ADVANCED DATA STRUCTURES AND ALGORITHMS

Associate Professor Dr. Raed Ibraheem Hamed

University of Human Development, College of Science and Technology
Computer Science Department

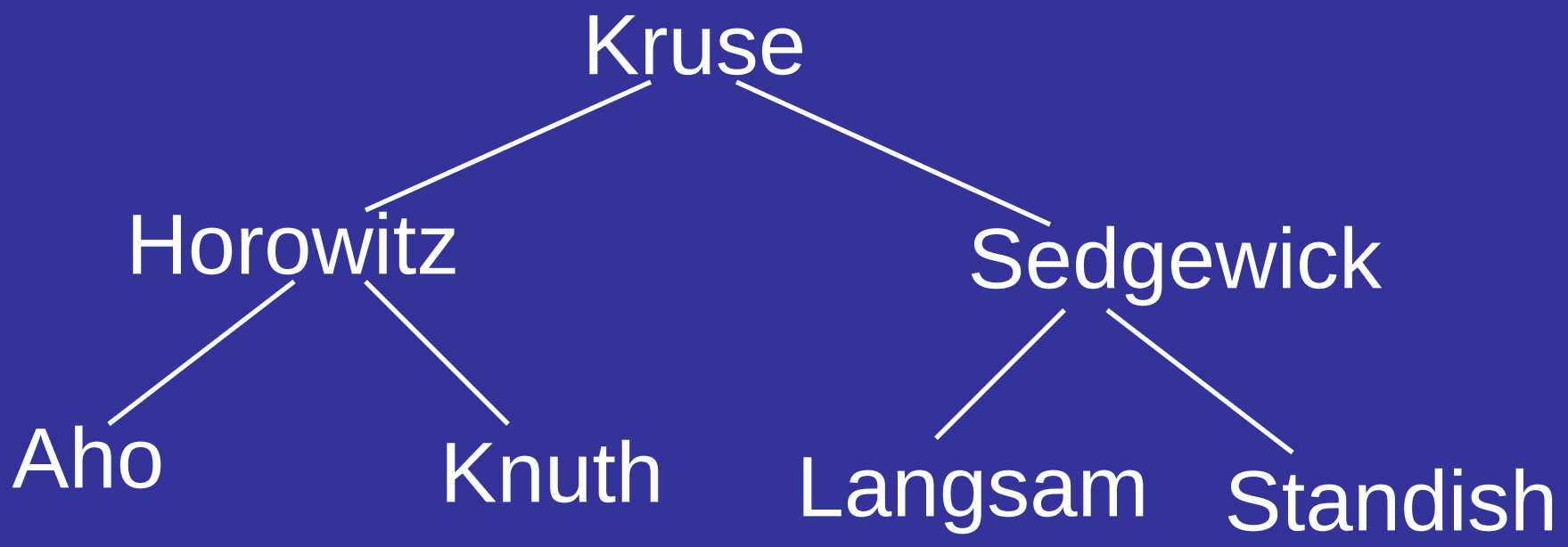
2015 – 2016

Overview

- ✓ Information Retrieval
- ✓ Review: Binary Search Trees
- ✓ Hashing.
- ✓ Applications.
- ✓ Example.
- ✓ Hash Functions.
- ✓ Collisions
- ✓ Linear Probing
- ✓ Problems with Linear Probing
- ✓ Chaining

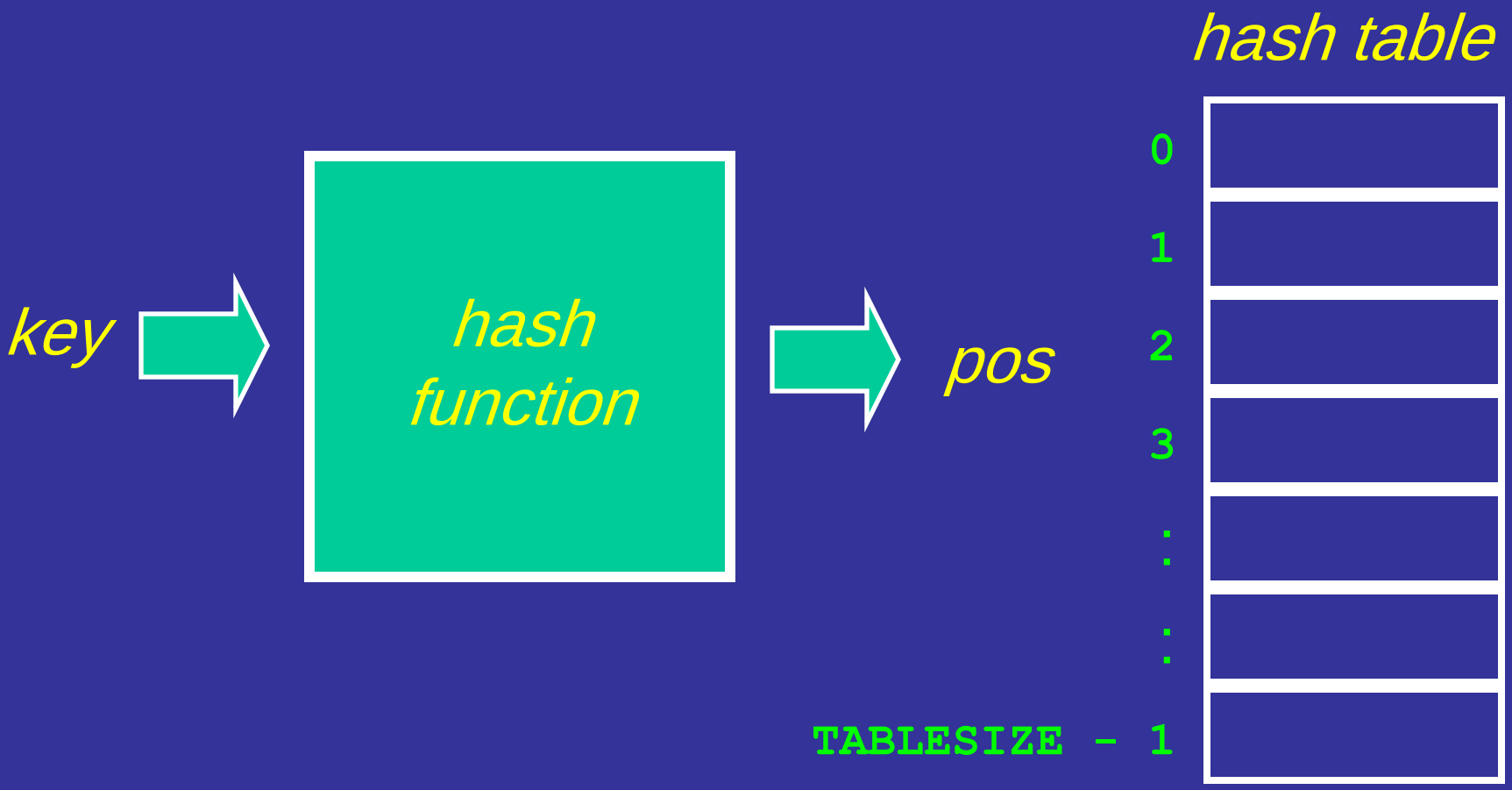
Hashing

Kruse	Horowitz	Sedgewick	Aho	Knuth	Langsam	Standish

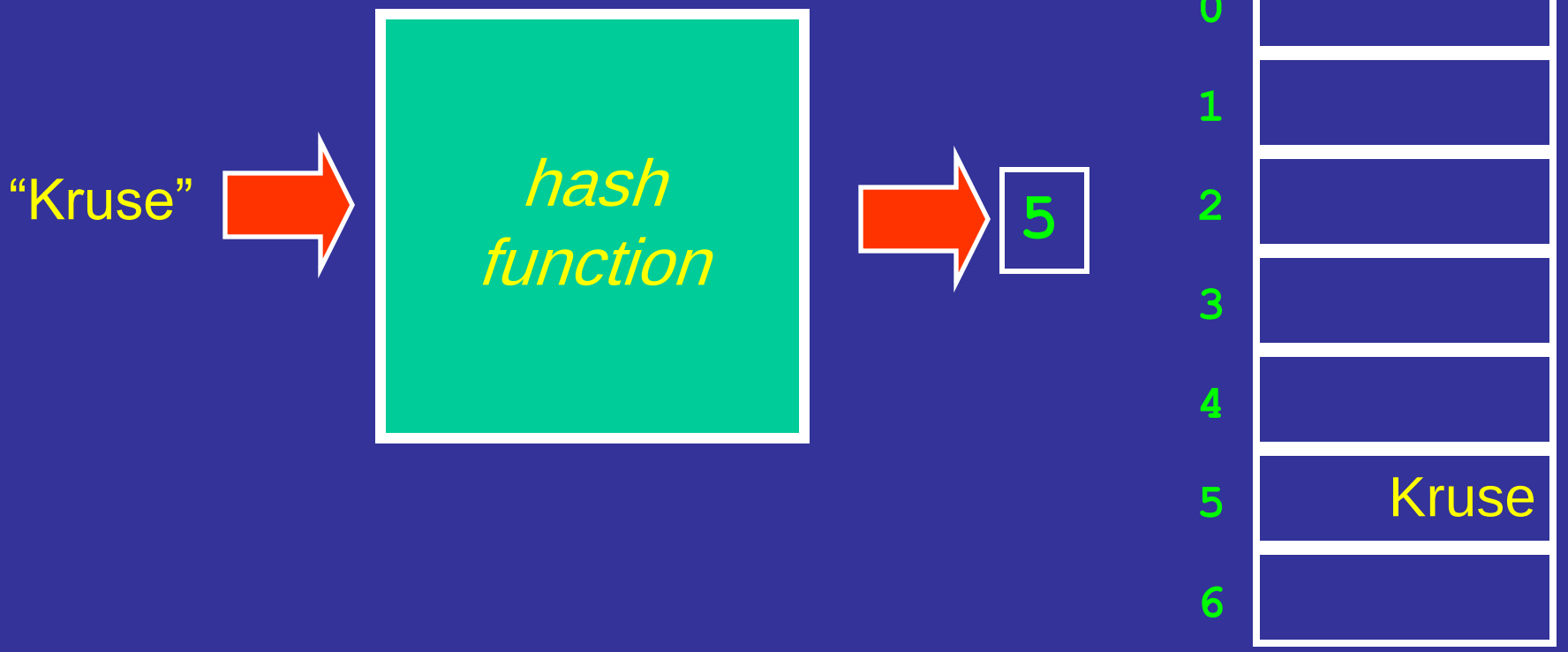


Insert the information into a Binary Search Tree, using the name as the key

Hashing



Example:



Hash Function

- Maps keys to positions in the Hash Table.
- Be easy to calculate.
- Use all of the key.
- Spread the keys uniformly.
- Each item has a unique key.
- Use a large array called a Hash Table.
- Use a Hash Function.

