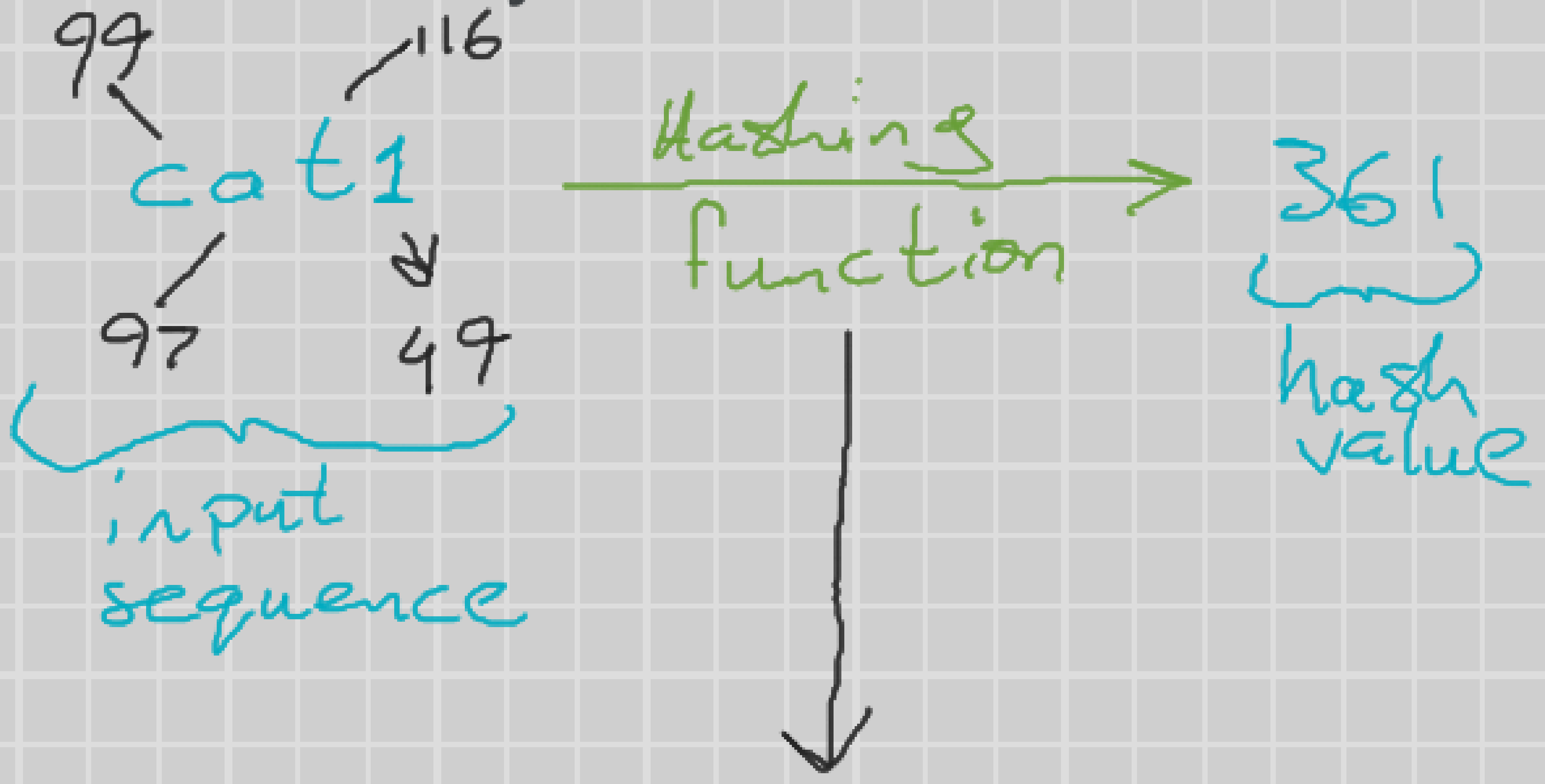
What is hashing?

-- Hashing is the process of transforming a sequence of alphanumeric characters to another value.

-- Hashing involves the use of a hashing algo - known as a Hashing Function.
-- A hashing function can be any function that's able to to transform an input sequence into a single value.



This hash function added together the ASCii values.

Applications of hashing

1) Security: ex. using SHA1 or other hash functions to hash and protect passwords. If you're transmitting a password from a client to a server you don't want plain text passwords being transferred.
-- Hash functions are computationally cheap to apply, but very difficult to reverse (computationally expensive), it's difficult to recover the original value from a has value.
-- Hash function are also called one-way functions because of this.

In our example hash function on slide 1, it's possible for different inputs to have the same hash value: a0K1, 2aK0, etc all have the hash value 270. There are other values that may add up to 27
they don't necessarily need to have the same input characters.

-- We want passwords to have unique hash values, so to do achieve that we can use more complex hash algos such as SHA (Secure hash algos).

2) Content verification (authenication): This is verifying the integrity of content.
-- If we send sensitive docs through a network how can we be sure that it hasn't been tampered with?
-- We can 't stop malicious tampering or avoid tranmission errors, but we can alert the receiver that the doc has been tampered with.

2) Content verification cont'd

To verify that the information wasn't tampered with or damaged in transition, sender applies cryptographic hash function and sends hash value, so receiver can check if information is the same on their end.

3) Fast Searching

If we hash numbers when we store them using a hashing function, we can use the same hash function to search for a value at an index without having to search the entire array. We would not need to search further if the value at the index providedisn't the one we're searching for. It can also be used for searching and deleting values (these are both constant time operations).

Example: We want to store the values: 23, 45, 120, 54 in a 4 element array. And we'll use the hash function:

-- $h(k) = k \% 4$. This is a reduction function that ensures our hash values always fall within the size of the array

Take h(k) of each value to identify where they should be placed in the hash table: [120, 45, 54, 23]

We've preprocesed the #s to be stored at specific positions in the array using the hash function, we can now search efficiently.

If we want to search our hash table to check if a number is present, we apply the hash function. Search for the number 9 in the hash table. h(9) = 9%4 = 1. So, we check the hash table at index 1 to see if the hash value matches the number 9. So here we have the basis for a search function that take constant time to run.

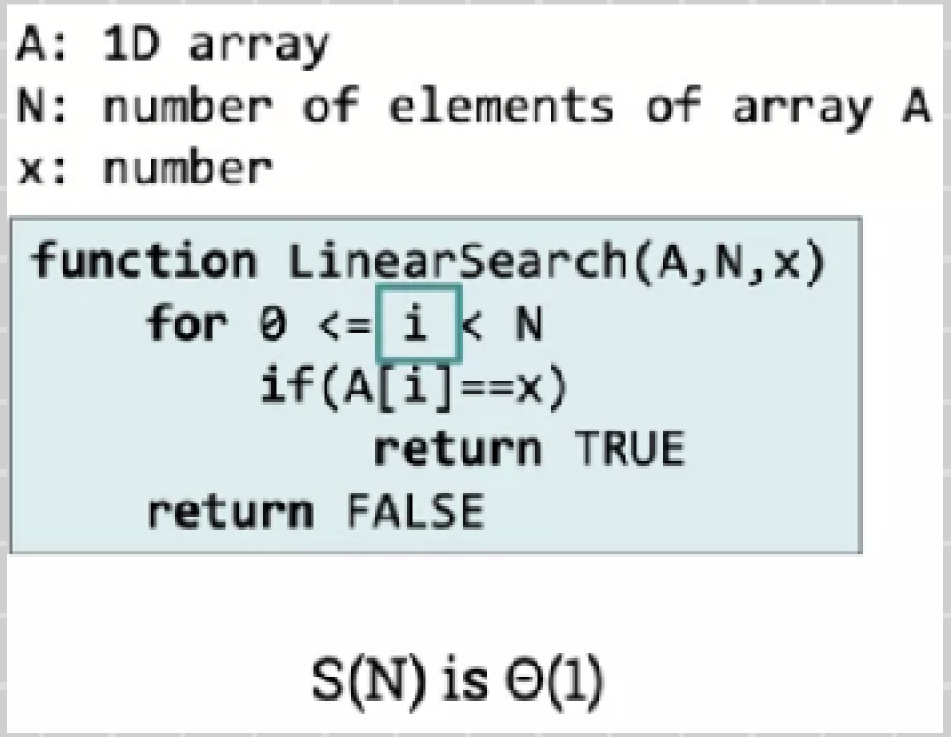
- When we added values to the array using the hash function, we created a Hash table.

