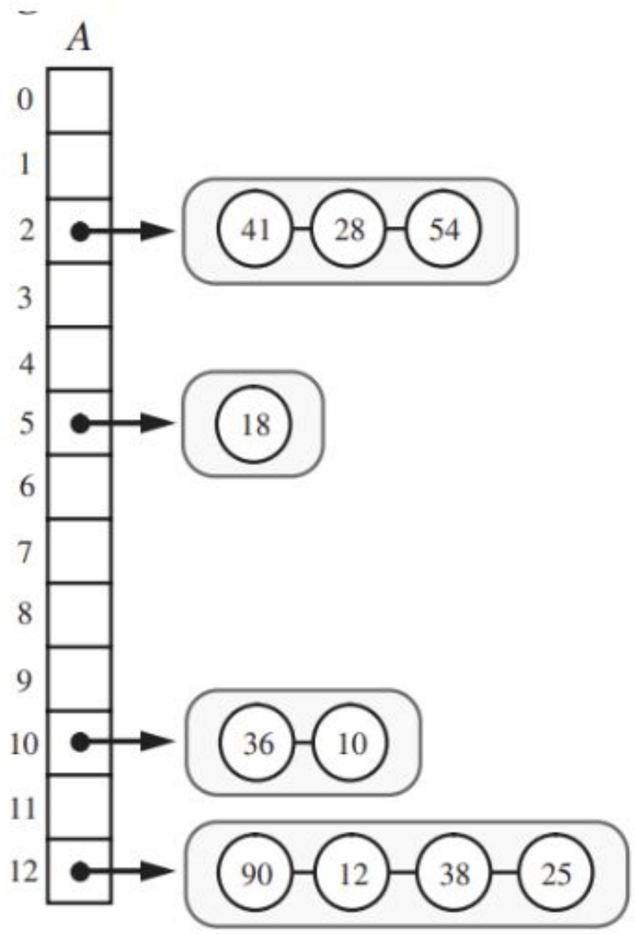
remove(k): Remove from M an item with key equal to k, and return this item. If M has no such item, then return the special element NULL.

Retrieving an item

- get(k):

  $B \leftarrow A[h(k)]$
  if $B = \text{NULL}$ **then**
      return NULL
  else
      return $B.\text{get}(k)$ // do a lookup in the list B