Example: Hash Function #1

```
unsigned hash(char* s)
{
        int i = 0;
            unsigned value = 0;

                    while (s[i] != '\0')
    {
                value = (s[i] + 31*value) % 101;
          i++;
    }
            return value;
}
```

Example: Hash Function #1

value = (s[i] + 31*value) % 101;

- A. **Aho**, J. Hopcroft, J. Ullman, *"Data Structures and Algorithms"*, 1983, Addison-Wesley.

'A' = 65 'h' = 104 'o' = 111

```
value = (65 + 31 * 0) % 101 = 65
value = (104 + 31 * 65) % 101 = 99
value = (111 + 31 * 99) % 101 = 49
```

Key Codes

a = 97	shift + a = A = 65
b = 98	shift + b = B = 66
c = 99	shift + c = C = 67
d = 100	shift + d = D = 68
e = 101	shift + e = E = 69
f = 102	shift + f = F = 70
g = 103	shift + g = G = 71
h = 104	shift + h = H = 72
i = 105	shift + i = I = 73
j = 106	shift + j = J = 74
k = 107	shift + k = K = 75
l = 108	shift + l = L = 76
m = 109	shift + m = M = 77
n = 110	shift + n = N = 78
o = 111	shift + o = O = 79
p = 112	shift + p = P = 80
q = 113	shift + q = Q = 81
r = 114	shift + r = R = 82
s = 115	shift + s = S = 83
t = 116	shift + t = T = 84
u = 117	shift + u = U = 85
v = 118	shift + v = V = 86
w = 119	shift + w = W = 87
x = 120	shift + x = X = 88
y = 121	shift + y = Y = 89
z = 122	shift + z = Z = 90

Example: Hash Function #1

value = (s[i] + 31*value) % 101;

Key	Hash Value
Aho	49
Kruse	95
Standish	60
Horowitz	28
Langsam	21
Sedgewick	24
Knuth	44

resulting table is "sparse"

Example: Hash Function #2

value = (s[i] + 1024*value) % 128;

Key	Hash Value
Aho	111
Kruse	101
Standish	104
Horowitz	122
Langsam	109
Sedgewick	107
Knuth	104

likely to result in "clustering"

Example: Hash Function #3

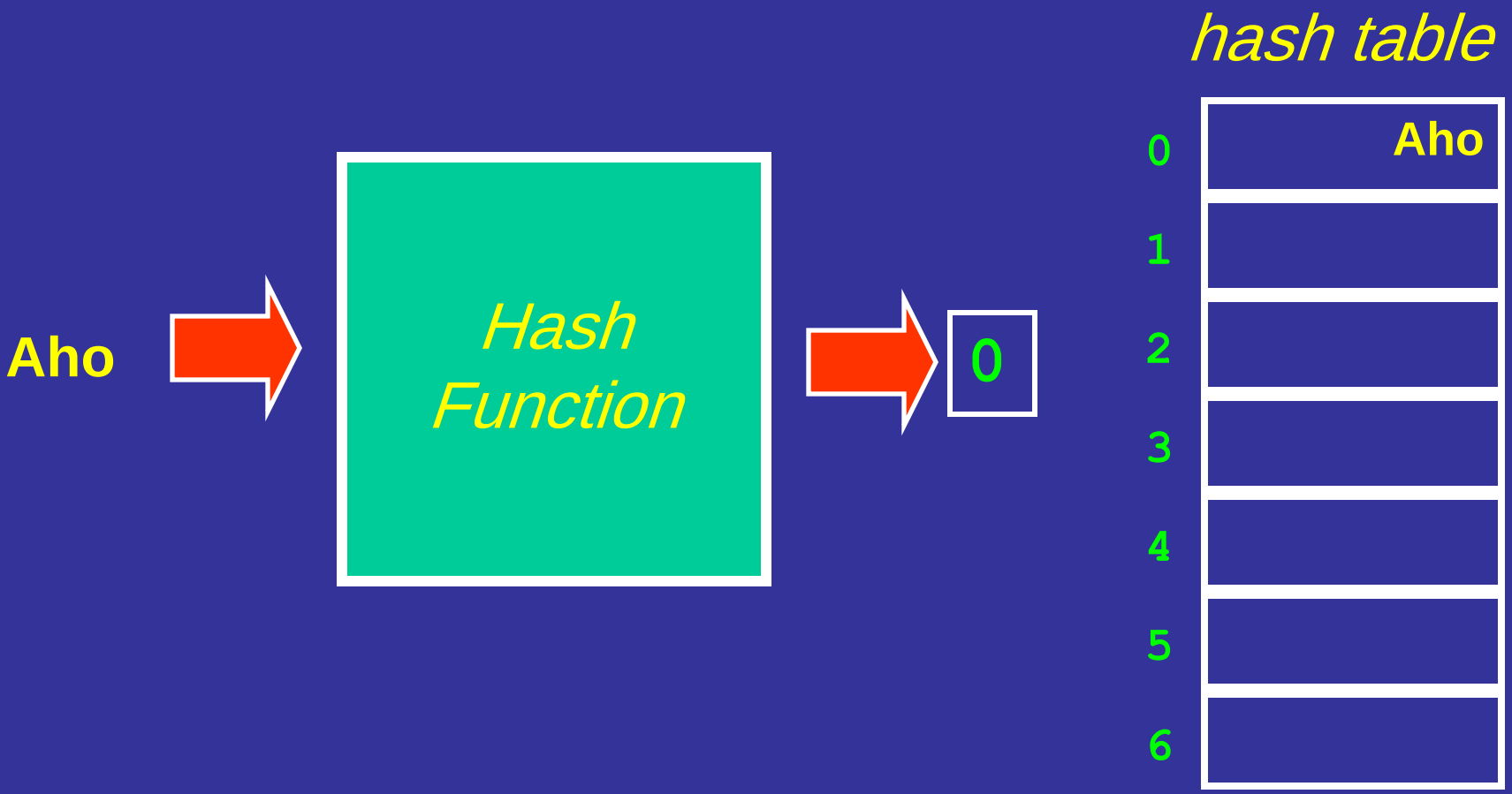
value = (s[i] + 3*value) % 7;

Key	Hash Value
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

"collisions"

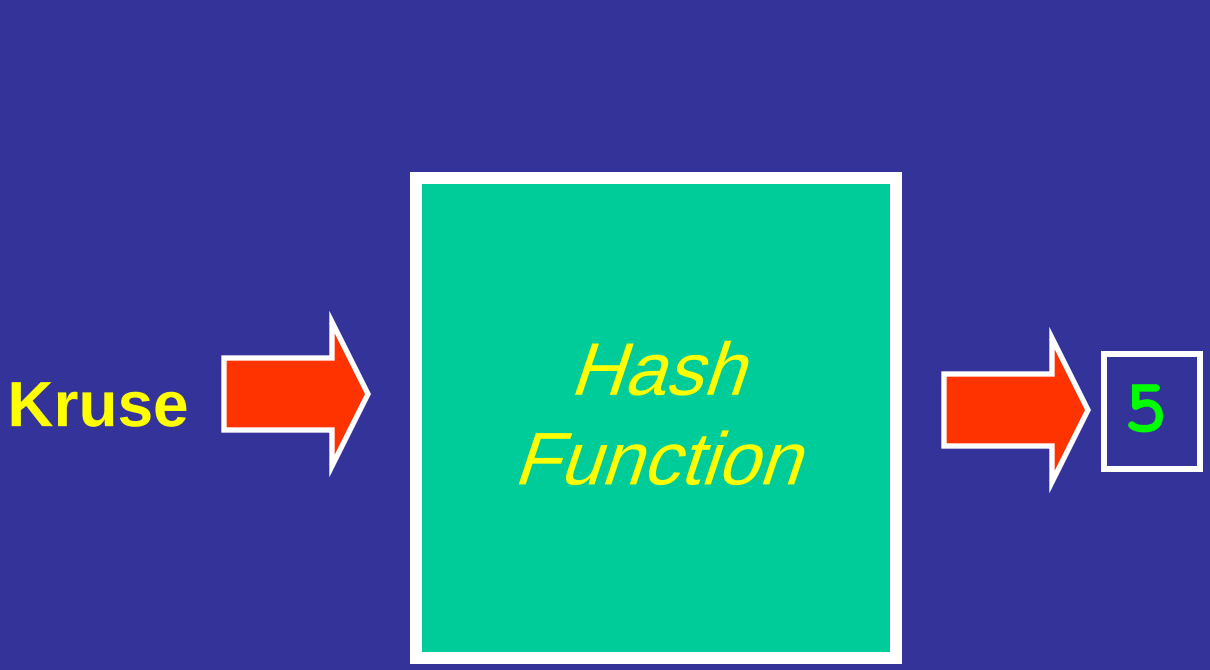
Example: Insert

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Example: *Insert*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth

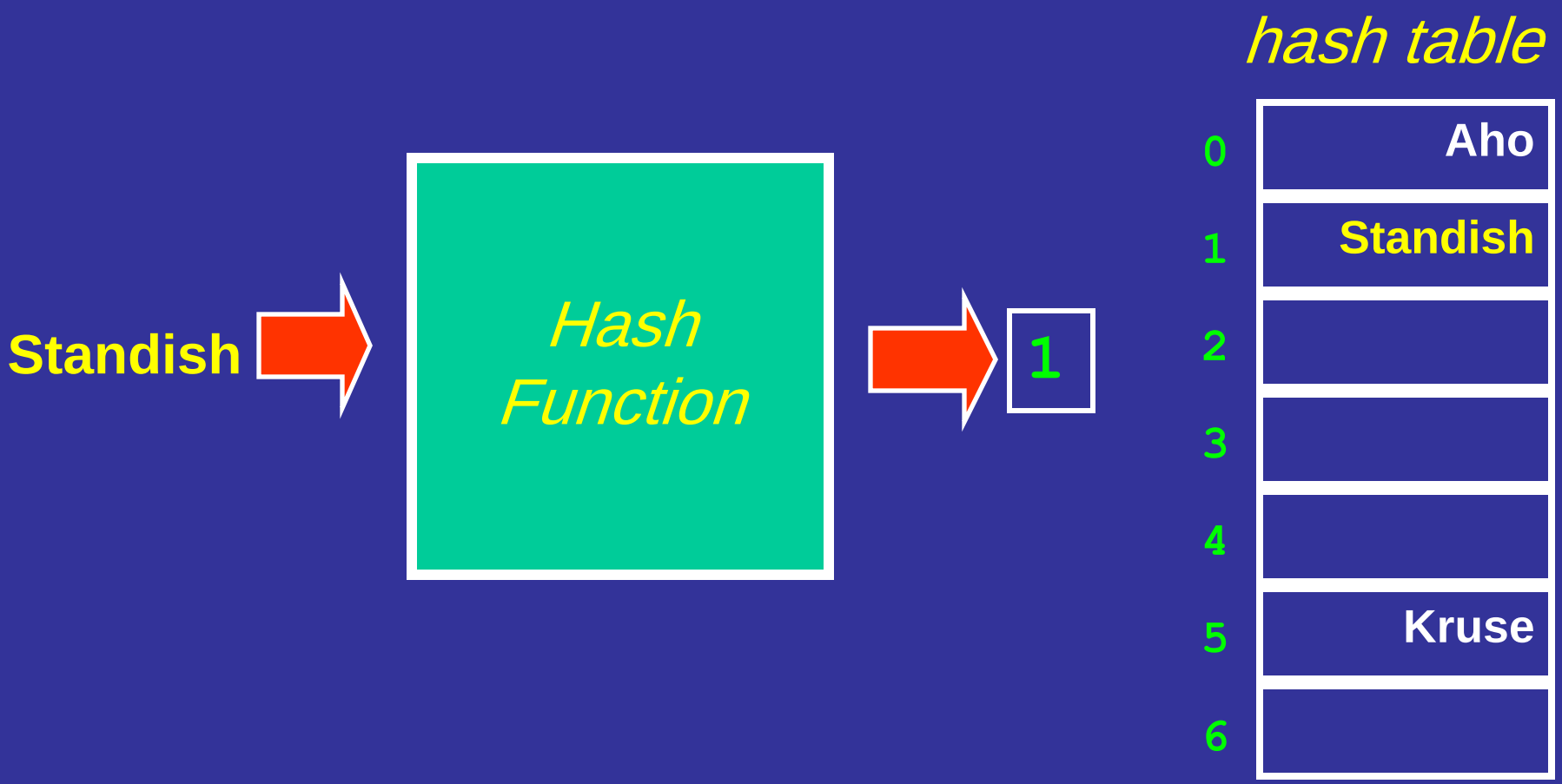


hash table

0	Aho
1	
2	
3	
4	
5	Kruse
6	

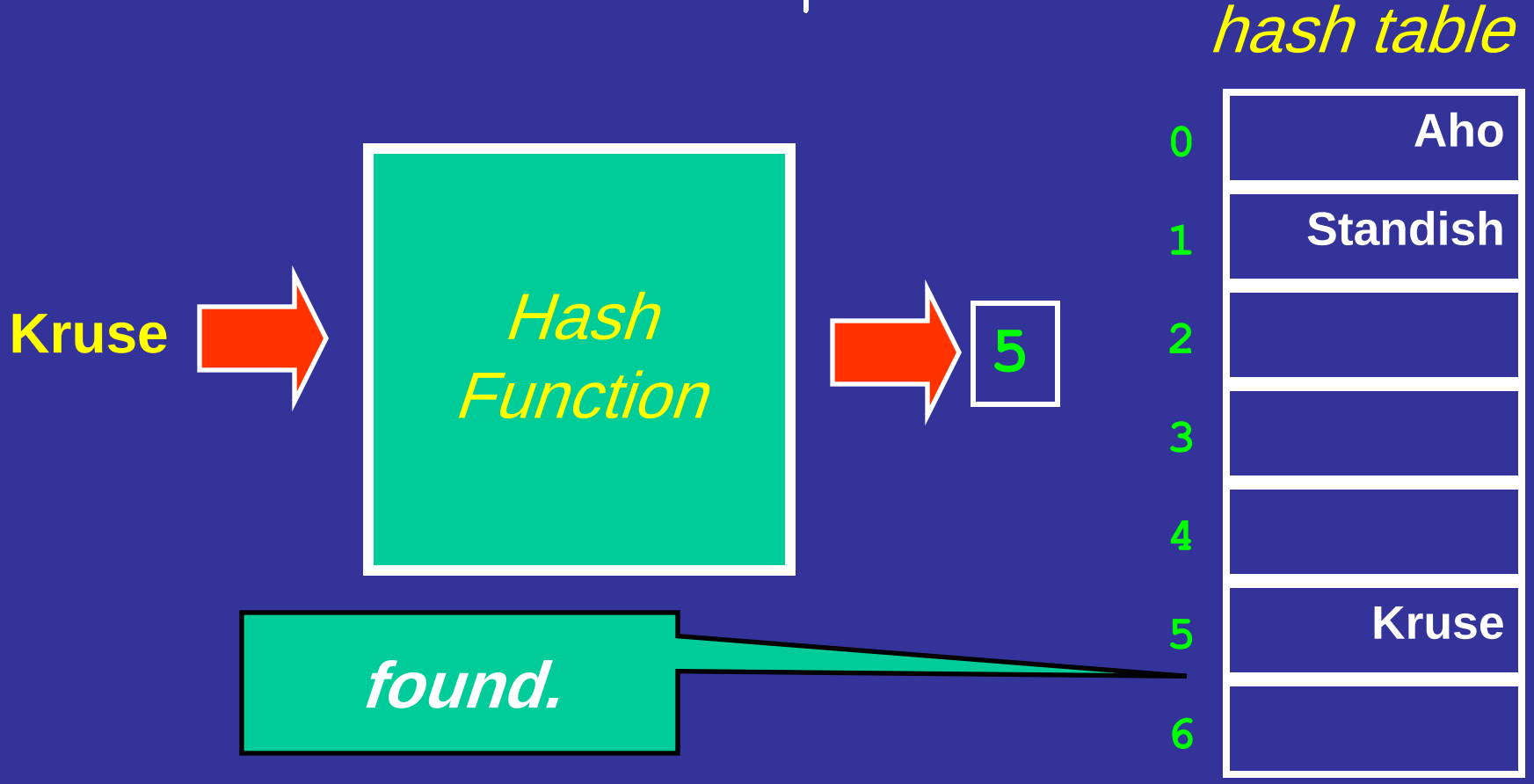
Example: Insert

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



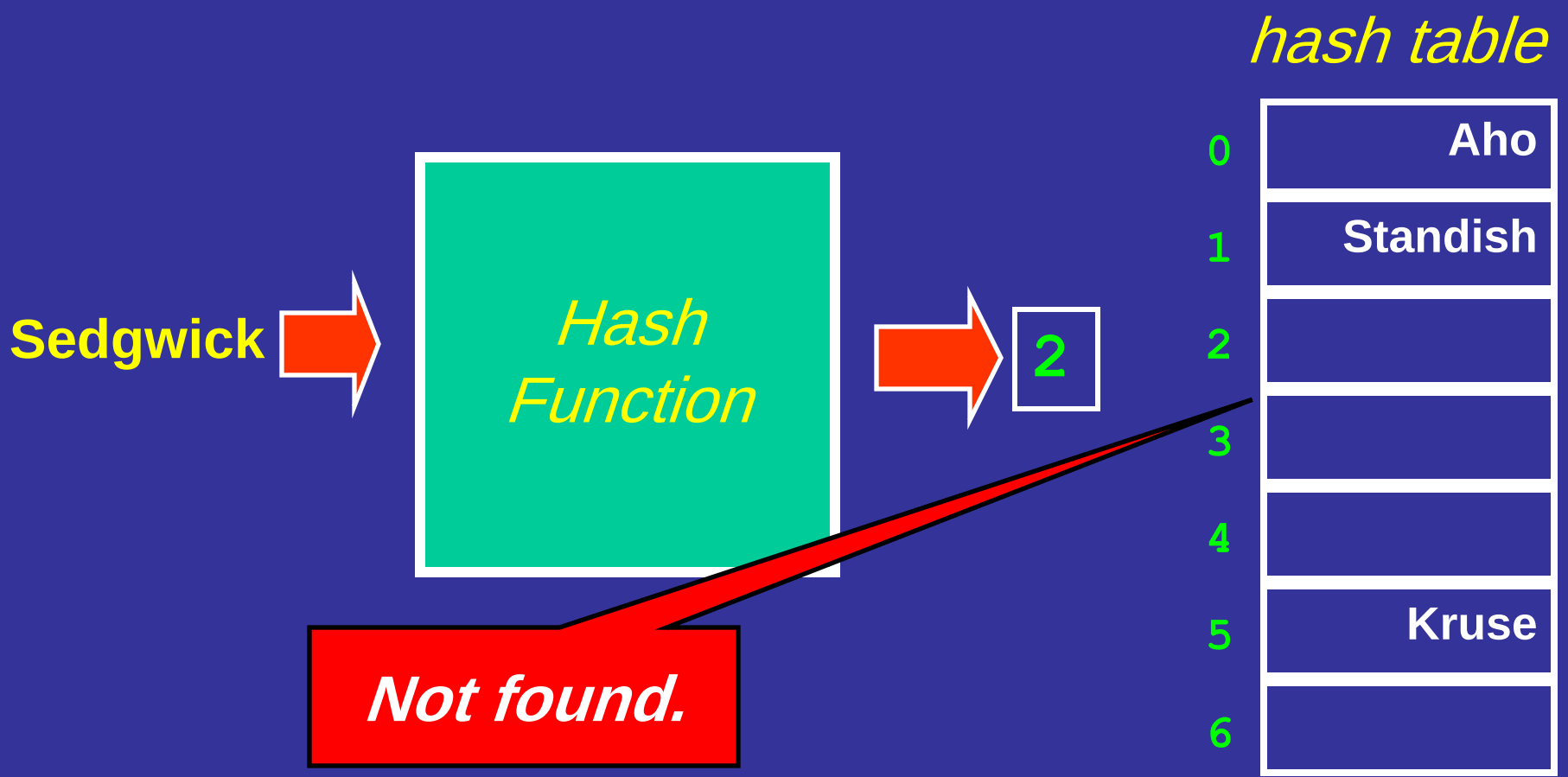
Example: Search

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Example: Search

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth

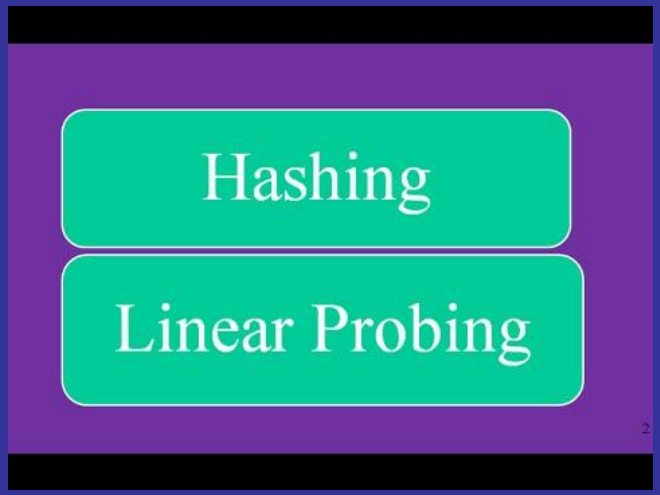


Insert with Linear Probing

- Apply hash function to get a position.
- Try to insert key at this position.
- Deal with collision.
  - *Must also deal with a full table!*
- Two methods are commonly used:
  - Linear Probing.
  - Chaining.

What is Linear probing

Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key.



Hashing with Linear Probe

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.
If an empty slot is not found before reaching the point of collision, the table is full.

Hashing with Linear Probe

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

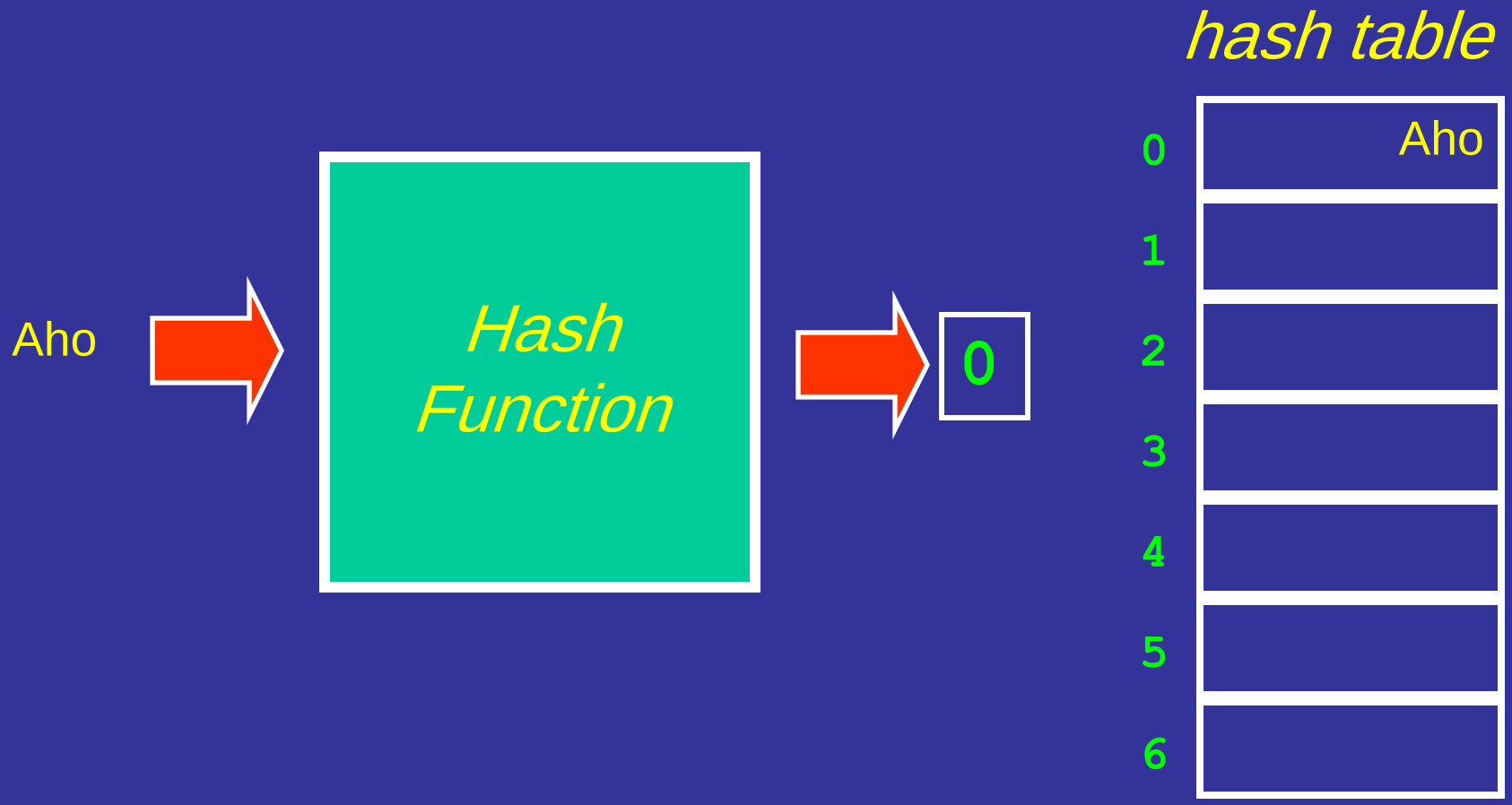
5 = 5 % 8

7 = 15 % 8

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

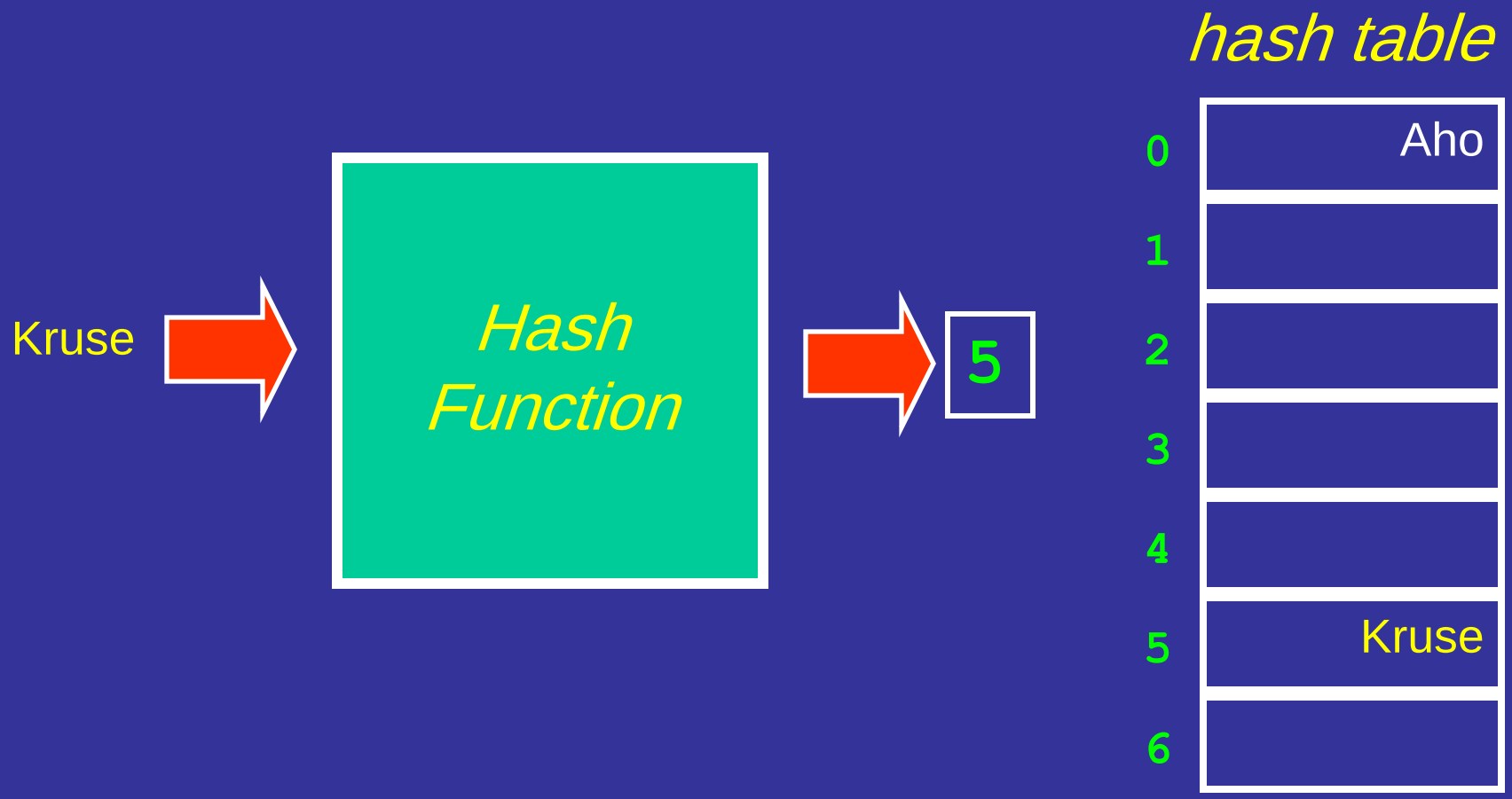
Example: *Insert with Linear Probing*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