Searching hash tables

Let's define the problem of searching:
-- the input into a search will be an array of numbers, and a number to search for.
-- the algo must return either TRUE or FALSE, if the value x is present in the input array.

Solution 1: Linear search

-- must check every position in an array in the worst case to determine if the value, x, is present or not. T(N) is θ(N).
-- in the best case the value, x, is the first element in the array. T(N) is θ(1).
-- In terms of memory, the space complexity is the same for best and worst case because only one simple variable "i" is created in a for loop.



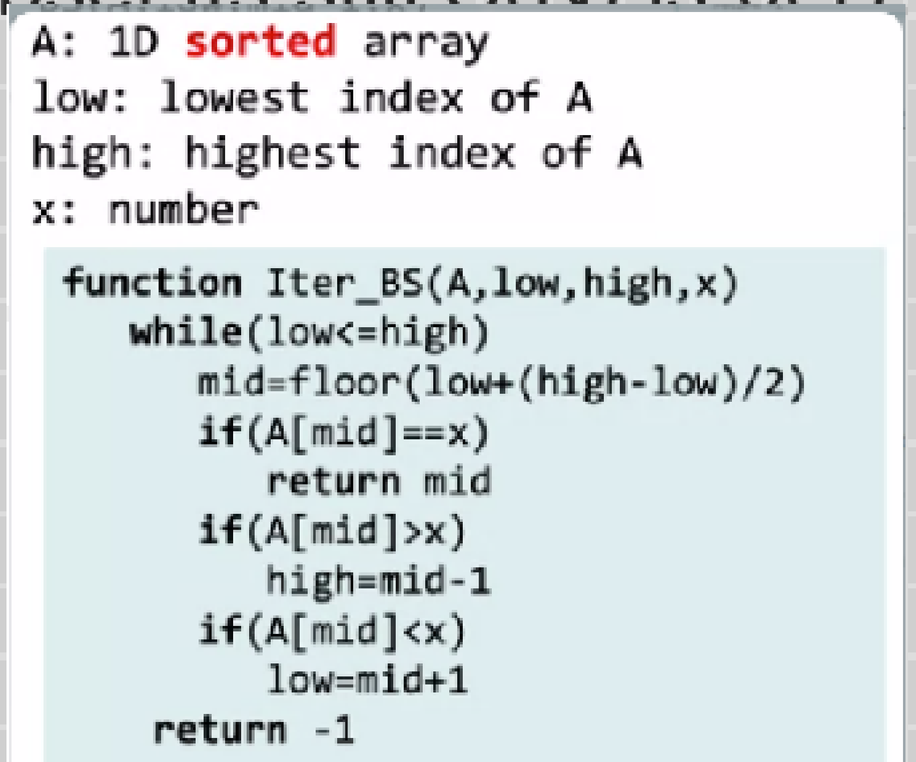
```
A: 1D array
N: number of elements of array A
x: number

function LinearSearch(A,N,x)
    for 0 <= i < N
        if(A[i]==x)
            return TRUE
    return FALSE
```

S(N) is Θ(1)

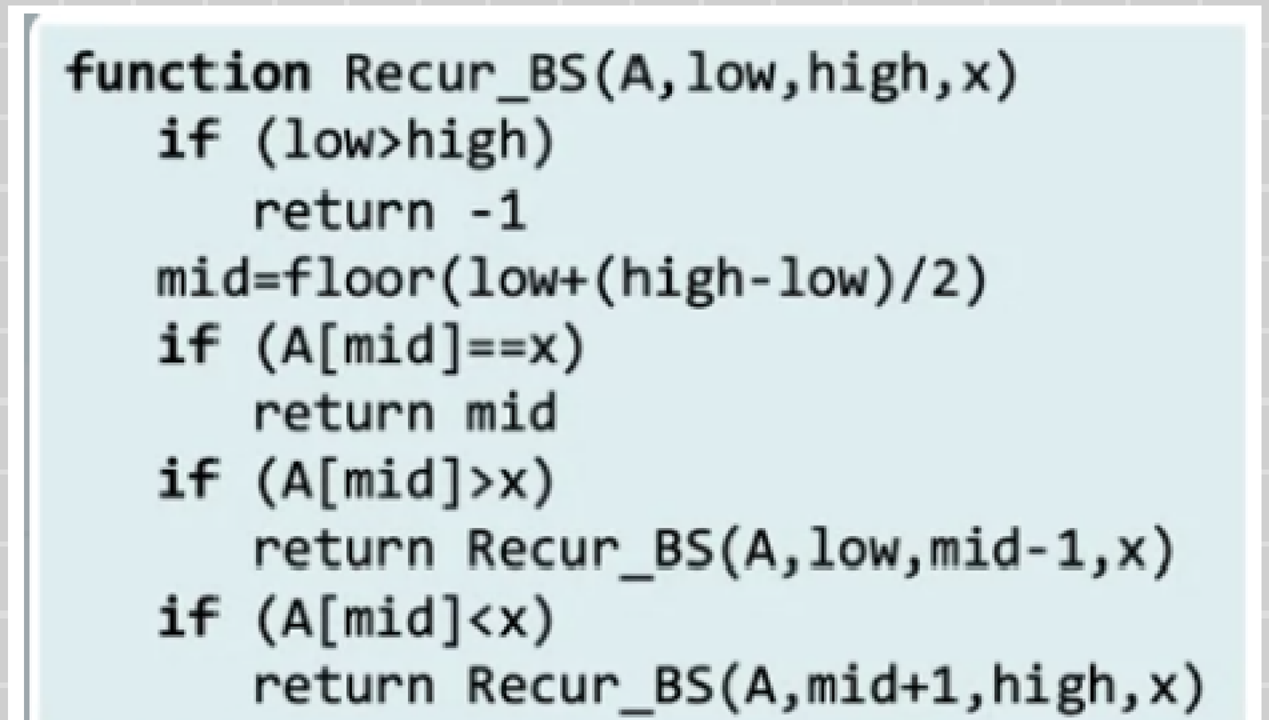
Solution 2: Binary search

-- requires that the input array is already sorted.
-- continuoulsy splits the array in half, discarding one half each time until the value, x, is found or the entire array has been searched.
--The time complexity of Binary search in both the iterative and recursive forms, in the worst case is T(N) is θ(log N).
-- In the best case, the value we are searching for is at the first midpoint and T(N) θ(1).

