

Collision Resolution

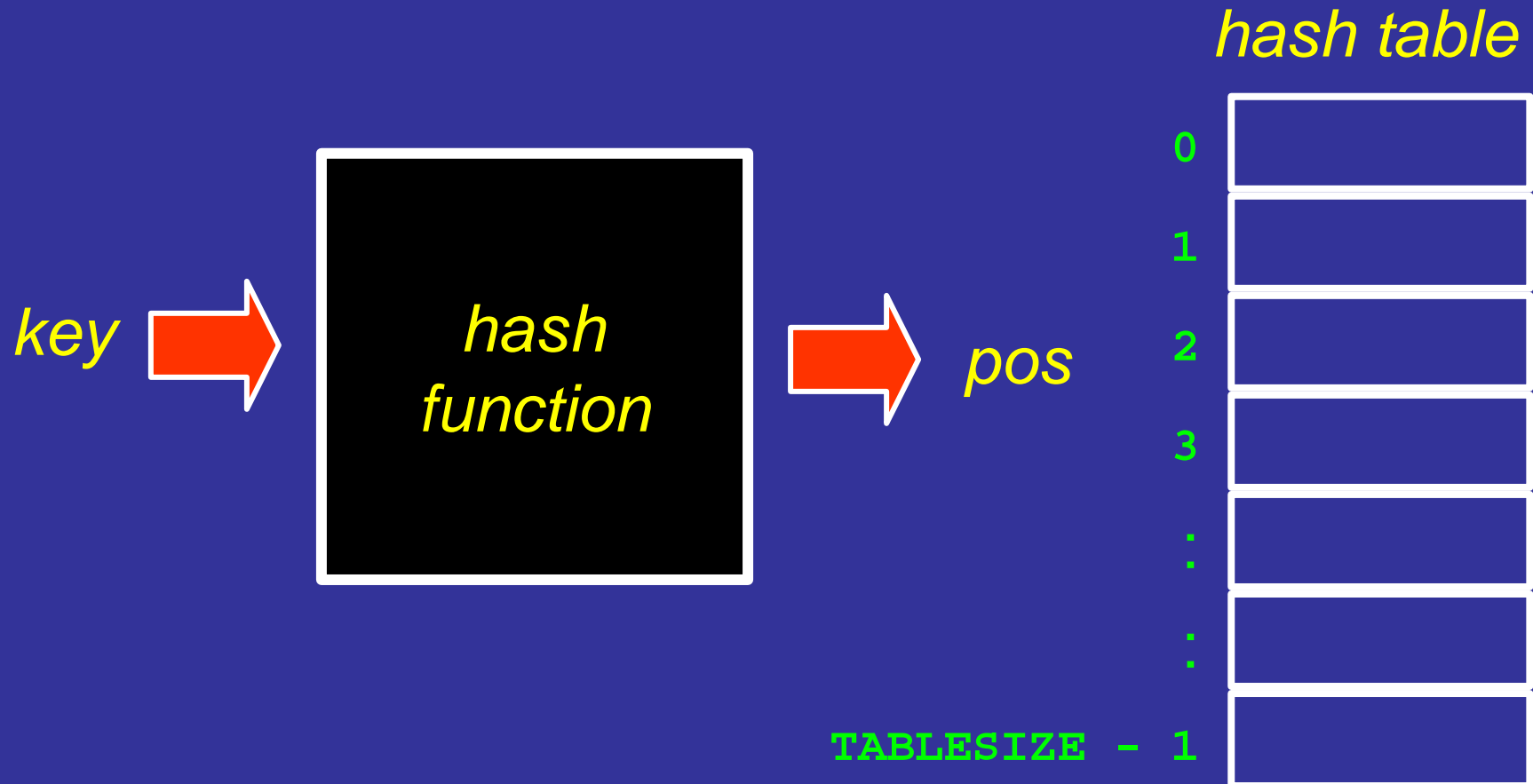
CSE1303 Part A

Data Structures and Algorithms

Overview

- Hash Tables
- Collisions
- Linear Probing
- Problems with Linear Probing
- Chaining

Hashing



Example:



hash table

0	
1	
2	
3	
4	
5	Kruse
6	

Hashing

- Each item has a **unique key**.
- Use a large array called a **Hash Table**.
- Use a **Hash Function**.