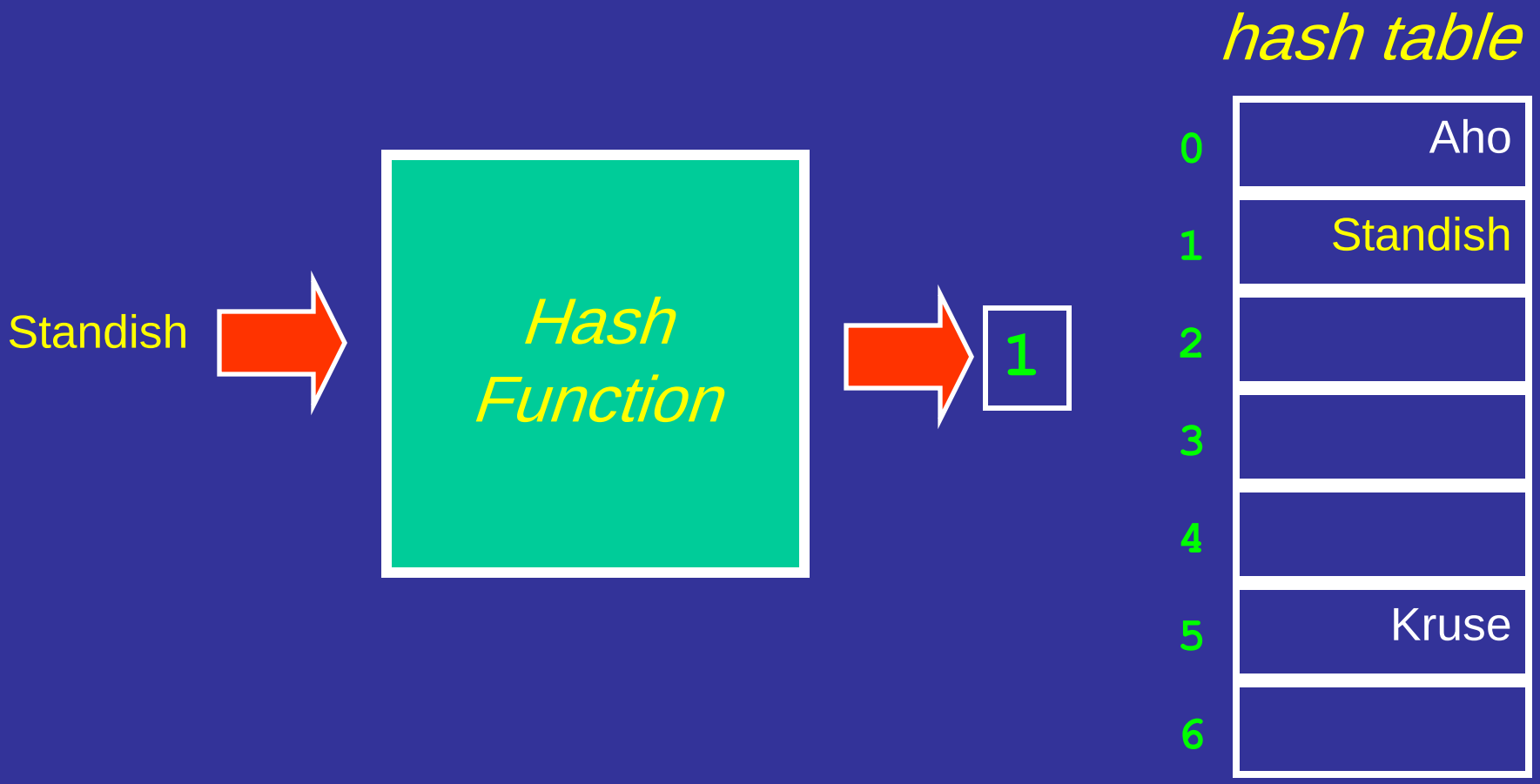
Example: *Insert with Linear Probing*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



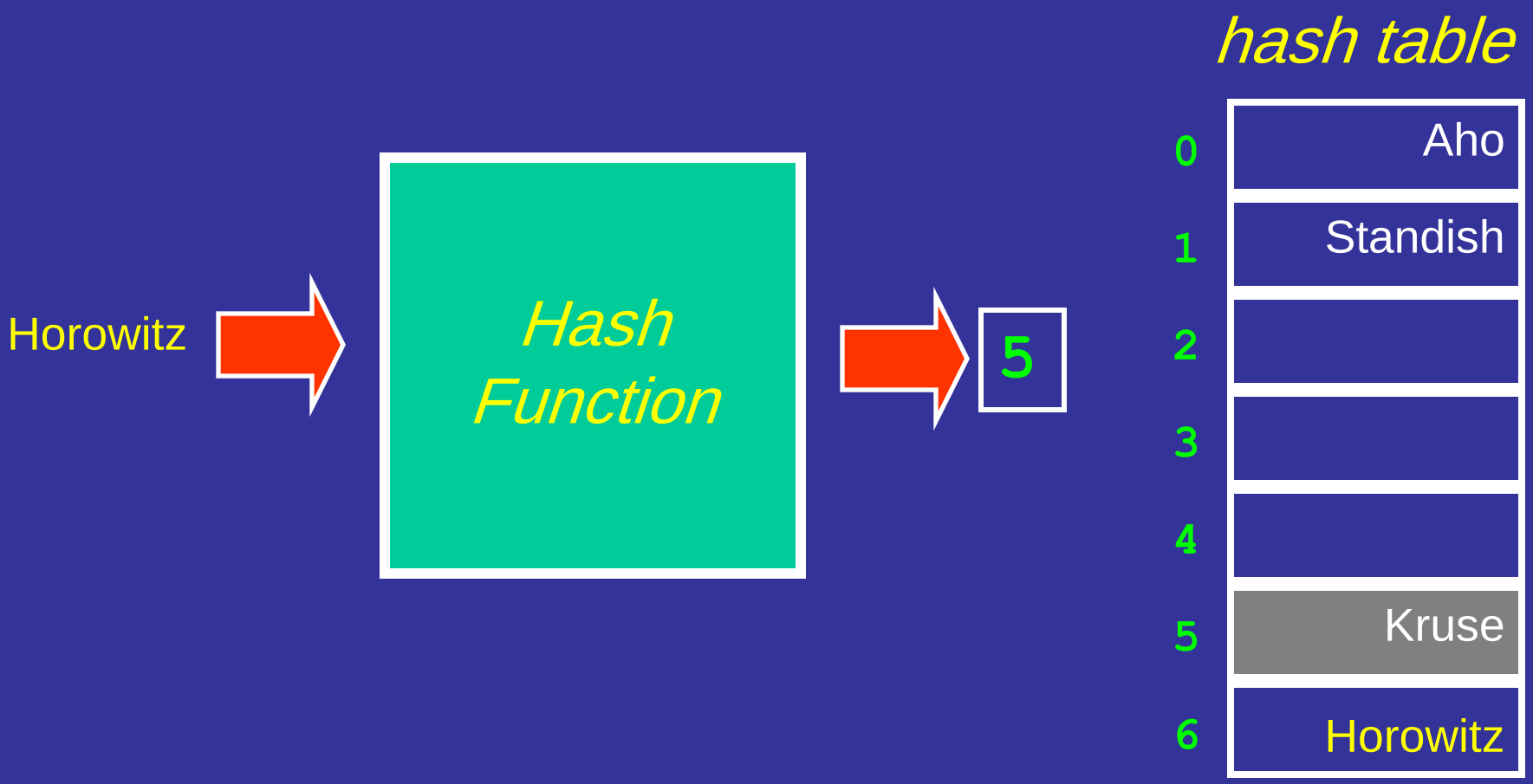
Example: *Insert with Linear Probing*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