-- In terms of space complexity both versions differ from each other:
-- in the iterative version "mid" is a simple variable created once, so space complexity isS(N) is θ(1) in both the best and worst cases.
-- In the recursive version "mid" has to be created at every recursive call, and therefore S(N) is θ(log N) in the worst case. In the bset its still S(N) is θ(1).



```
A: 1D sorted array
low: lowest index of A
high: highest index of A
x: number

function Iter_BS(A,low,high,x)
   while(low<=high)
      mid=floor(low+(high-low)/2)
      if(A[mid]==x)
         return mid
      if(A[mid]>x)
         high=mid-1
      if(A[mid]<x)
         low=mid+1
   return -1
```

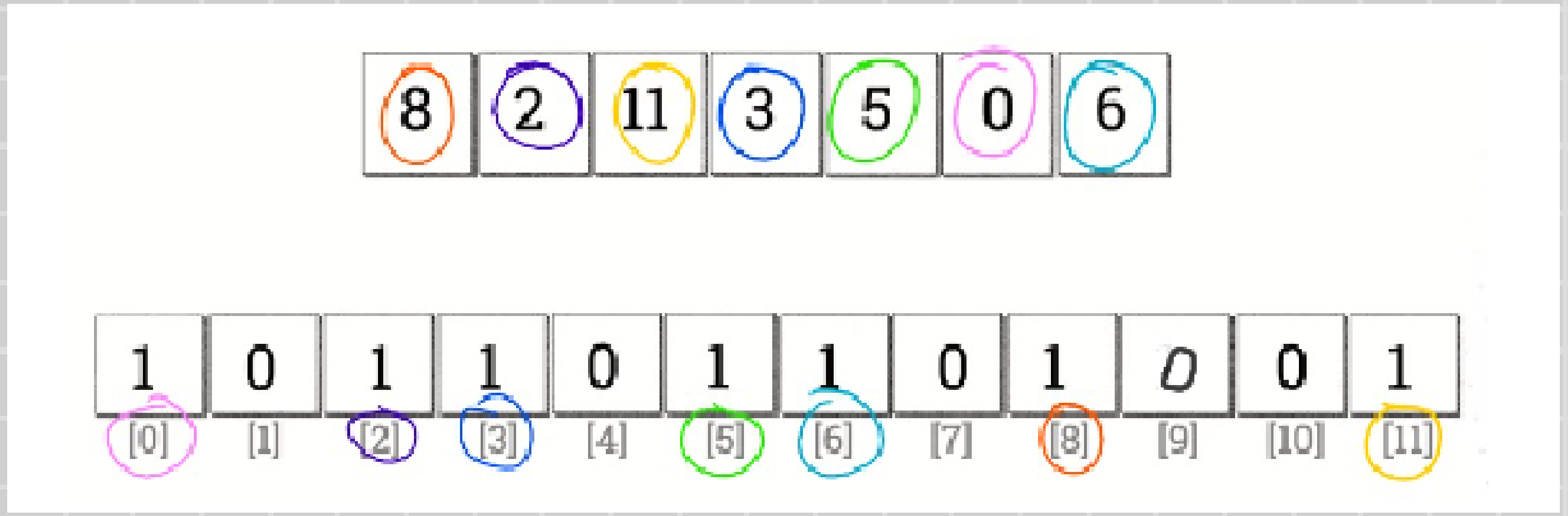


```
function Recur_BS(A,low,high,x)
   if (low>high)
      return -1
   mid=floor(low+(high-low)/2)
   if (A[mid]==x)
      return mid
   if (A[mid]>x)
      return Recur_BS(A,low,mid-1,x)
   if (A[mid]<x)
      return Recur_BS(A,mid+1,high,x)
```

Solution 3: Direct addressing

-- This uses the index of an array to represent a #, similar to counting sort.
A = [8, 2, 11, 3, 5, 0, 6]
-- We would create a new array called a Bit Vector (direct address table), where each index represents the number in the original array. It's called a bit vector because 0's are placed if an element is not in the original array, and a 1 if it is. If "k" is the largest value in the original array then the bit vector must be of size "k+1".

0 1 2 3 4 5 6 7 8 9 10 11
Bit vector = [1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1]



Searching for a number in a bit vector psuedocode:

```
B: bit vector
x: # being searched for
function DirAddSearch(B, x)
    return B[x]
```

This function has a time complexity of θ(1). The potential problem with this method is the size of the bit vector that needs to be created. If the max in the input array is very large, then we must create a bit vector of size max+1. Therefore space complexity is S(N) is θ(k), where k is max val in original array.

Solution 4: Hash tables

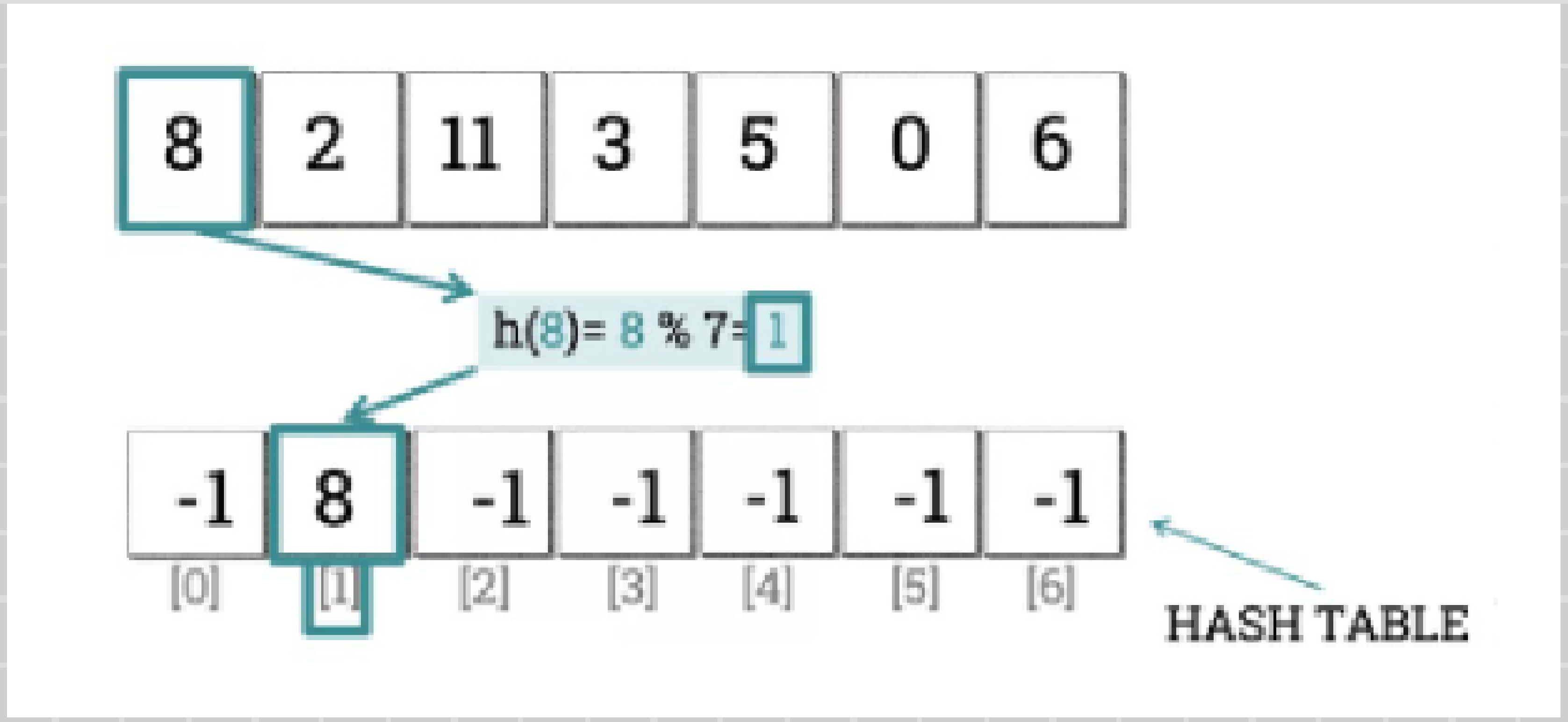
This method is similar to direct addressing in that it take θ(1) time complexity to search.