Operations

- Initialize
 - all locations in Hash Table are **empty**.
- Insert
- Search
- Delete

Hash Function

- **Maps** keys to positions in the Hash Table.
- Be **easy** to calculate.
- Use **all** of the key.
- Spread the keys **uniformly**.

Example: Hash Function #3

$$\text{value} = (\text{s}[i] + 3 * \text{value}) \% 7;$$

Aha	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

“collisions”

Collision

- When two keys are mapped to the same position.
- Very likely.

Birthdays

Number of People	Probability
10	0.1169
20	0.4114
30	0.7063
40	0.8912
50	0.9704
60	0.9941
70	0.9992

Collision Resolution

- Linear Probing.
- Chaining

Linear Probing

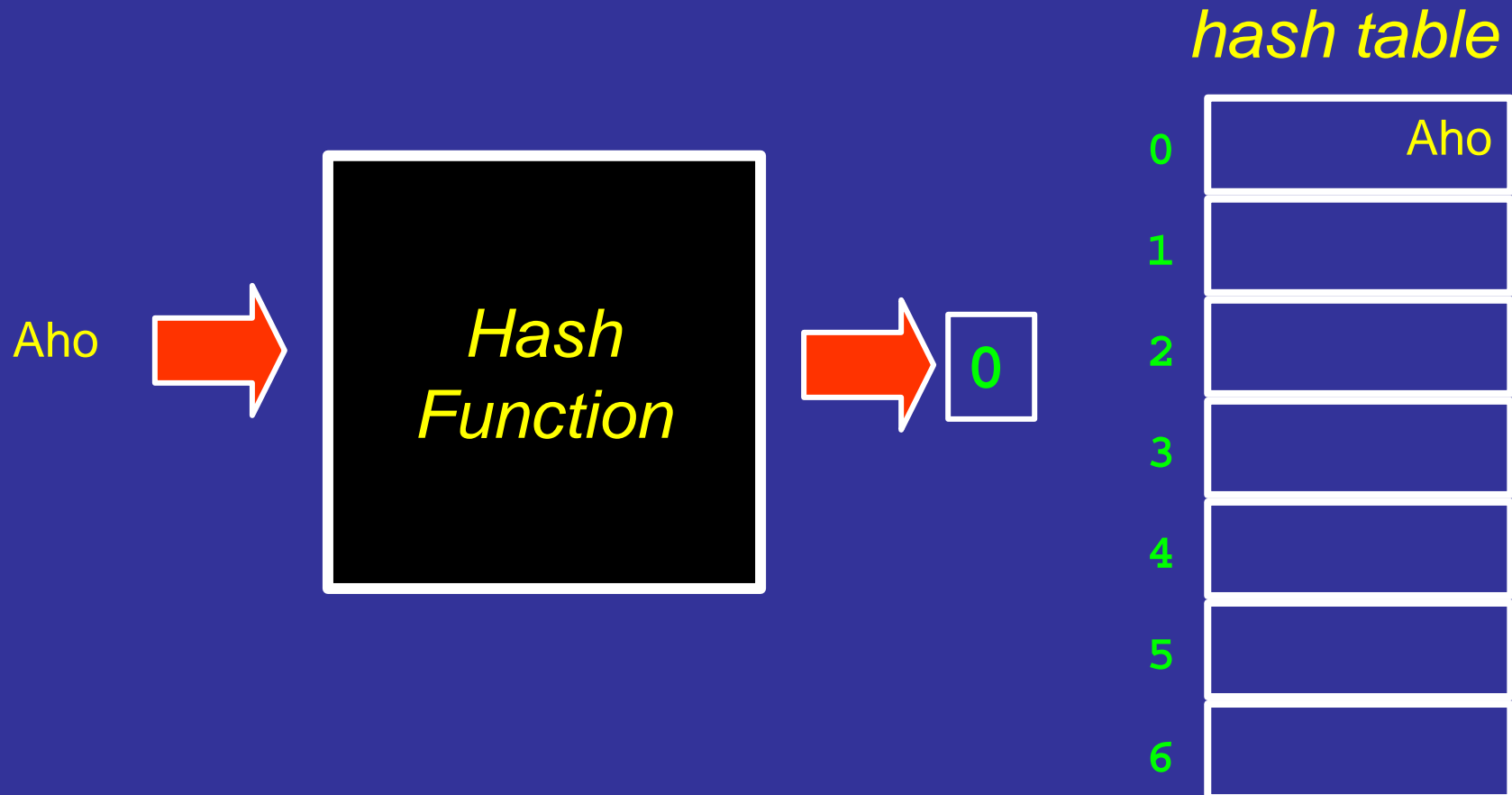
- Linear search in the array from the position where collision occurred.