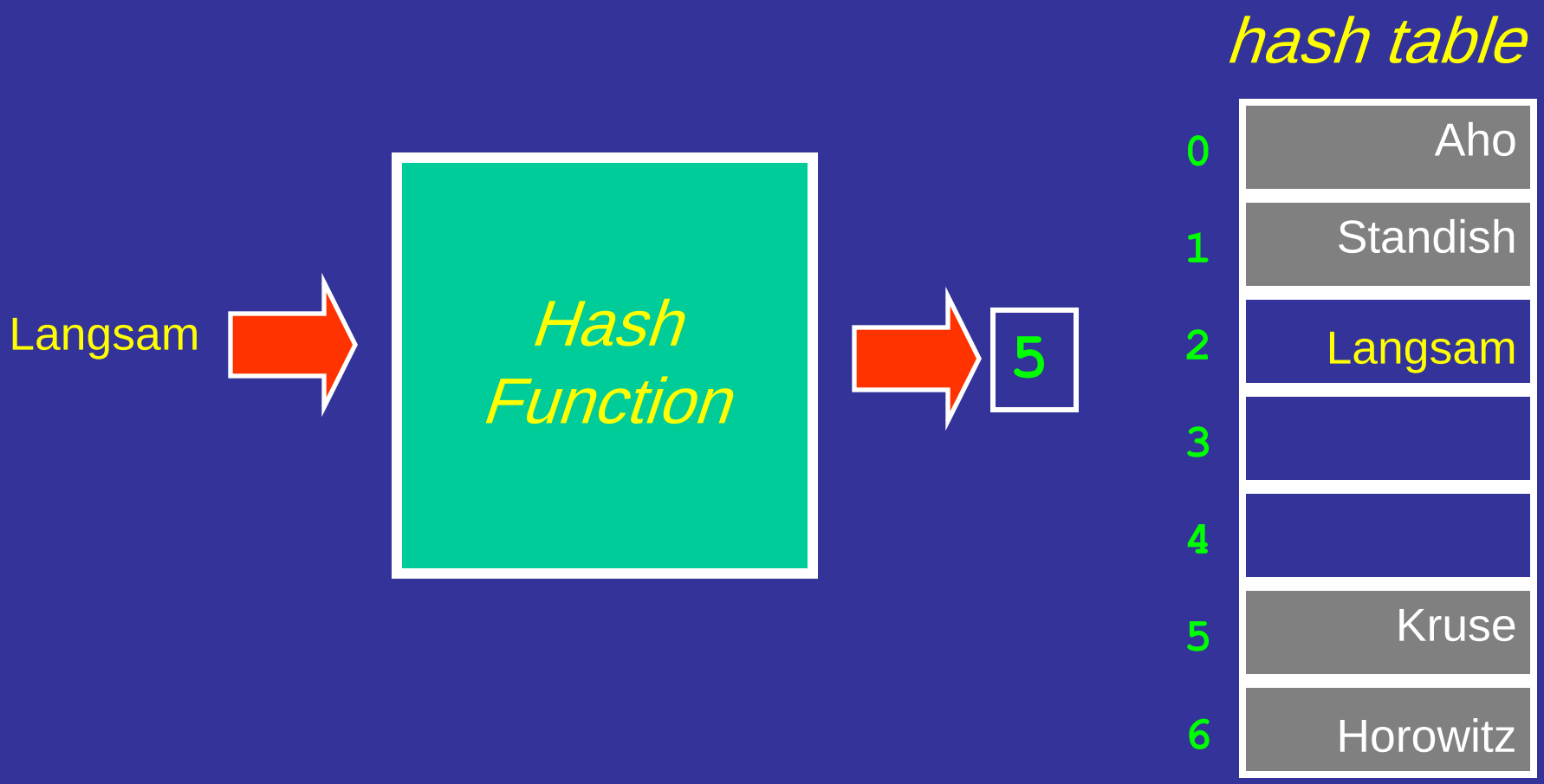
Example: *Insert with Linear Probing*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Example: *Insert with Linear Probing*

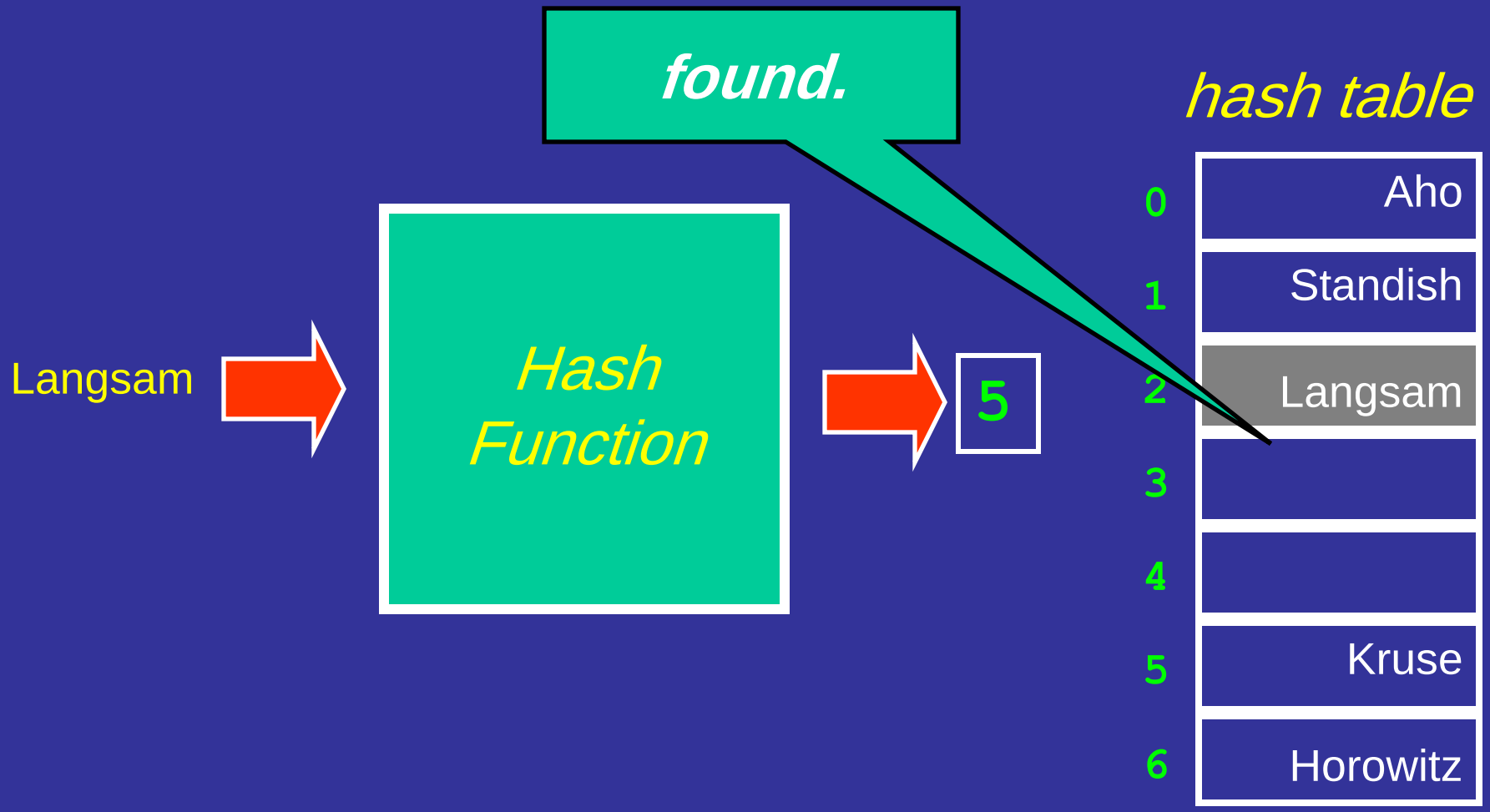
Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



```
module linearProbe(item)
{
   position = hash(key of item)
   count = 0
   loop {
      if (count == hashTableSize) then {
         output “Table is full”
         exit loop
      }
      if (hashTable[position] is empty) then {
          hashTable[position] = item
          exit loop
      }
      position = (position + 1) % hashTableSize
      count++
   }
}
```

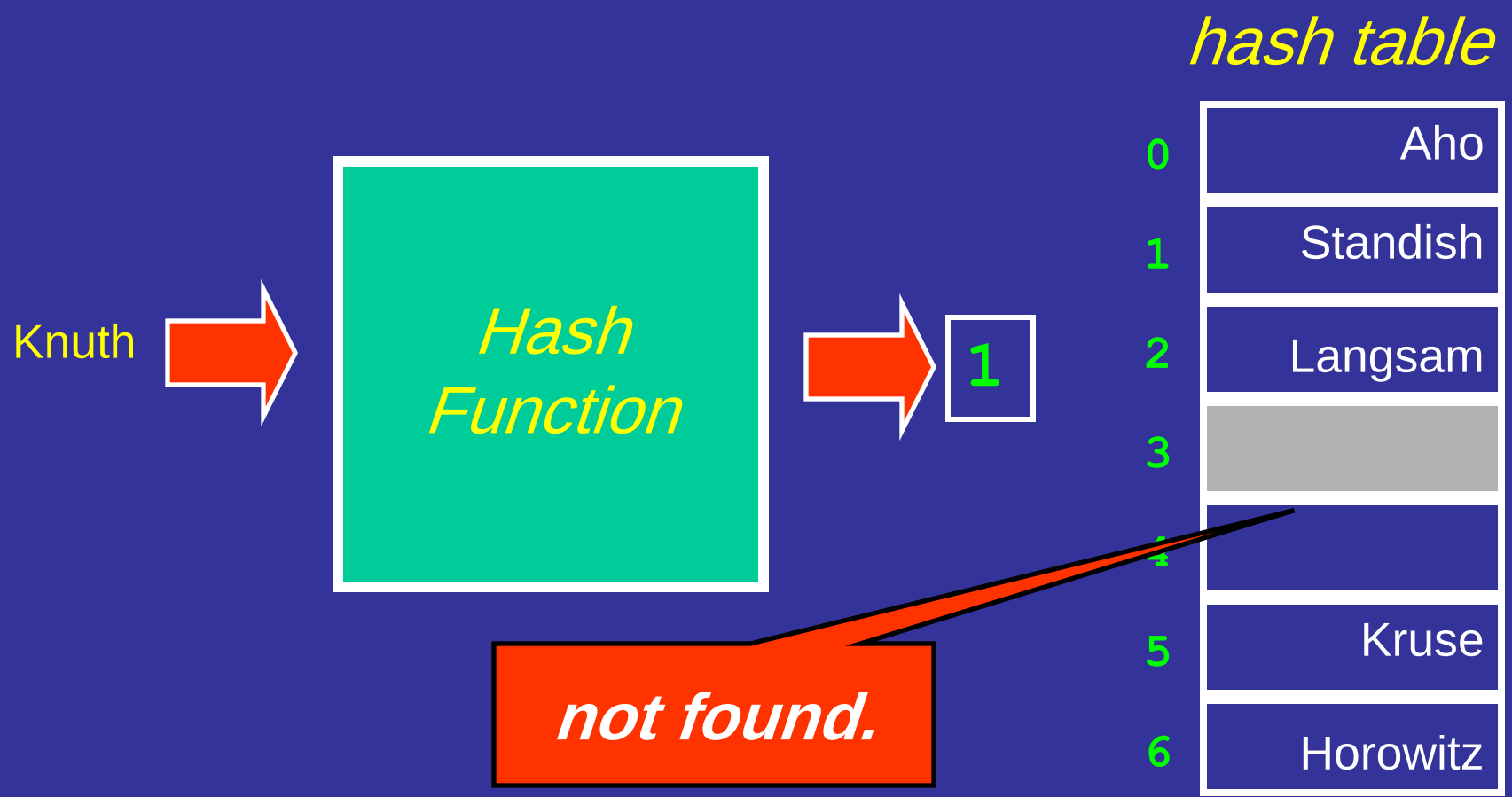
Example: ***Search*** *with Linear Probing*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