-- The hash table requires less elements than the bit vector.

-- We can create a hash table of the same size as the original array when we know how many items are in the array.

-- We initialize the hash table with values of -1.

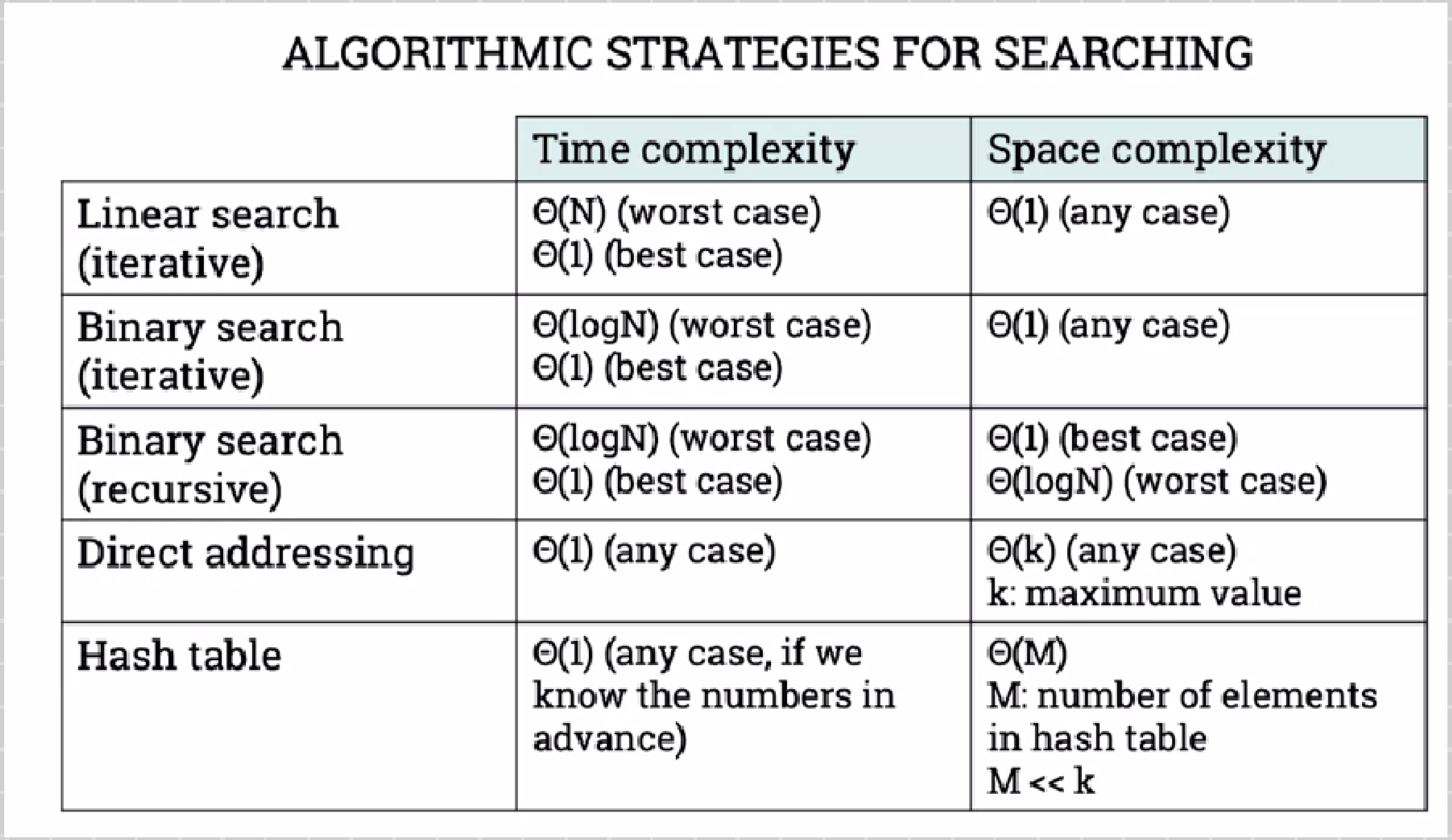
-- Using a hash function we transform the numbers from the original array into an index value, and place them in the hash table.



```
H: hash table
x:  number
function hashSearch(H, x)
    index = h(x) // hash function
    if (H[index] === x) then
        return TRUE
    return FALSE
end function
```

Solution 4: Hash tables cont'd

-- hash table searching has a time complexity of T(N) is θ(1) because we only need to check one value.
-- in terms of space complexity, S(N) is θ(M) =, where M is the size of the original array



ALGORITHMIC STRATEGIES FOR SEARCHING

	Time complexity	Space complexity

Linear search (iterative)	Θ(N) (worst case)
	Θ(1) (best case)
		Θ(1) (any case)

Binary search (iterative)	Θ(logN) (worst case)
	Θ(1) (best case)
		Θ(1) (any case)

Binary search (recursive)	Θ(logN) (worst case)
	Θ(1) (best case)
		Θ(1) (best case)
		Θ(logN) (worst case)