Insert

- Apply hash function to get a position.
- Try to insert key at this position.
- Deal with collision.

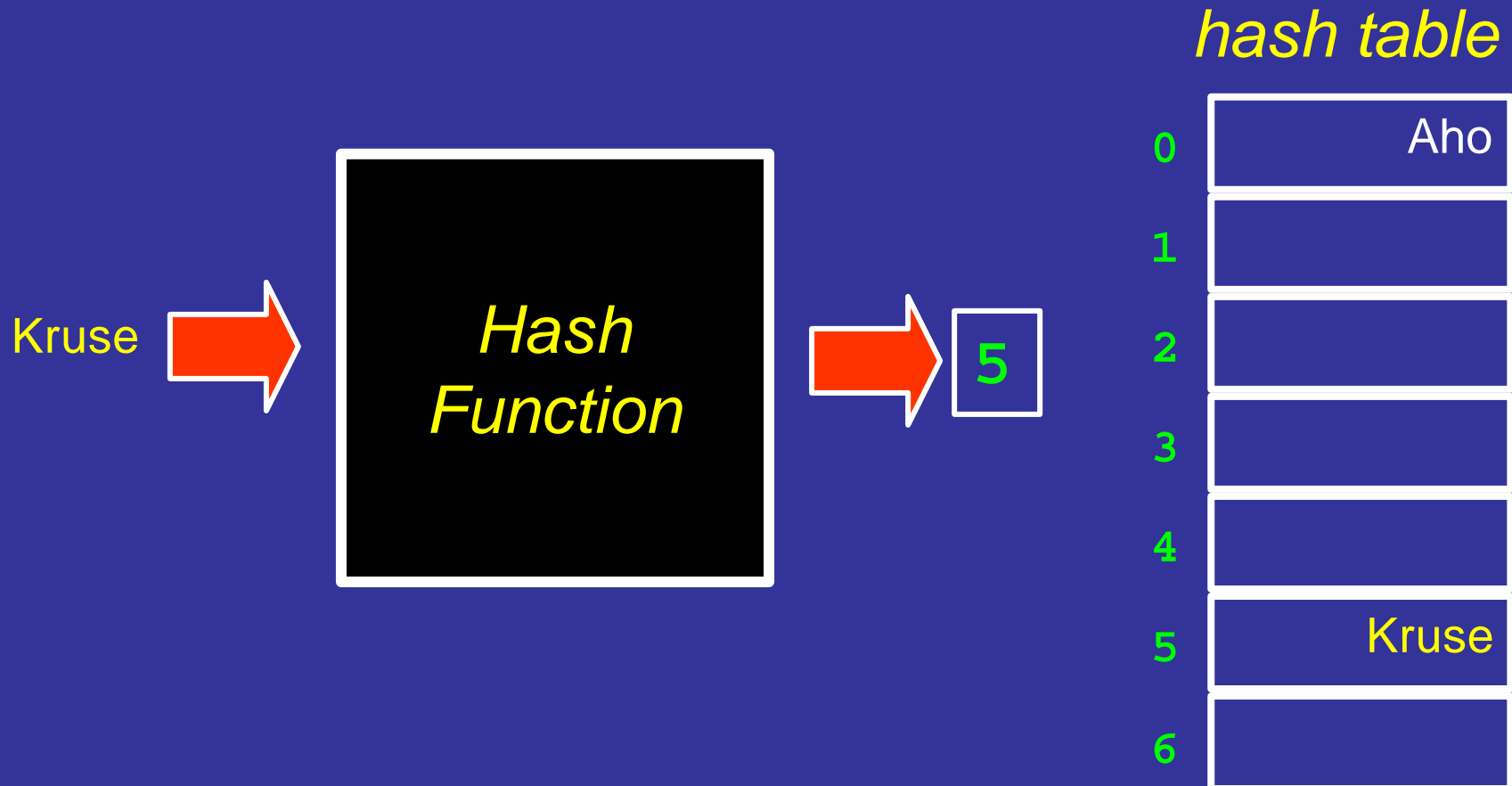
Example: Insert with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