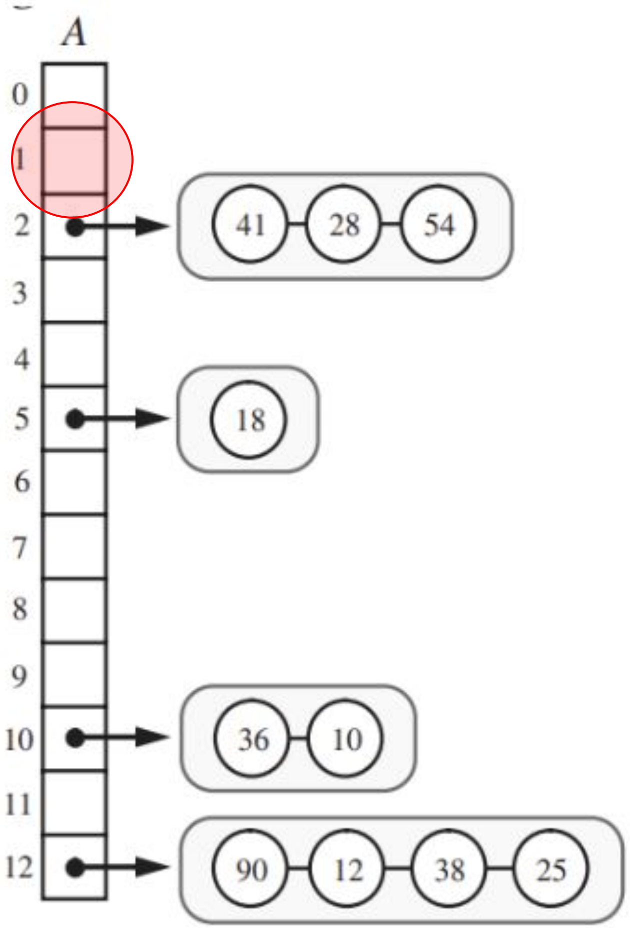
Retrieving an item



`hashtable.get(14)`

1. Compute h(14) = 14 % 13 = 1

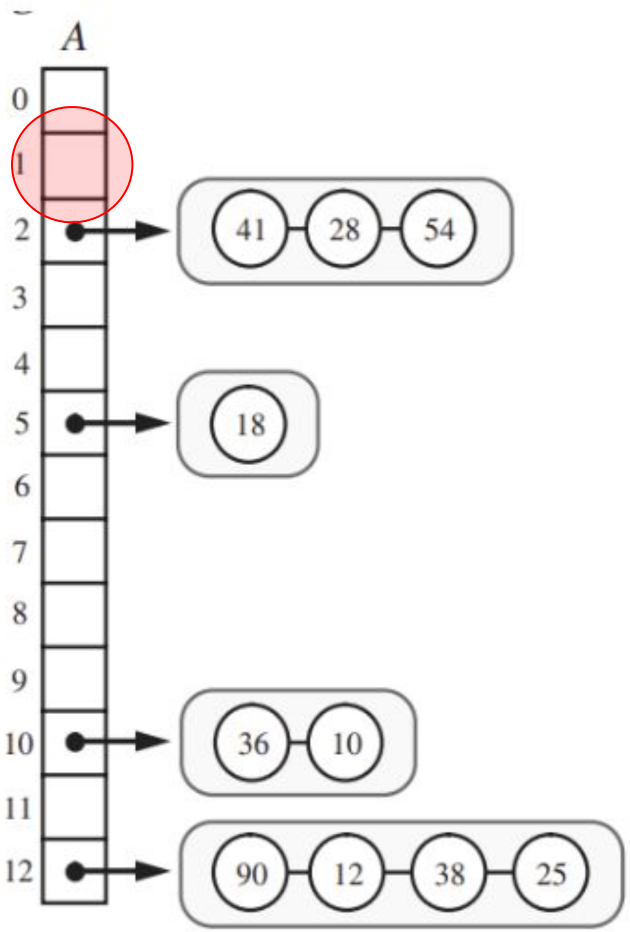
Retrieving an item




```
hashtable.get(14)
```

1. Compute h(14) = 14 % 13 = 1
2. Access A[1]

Retrieving an item

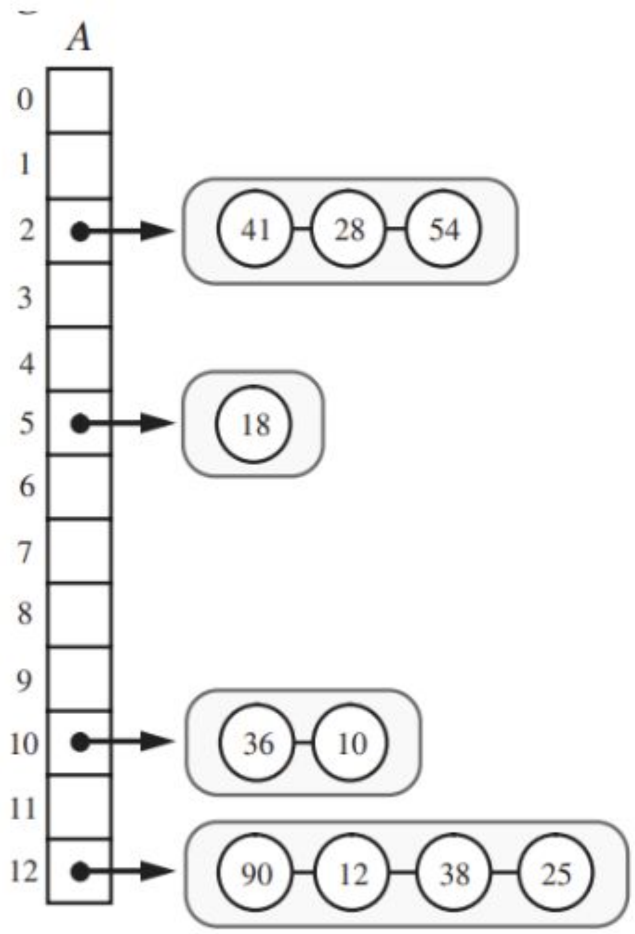


`hashtable.get(14)`

1. Compute h(14) = 14 % 13 = 1
2. Access A[1]
3. Since A[1] is empty (null), 14 does not exist.

Anwarul Bashir Shuaib [AWBS]