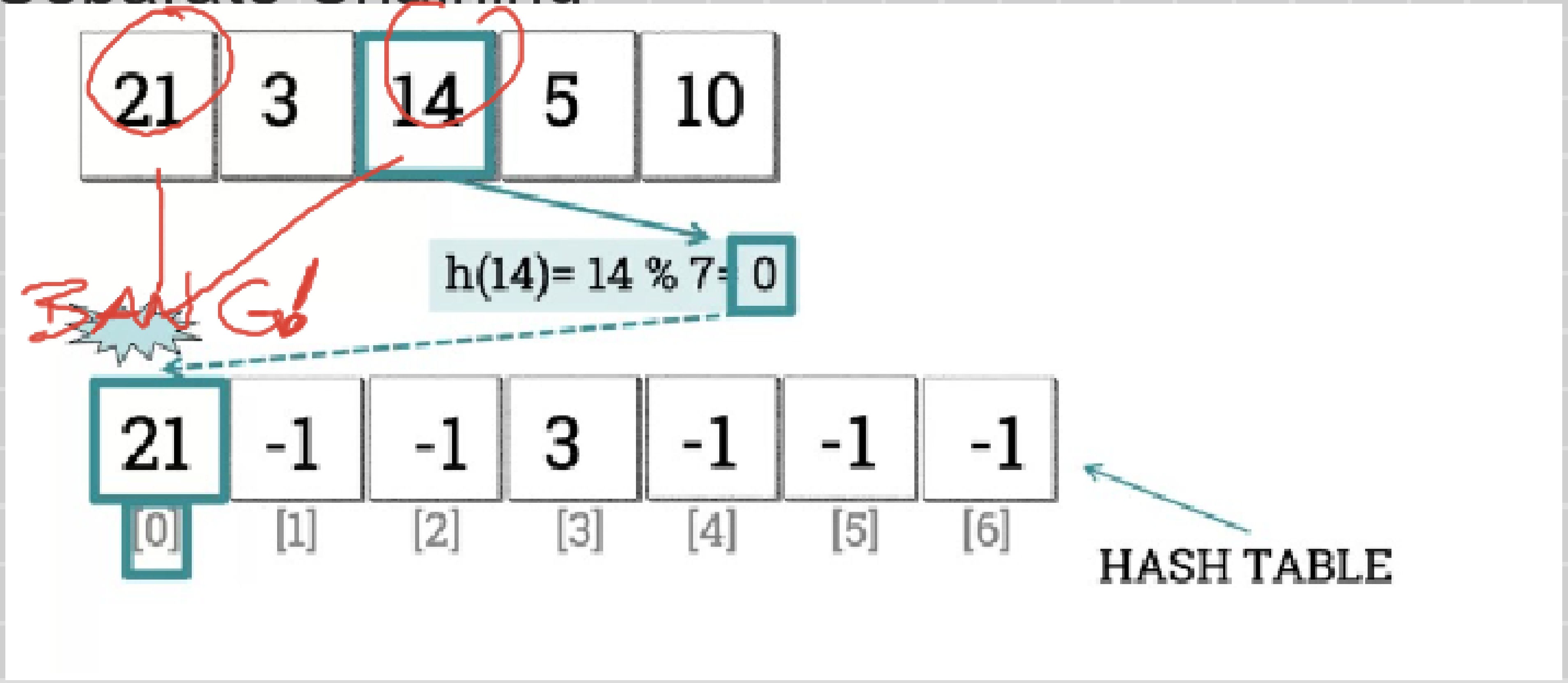
Direct addressing	Θ(1) (any case)	Θ(k) (any case) k: maximum value
Hash table	Θ(1) (any case, if we know the numbers in advance)	Θ(M) M: number of elements in hash table M << k

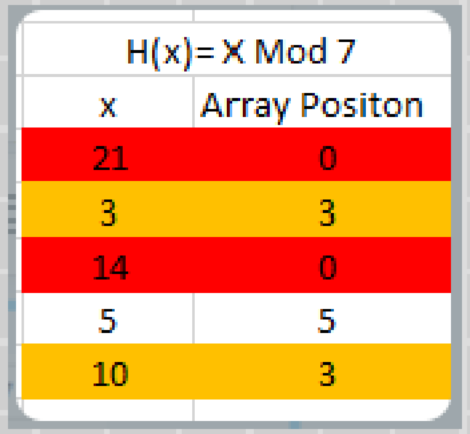
Collisions in hash tables

What happens when we don't know what numbers will be placed in a hash table or we don't know the size of the input array?

-- One problem that we will run into is that the hash function will map two more values to the same index.

-- To address this problem we have three solutions: 1) Extend & rehash 2) Linear probing 3) Separate Chaining





H(x)= X Mod 7	
x	Array Positon
21	0
3	3
14	0
5	5
10	3

Method 1: Extend & rehash

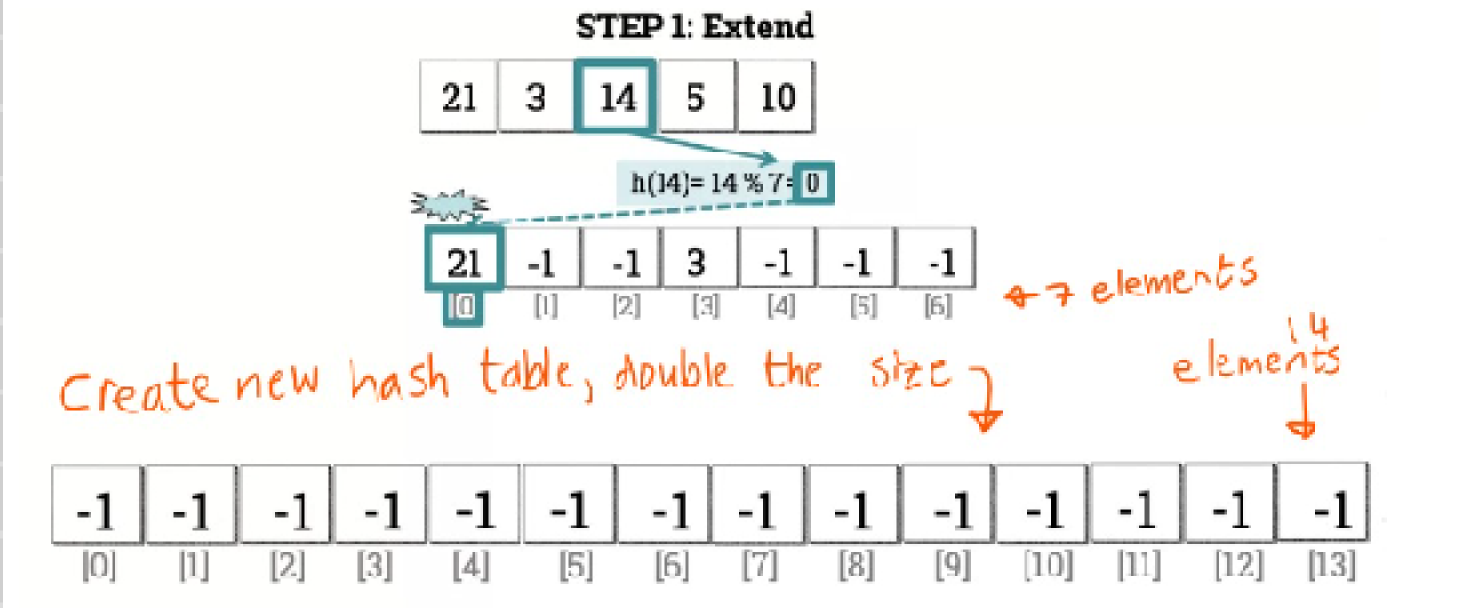
-- when two values from the original the orginal array are mapped to the same value, we can:

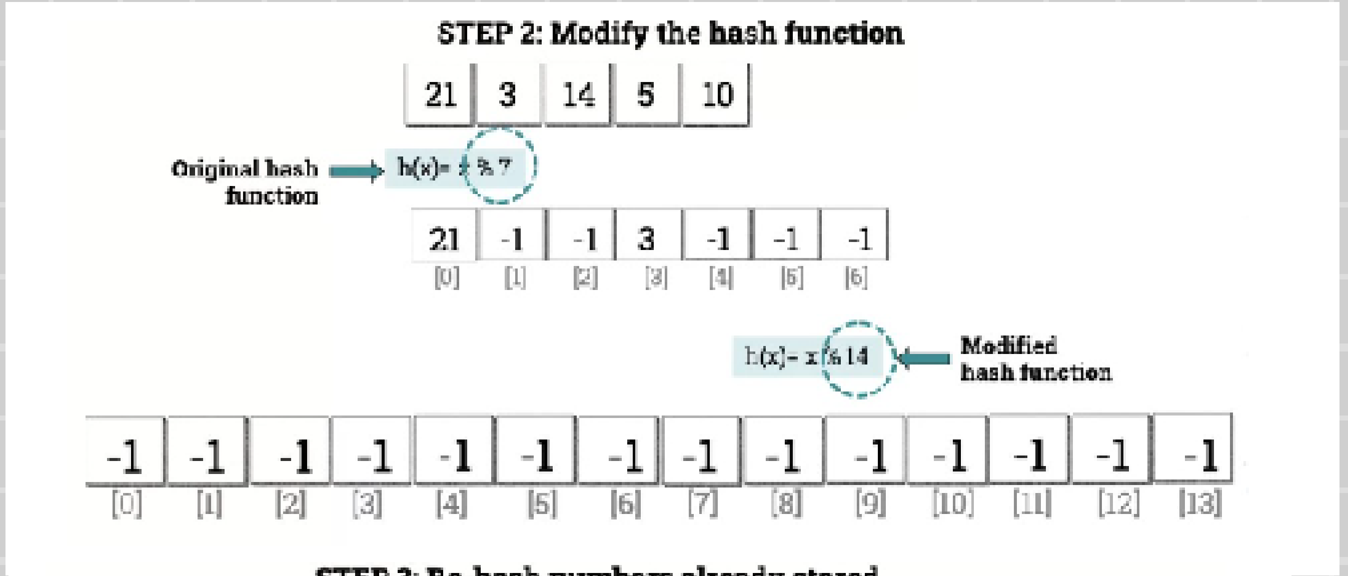
1) enlarge the hash table to fit more items, we can double it to start.
2) Modify our reduction function (hash function). We must accomadate the larger hash table size to reduce the input values to fit the new range.
3) Re-hash numbers that are already stored in the hash table. Since we have updated the reduction function all of the numbers must be run through the has function again to move them to their new postions in the hash table.

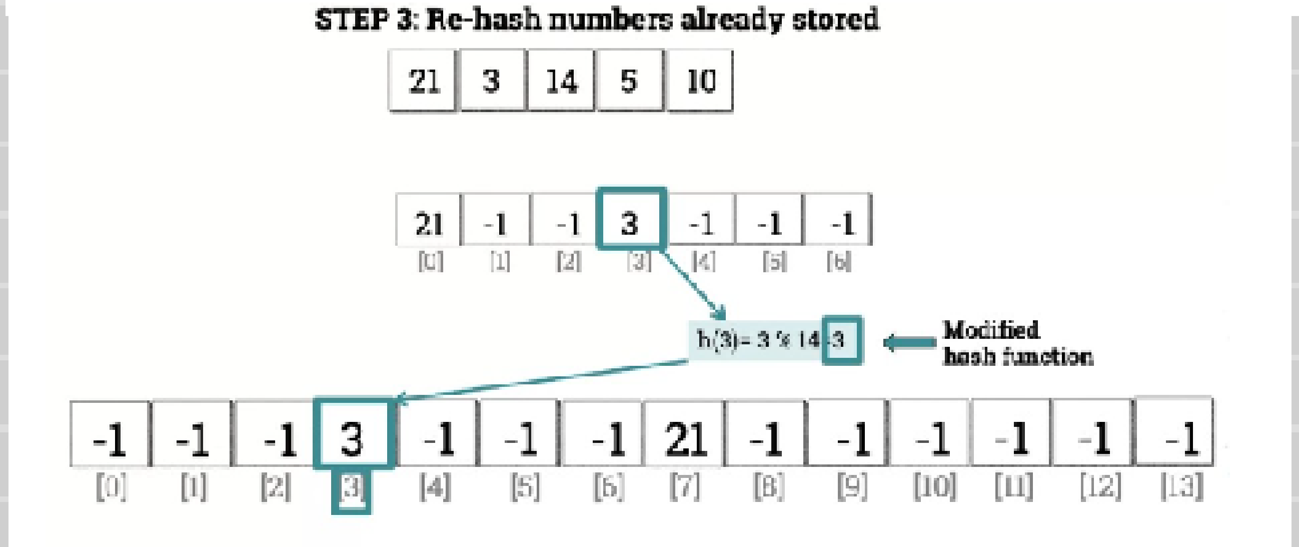
Extend and re-hash can be applied in two ways: reactive way, immediately after the collision occurs, and proactive way, every time that specific number of positions is filled. This number is known as load factor.

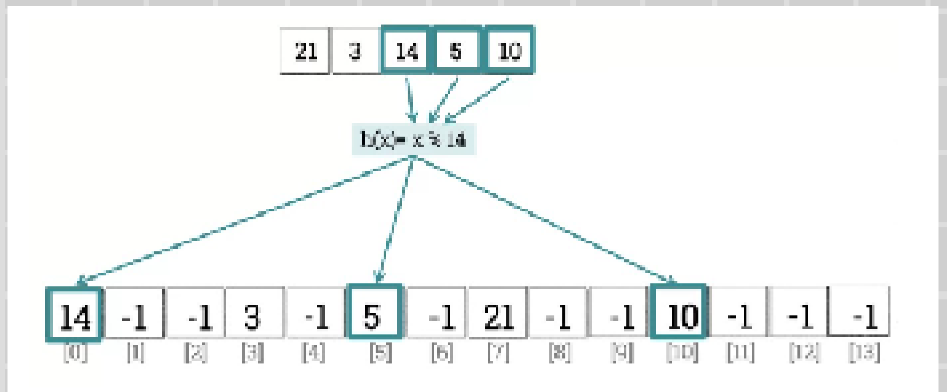
Load factor = buckets used/total # of buckets

--used every time the load factor exceeds a given threshold.