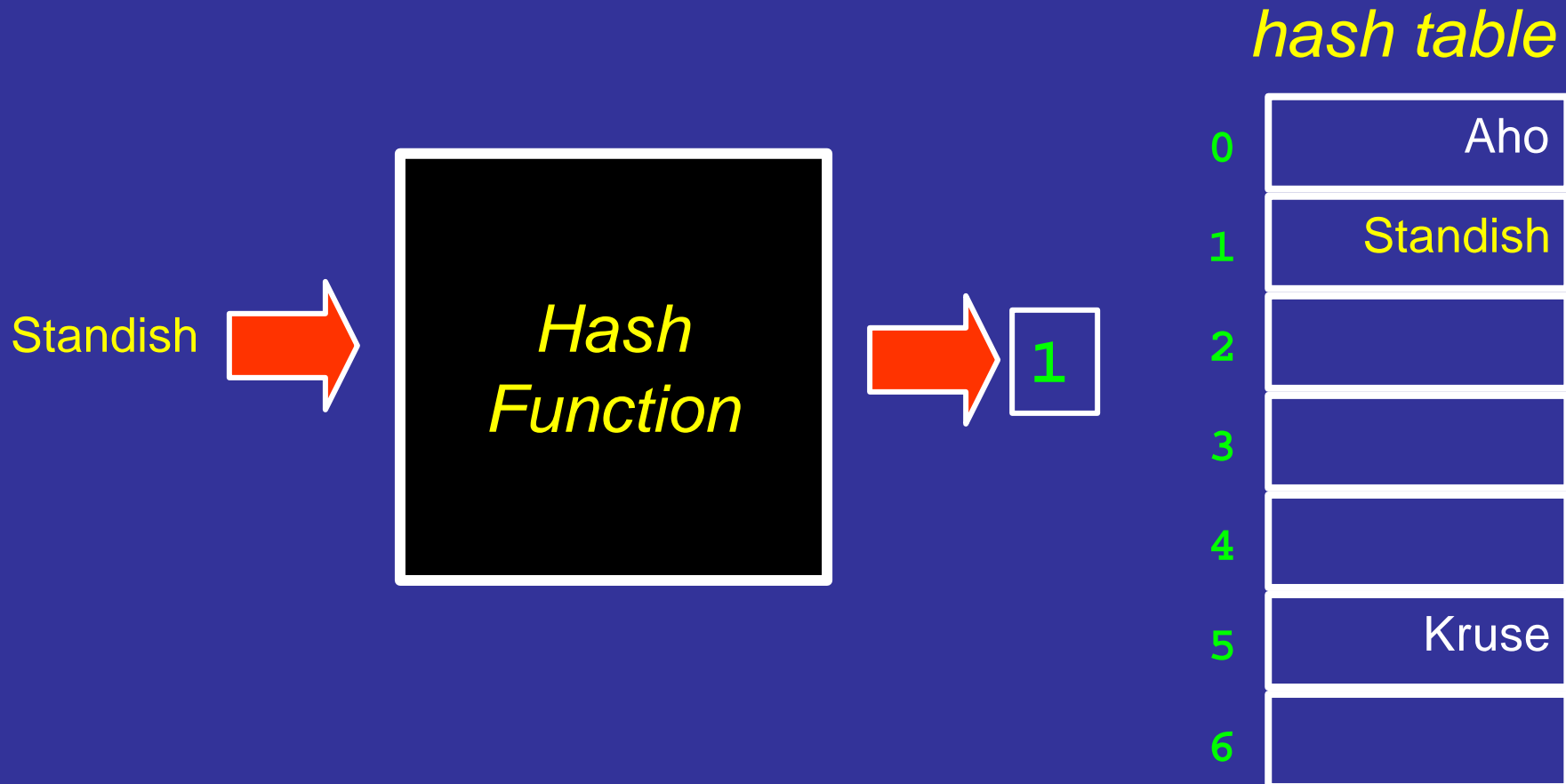
Example: Insert with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