Example: ***Search*** *with Linear Probing*

Aho, Kruse, Standish, Horowiz, Langsam, Sedgewick, Knuth



```
module search(target)
{
   count = 0
   position = hash(key of target)
   loop {
      if (count == hashTableSize) then {
         output “Target is not in Hash Table”
         return  -1.
      }
      else if (hashTable[position] is empty) then {
         output “Item is not in Hash Table”
         return  -1.
      }
      else if (hashTable[position].key == target) then {
          return position.
      }
      position = (position + 1) % hashTableSize
      count++
   }
}
```

Delete with Linear Probing

- Use the search function to find the item
- If found check that items after that also don't hash to the item's position
- If items after do hash to that position, move them back in the hash table and delete the item.

Very difficult and time/resource consuming!

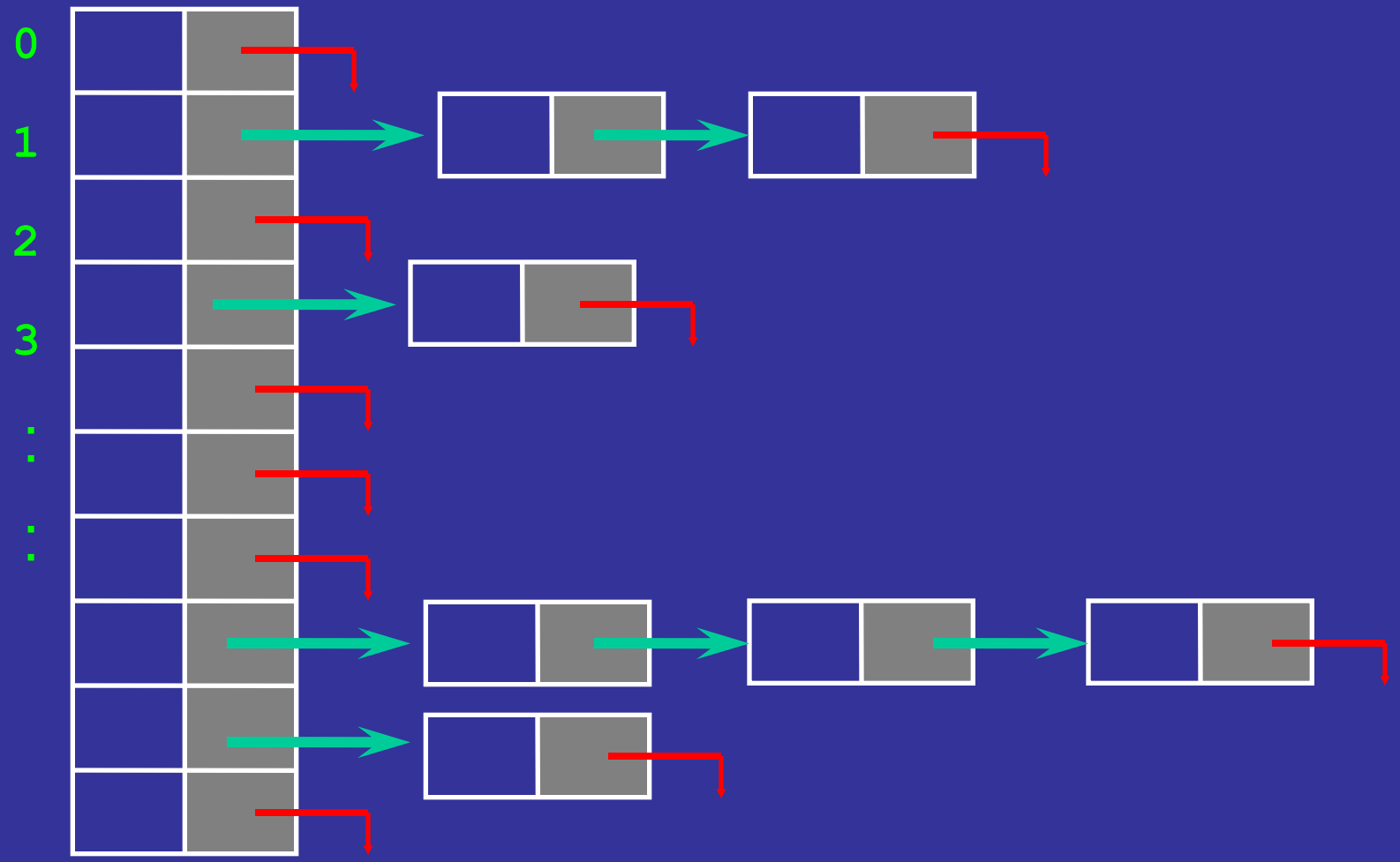
Linear Probing: Problems

- Speed.
- Tendency for clustering to occur as the table becomes half full.
- Deletion of records is very difficult.
- If implemented in arrays – table may become full fairly quickly, resizing is time and resource consuming

Chaining

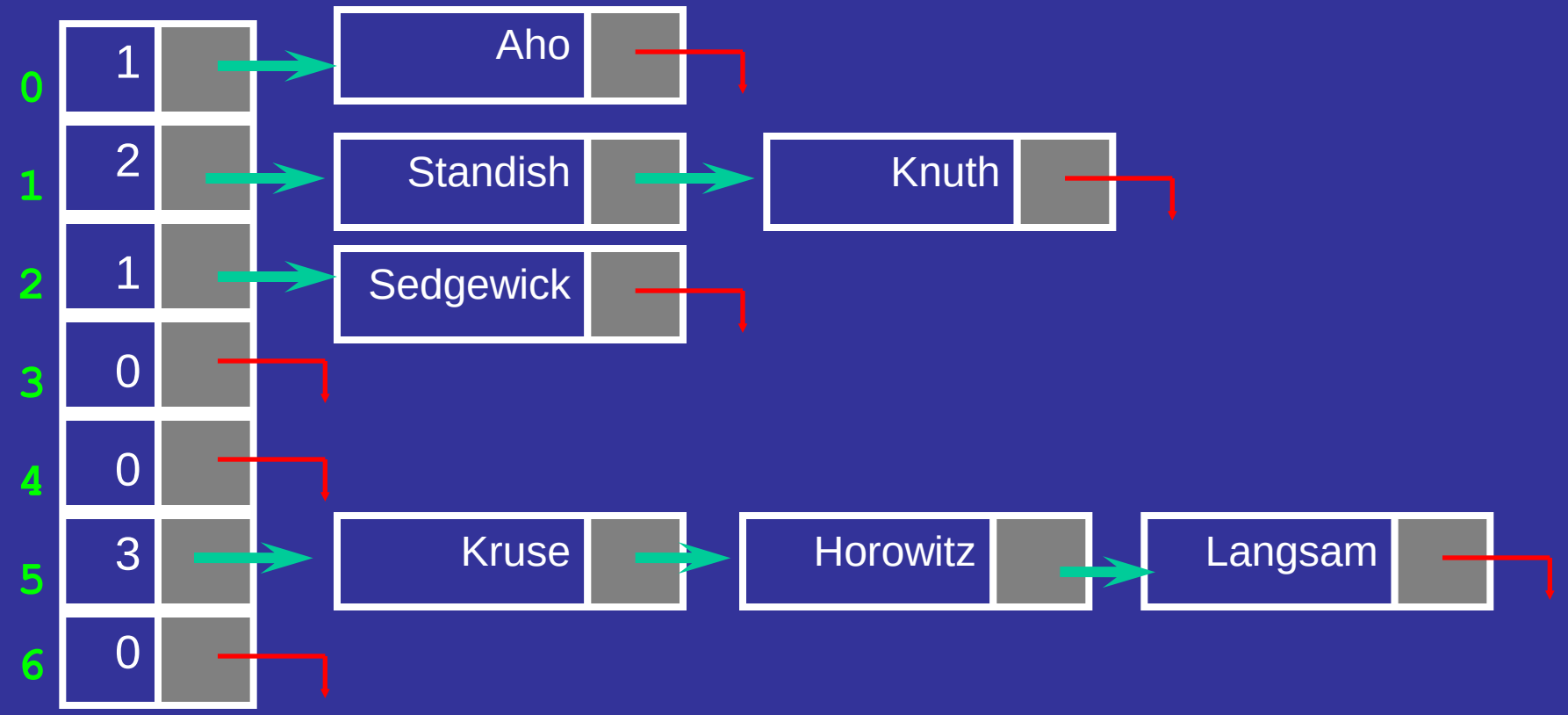
- Uses a Linked List at each position in the Hash Table.
  - Linked list at a position contains all the items that ‘hash’ to that position.
  - May keep linked lists sorted or not.