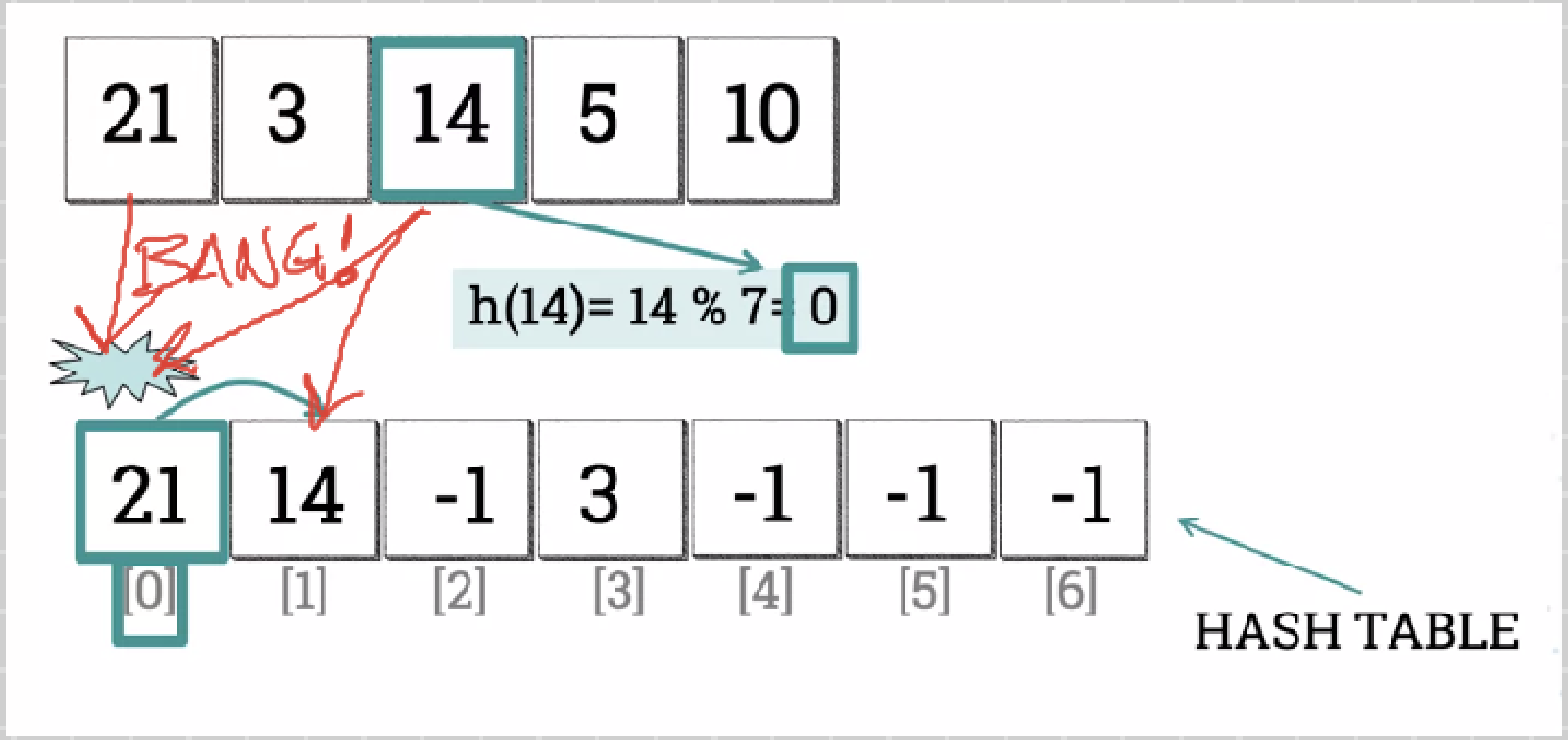




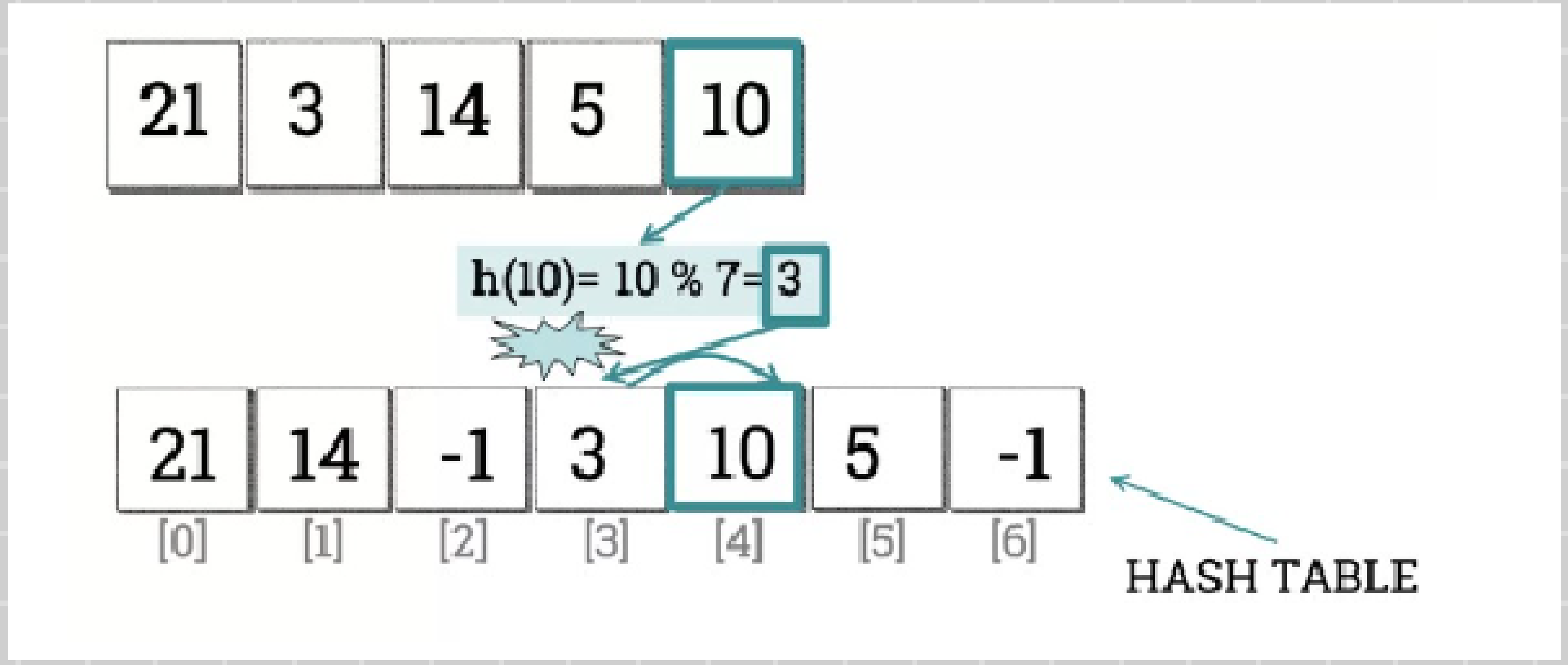


Method 2: Linear probing

-- Is when we move to the next available bucket in the even of a collision.