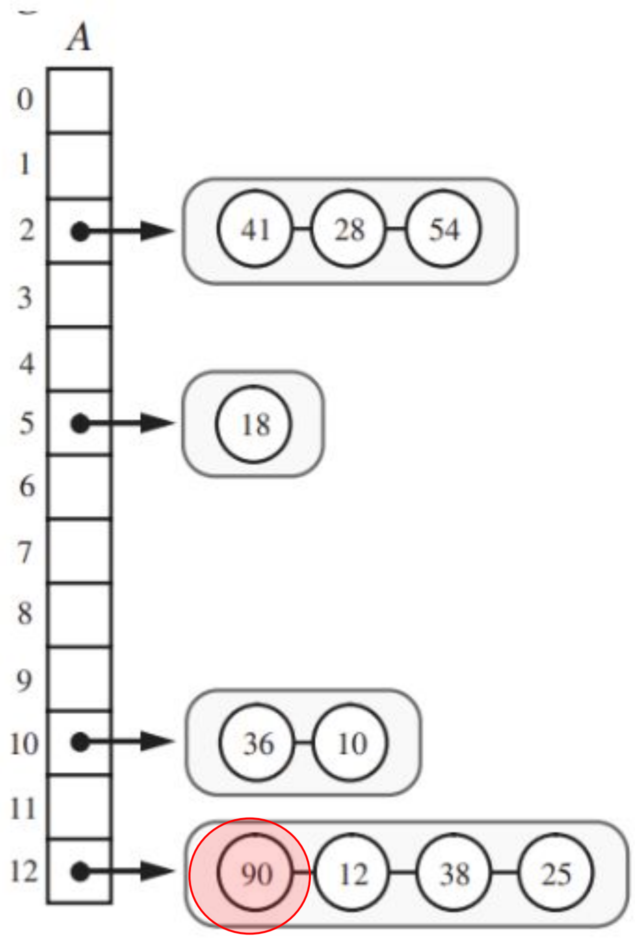
Retrieving an item



`hashtable.get(38)`

1. Compute h(38) = 38 % 13 = 12

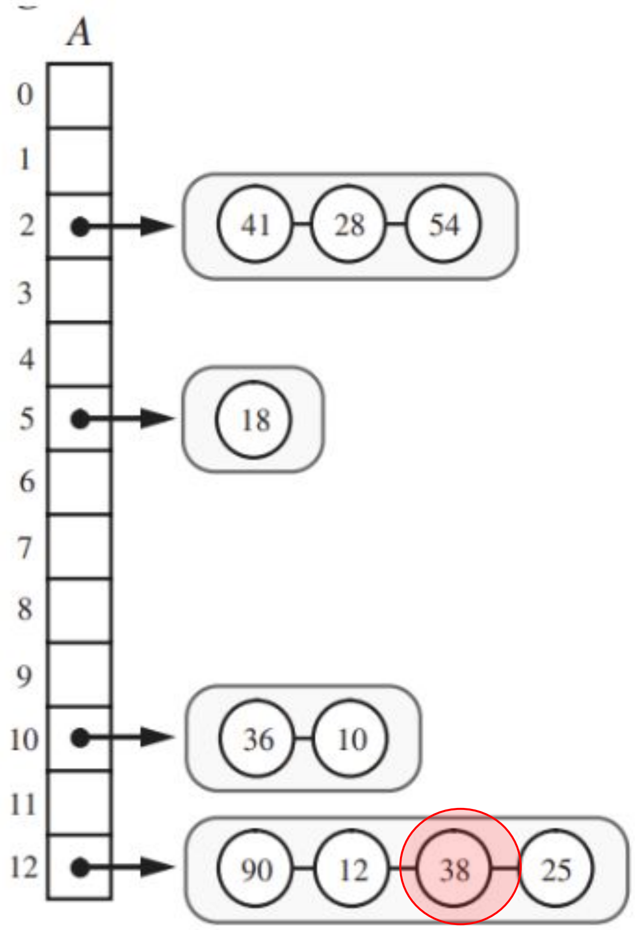
Retrieving an item



`hashtable.get(38)`

1. Compute h(38) = 38 % 13 = 12
2. Access the head at A[12]

Retrieving an item




```
hashtable.get(38)
```

1. Compute h(38) = 38 % 13 = 12
2. Access the head at A[12]
3. Perform a linear search starting from the head until you find the element