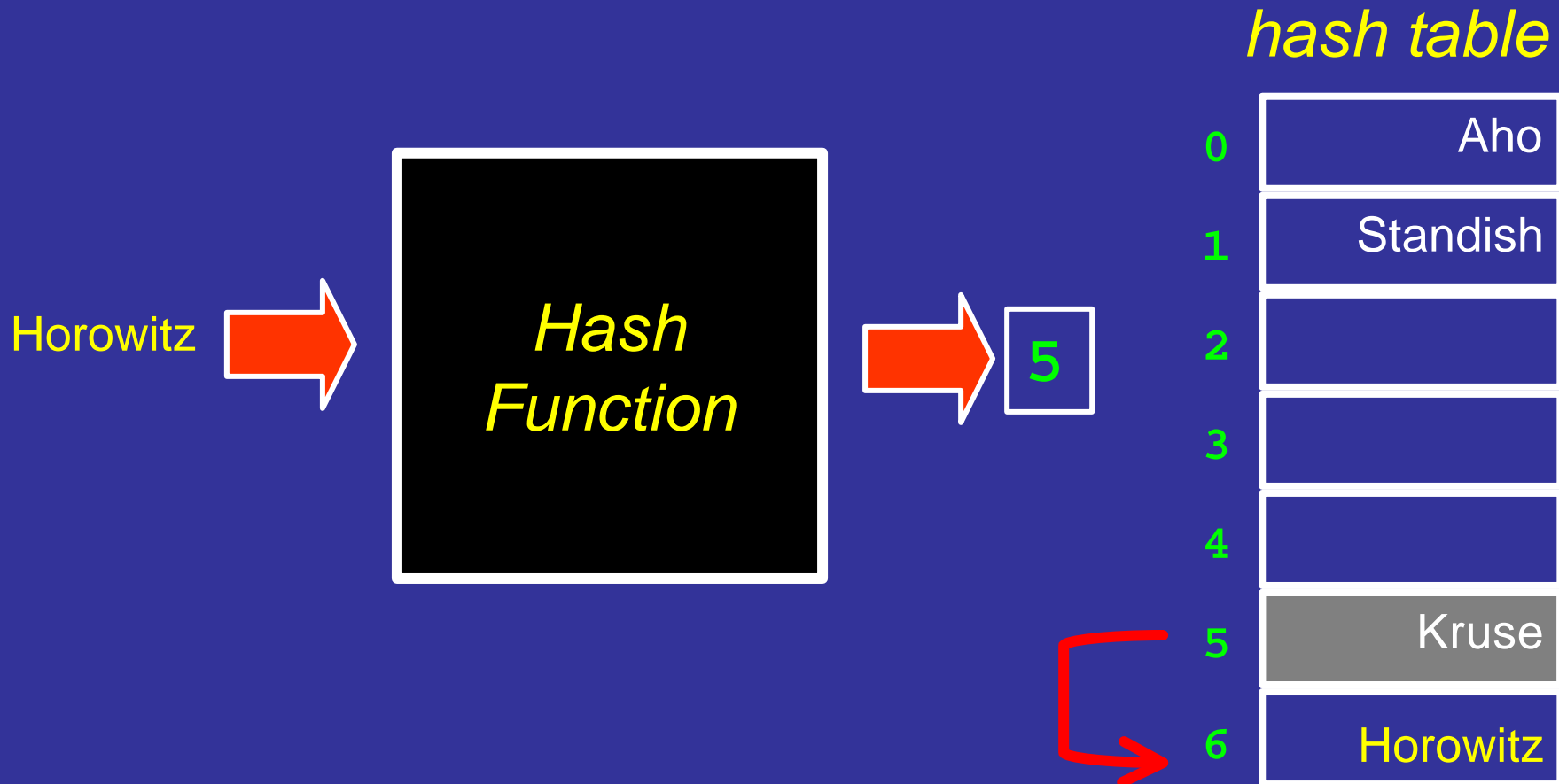
Example: Insert with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