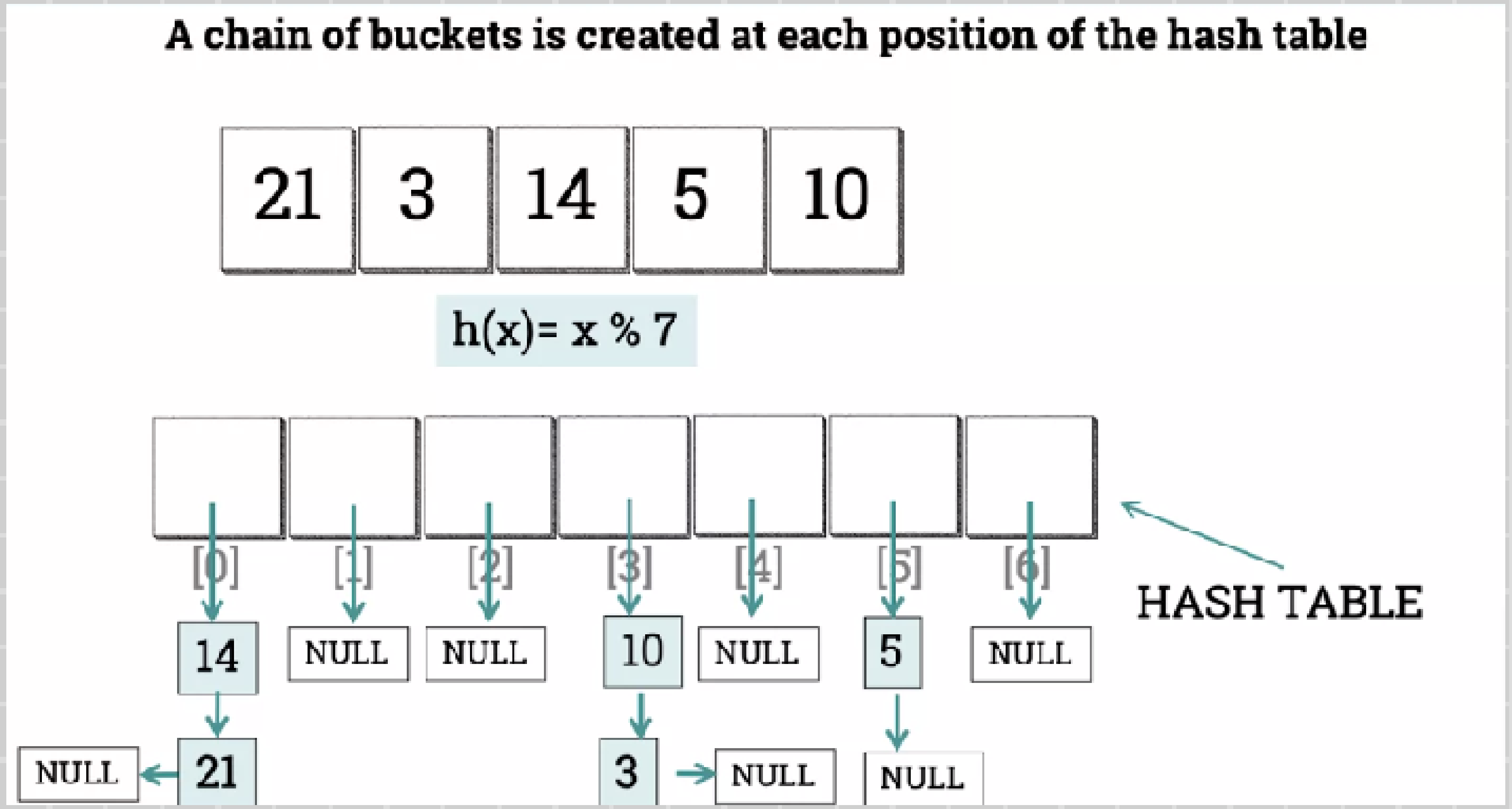


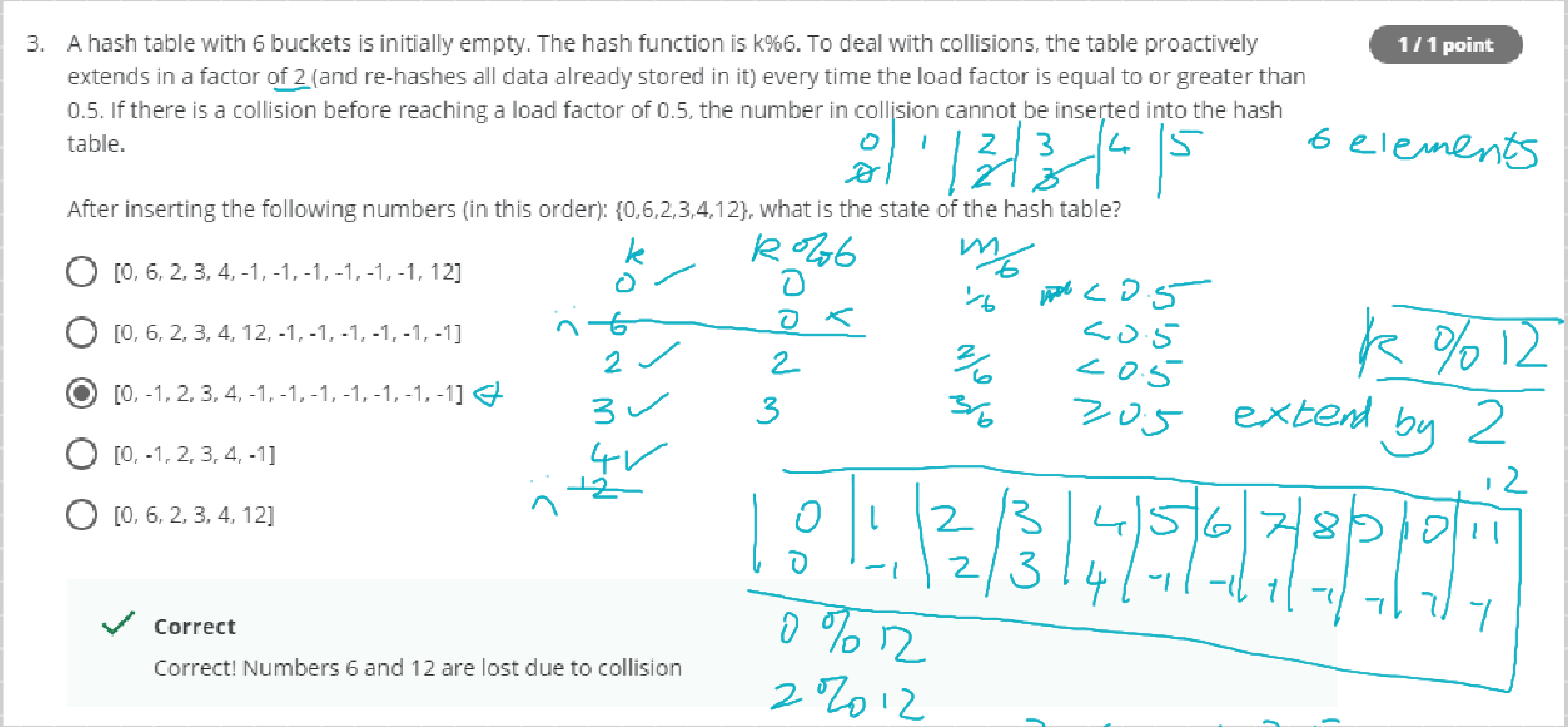
If we know we are using linear probing and we are searching for a value, and it isn't found at the index given by the hash function, we will simply check the next bucket until we encounter -1, or we have seached the entire array.



If searching the hash table where linear probing is used a solution to collisions we would have to modify our hashSearch function to avoid false negatives. You always have to modify your hash function based on the collision resolution used.

Method 3: Separate Chaining





3. A hash table with 6 buckets is initially empty. The hash function is k%6. To deal with collisions, the table proactively extends in a factor of 2 (and re-hashes all data already stored in it) every time the load factor is equal to or greater than 0.5. If there is a collision before reaching a load factor of 0.5, the number in collision cannot be inserted into the hash table.

1 / 1 point

After inserting the following numbers (in this order): {0,6,2,3,4,12}, what is the state of the hash table?

- ○ [0, 6, 2, 3, 4, -1, -1, -1, -1, -1, -1, 12]
- ○ [0, 6, 2, 3, 4, 12, -1, -1, -1, -1, -1, -1]
- [✓ [0, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1]] ⇐
- ○ [0, -1, 2, 3, 4, -1]
- ○ [0, 6, 2, 3, 4, 12]

✓ **Correct**

Correct! Numbers 6 and 12 are lost due to collision

0	1	2	3	4	5
~~0~~		2	~~3~~		

6 elements

k	k%6	m/6	
0 ✓	0	1/6	< 0.5
~~6~~	~~0~~ ×		< 0.5
2 ✓	2	2/6	< 0.5
3 ✓	3	3/6	≥ 0.5 extend by 2
4 ✓			
~~12~~			

k % 12

0	1	2	3	4	5	6	7	8	9	10	11
0	-1	2	3	4	-1	-1	-1	-1	-1	-1	-1

12

0 % 12

2 % 12

3 % 12 $\quad 3/12 < 0.5$

4 % 12 $\quad 4/12 < 0.5$

~~12~~ % 12