Insertion

- $\text{put}(k, v)$:

  if $A[h(k)] = \text{NULL}$ **then**

  Create a new initially empty linked-list-based map, B

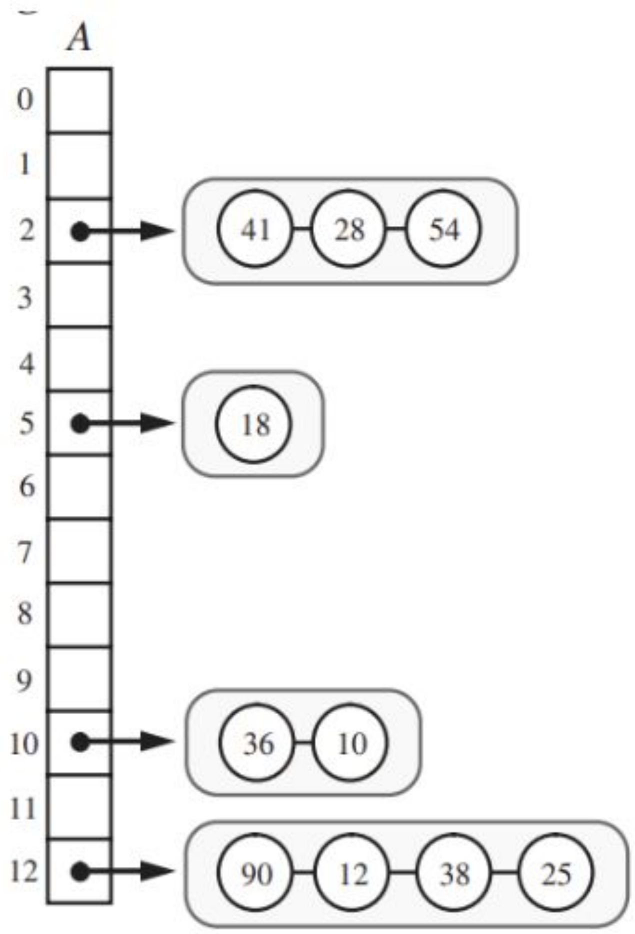
  $A[h(k)] \leftarrow B$

  else

  $B \leftarrow A[h(k)]$

  $B.\text{put}(k, v)$ // put (k, v) at the end of the list B

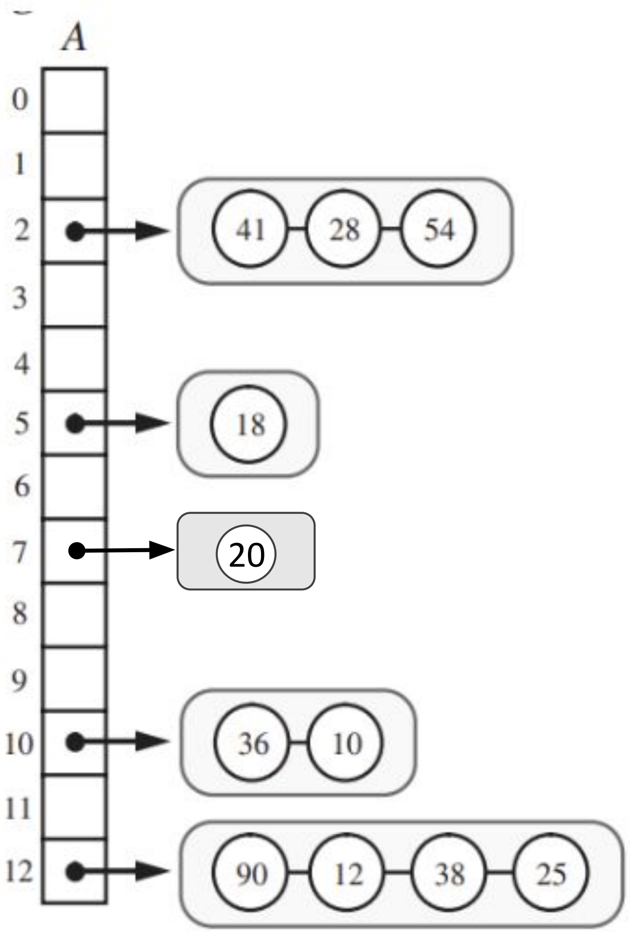
Insertion