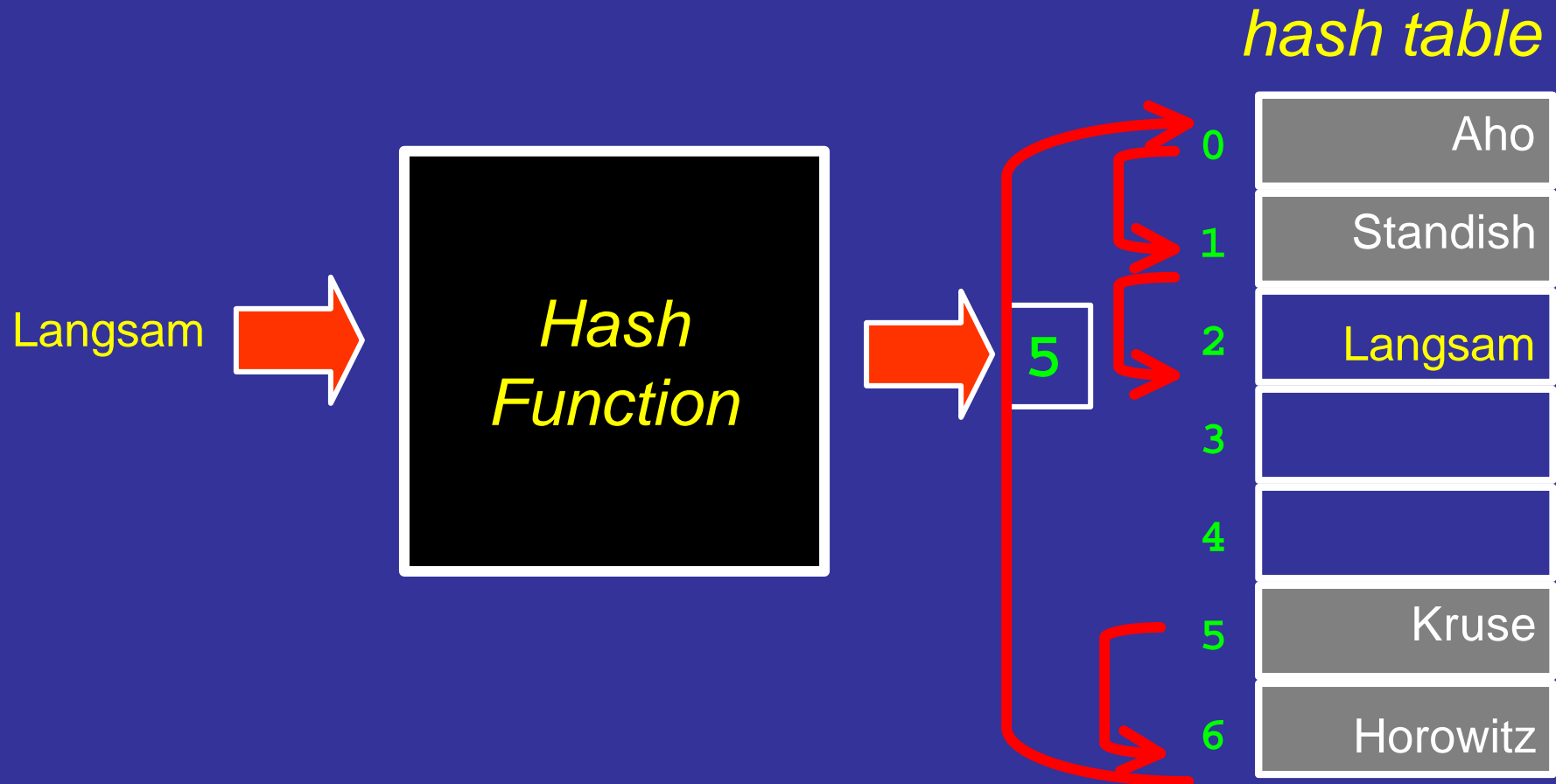
Example: Insert with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Example: Insert with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