hash table



Example: Chaining

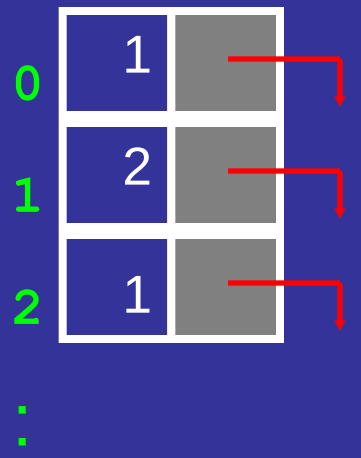
Aho,	Kruse,	Standish,	Horowiz,	Langsam,	Sedgwick,	Knuth
0,	5,	1,	5,	5,	2,	1



Hashtable with Chaining

- At each position in the array you have a list:

```
List hashTable[MAXTABLE];
```



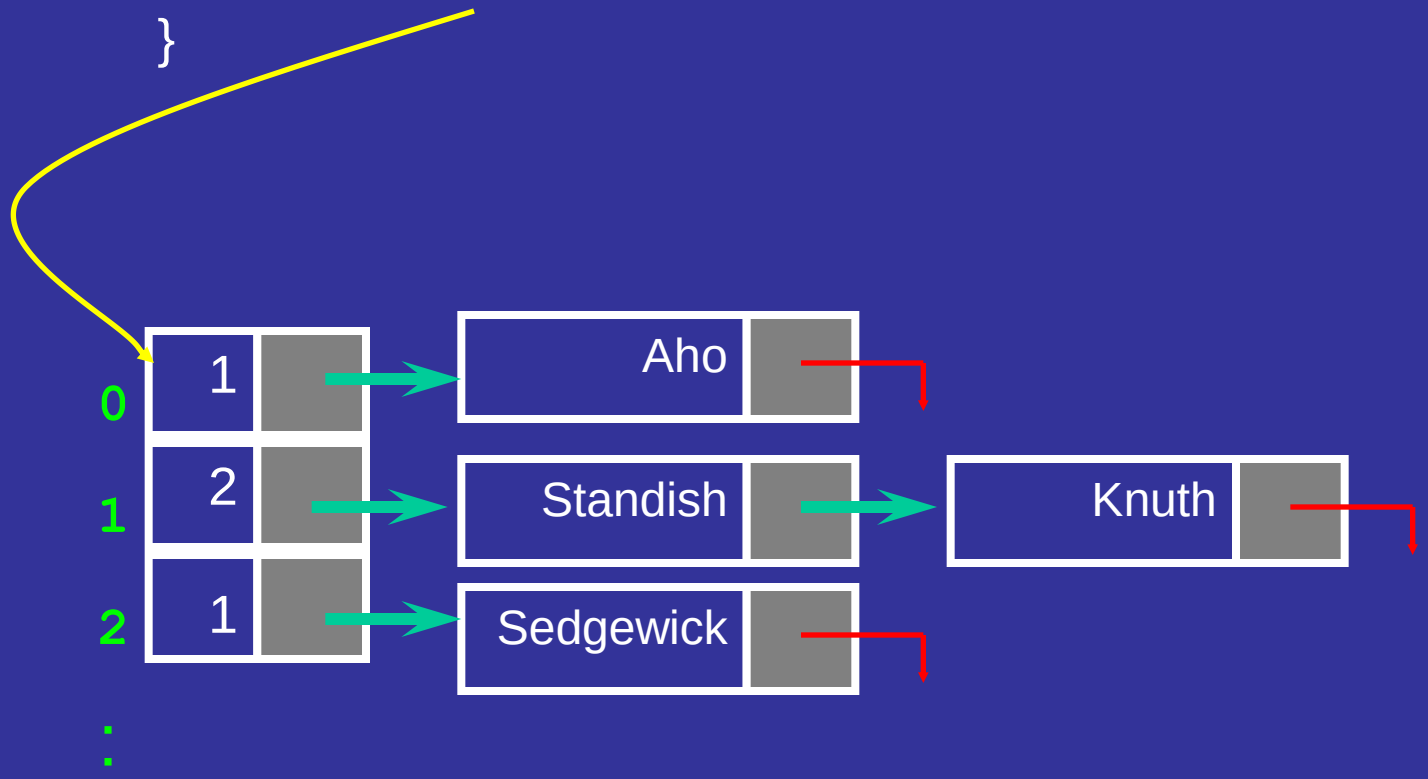
- You must initialise each list in the table.

Insert with Chaining

```
module InsertChaining(item)
{
   posHash = hash(key of item)

   insert (hashTable[posHash], item);

}
```



Search with Chaining

- Apply hash function to get a position in the array.
- Search the Linked List at this position in the array.

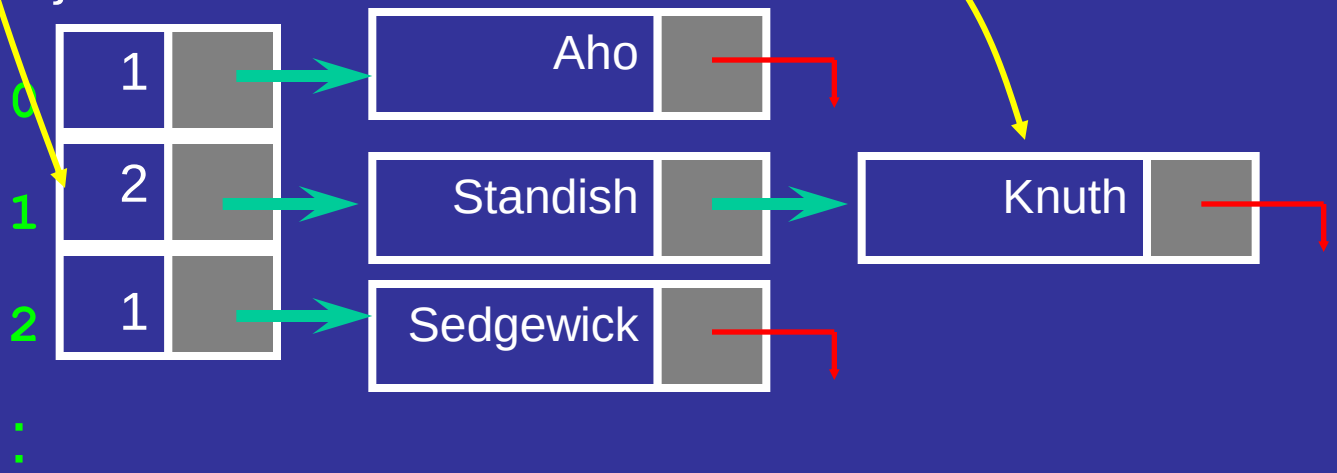
```
/* module returns NULL if not found, or the address of the
 *  node if found */

module SearchChaining(item)
{
   posHash = hash(key of item)
   Node* found;

   found = searchList (hashTable[posHash], item);

   return found;

}
```



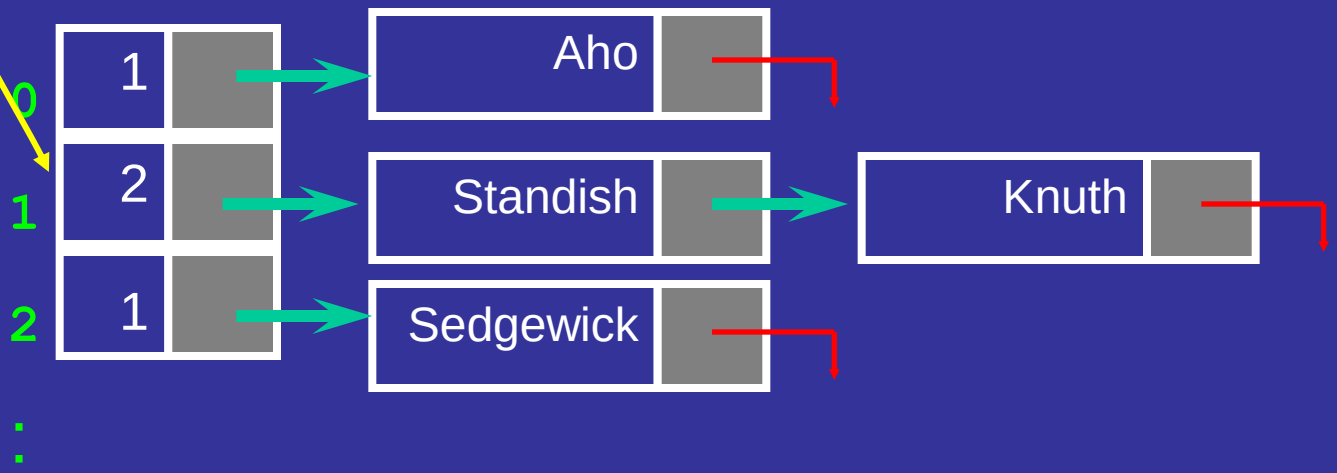
Delete with Chaining

- Apply hash function to get a position in the array.
- Delete the node in the Linked List at this position in the array.

```
/* module uses the Linked list delete function to delete an item
 *inside that list, it does nothing if that item isn't there.  */

module DeleteChaining(item)
{
   posHash = hash(key of item)

   deleteList (hashTable[posHash], item);
}
```



Disadvantages of Chaining

- Uses more space.
- More complex to implement.
  - Contains a linked list at every element in the array.
  - Requires linear searching.
  - May be time consuming.

Advantages of Chaining

- Insertions and Deletions are easy and quick.
- Allows more records to be stored.
- Naturally resizable, allows a varying number of records to be stored.

Thank You

???