`hashtable.put(20, 20) //` **key and value are the same**

1. Compute h(20) = 20 % 13 = 7

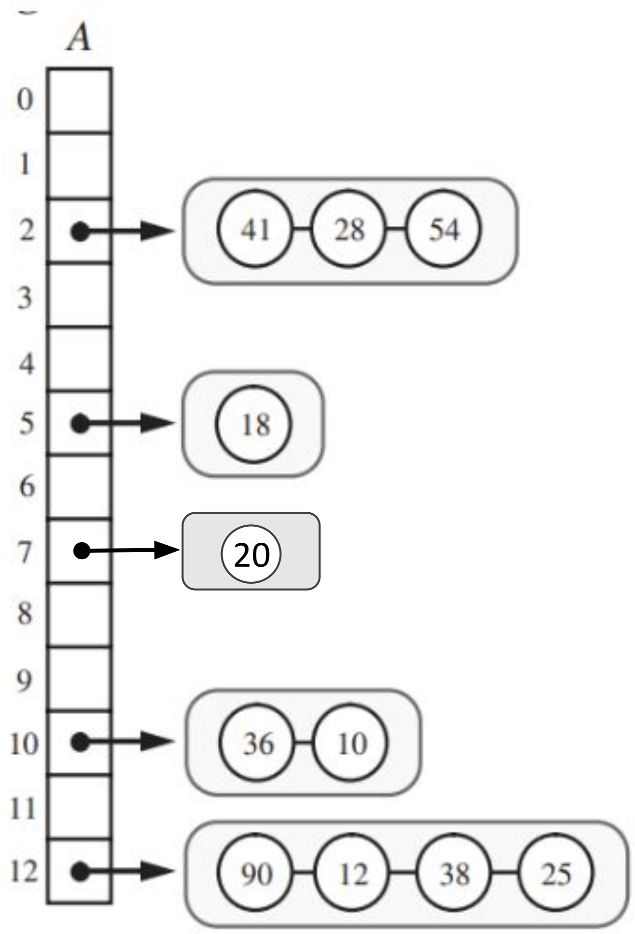
Insertion



`hashtable.put(20, 20) //` **key and value are the same**

1. Compute h(20) = 20 % 13 = 7
2. Insert element 20 at A[7]

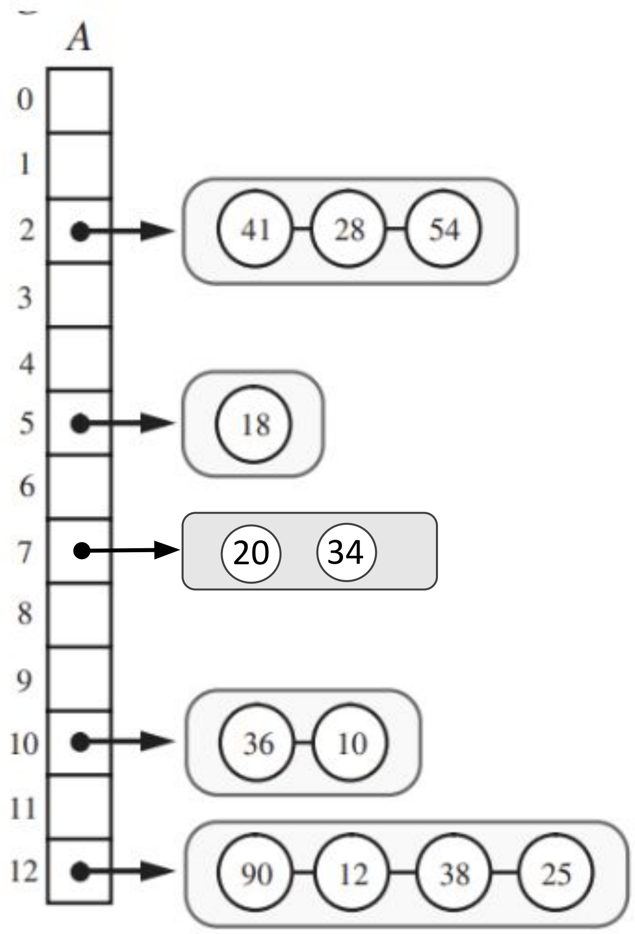
Insertion




```
hashtable.put(34, 34) //
```
key and value are the same