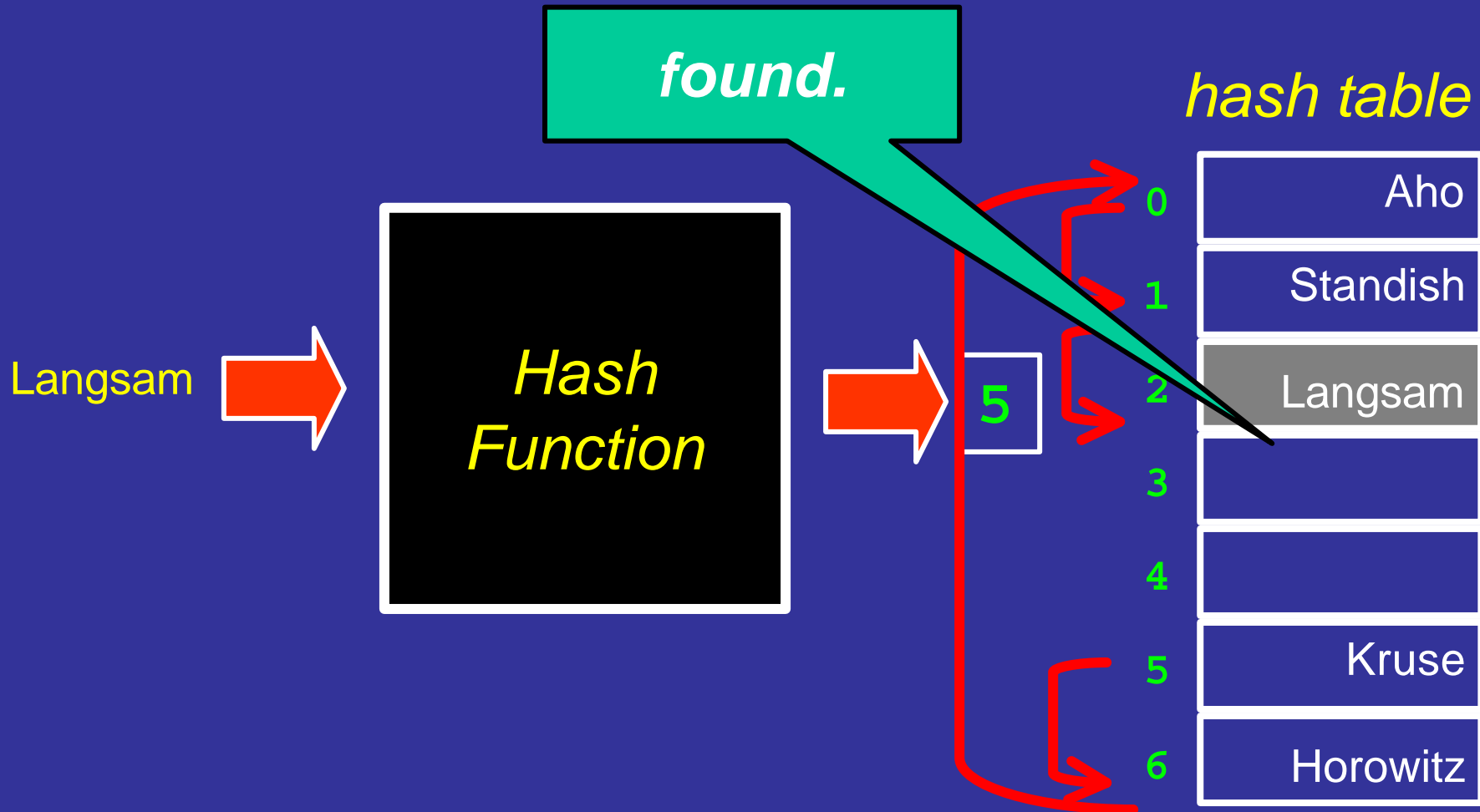
```
module linearProbe(item)
{
  position = hash(key of item)
  count = 0
  loop {
    if (count == hashTableSize) then {
      output "Table is full"
      exit loop
    }
    if (hashTable[position] is empty) then {
      hashTable[position] = item
      exit loop
    }
    position = (position + 1)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

Search