1. Compute h(34) = 34 % 13 = 7

Insertion



`hashtable.put(34, 34) //` **key and value are the same**

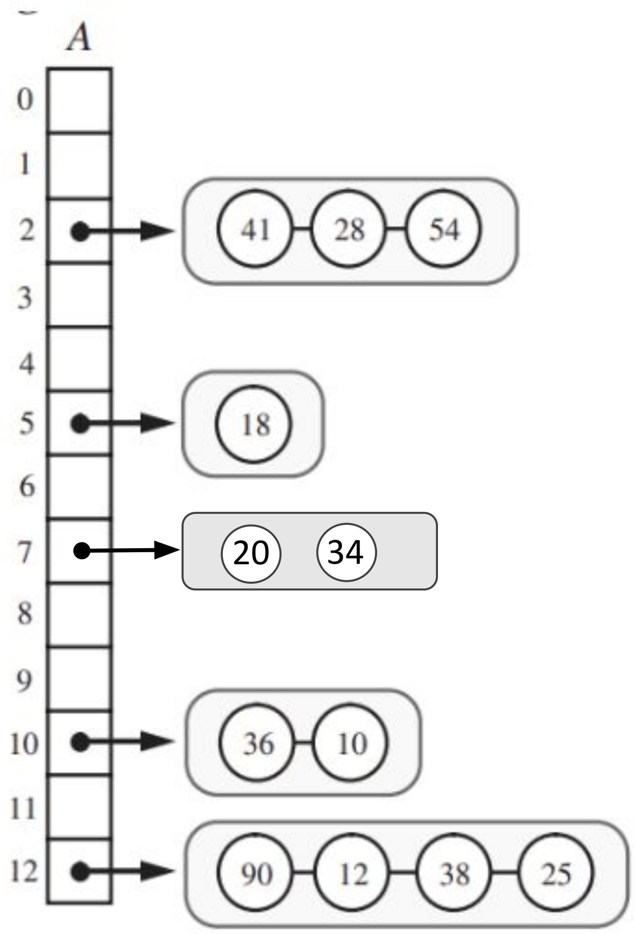
1. Compute h(34) = 34 % 13 = 7
2. Append 34 at A[7]

Removal

- remove(k):

  $B \leftarrow A[h(k)]$
  if $B =$ NULL **then**
      return NULL
  else
      return B.remove(k) // remove the item with key k from the list B

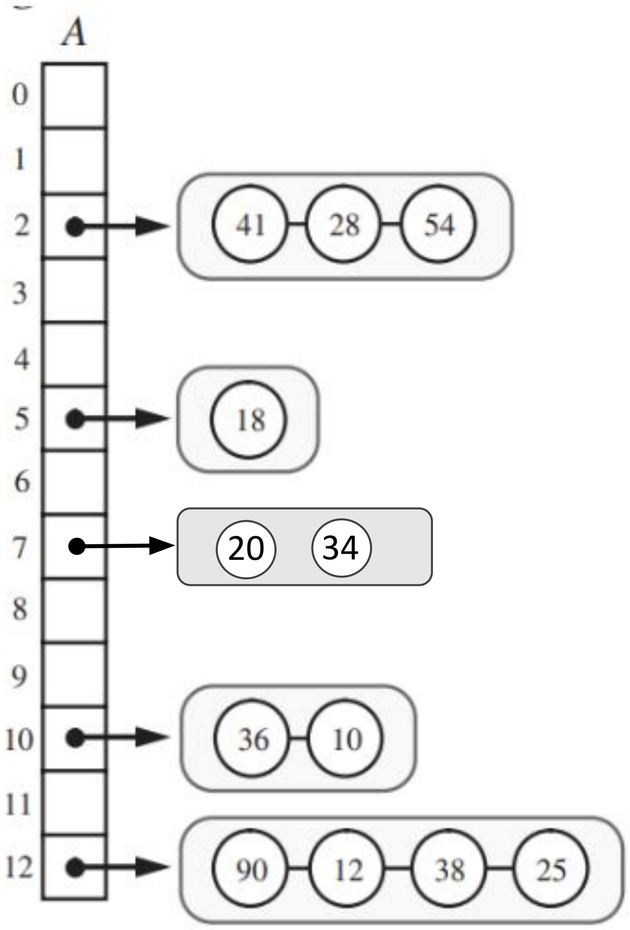
Insertion



`hashtable.remove(35)`

1. Compute h(35) = 35 % 13 = 8
2. Nothing to remove at A[8], so do nothing

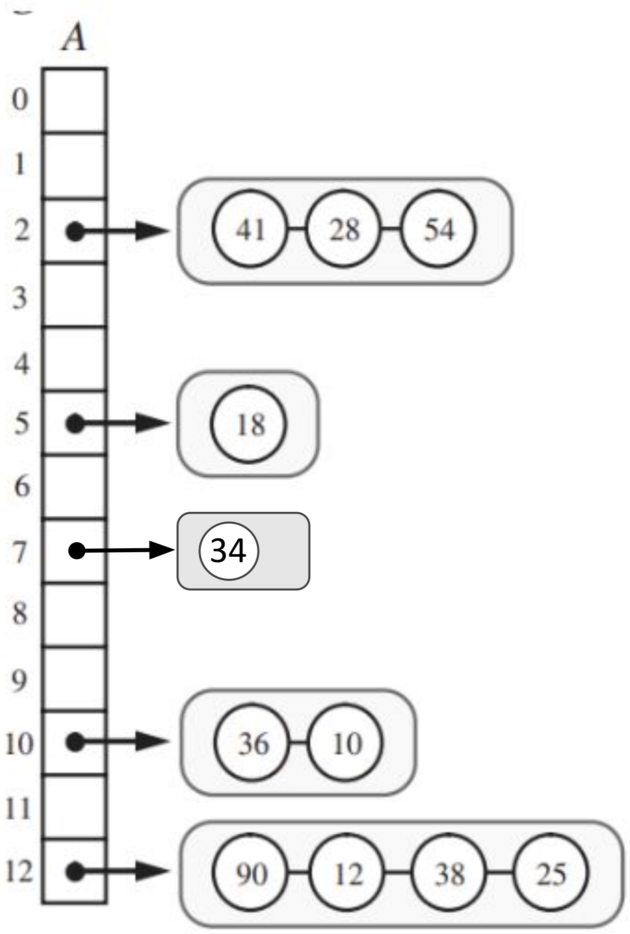
Insertion




```
hashtable.remove(20)
```

1. Compute h(20) = 20 % 13 = 7
2. Access the head at A[7]
3. Keep traversing the list at A[7] until you find 20, if found then remove it.

Insertion




```
hashtable.remove(20)
```

1. Compute h(20) = 20 % 13 = 7
2. Access the head at A[7]
3. Keep traversing the list at A[7] until you find 20, if found then remove it.

Why Constant Time?

Best Case:

- Perfect hash function → No collisions → **O(1)** time.

  Average Case:

- Good hash function + balanced table → Short linked lists → **≈O(1)**.

  Key Idea: Even with a few collisions, chains stay small if the table is large enough.

Summary

Hash Tables = Speed + Simplicity:

1. Hash function → instant index.
2. Forward chaining → handle collisions.
3. Result: Fast insert, search, delete!

   Real-World Use Cases: Databases, caches, spell-checkers.

Exercise

- Given an unsorted array of integers, find pairs (a,b) such that a+b = 0 in O(n) time.
  - Insert items into hashtable ⇒ O(n)
  - For each item x, find its equivalent negative pair (-x). If found, print (x, -x) ⇒ O(n)
  - Overall time: O(n)