- Apply hash function to get a position.
- Look in that position.
- Deal with collision.

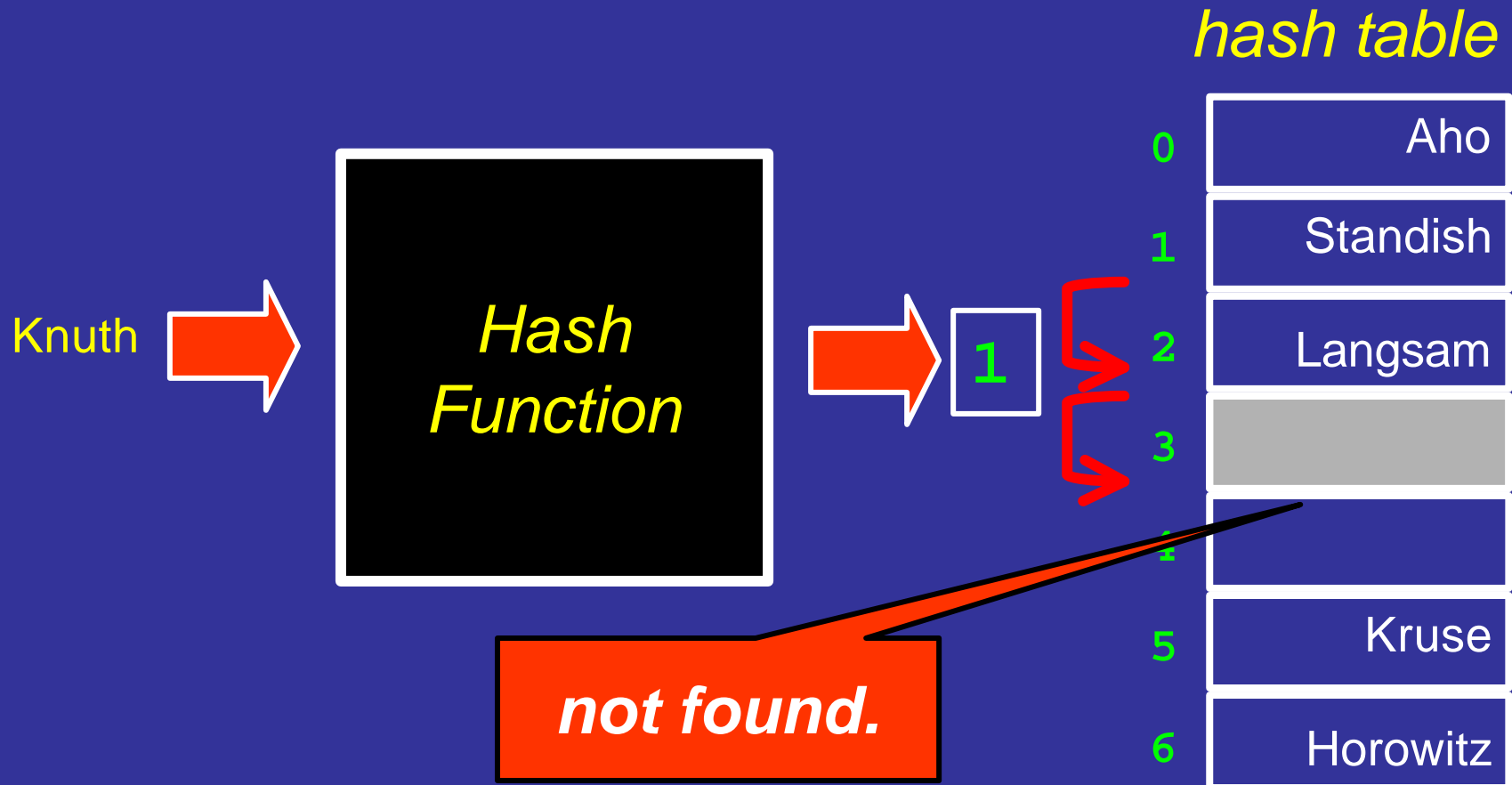
Example: Search with Linear Probing

Aho, Kruse, Standish, Horowitz, Langsam, Sedgewick, Knuth



Example: Search with Linear Probing

Aho, Kruse, Standish, Horowitz, 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

- 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.

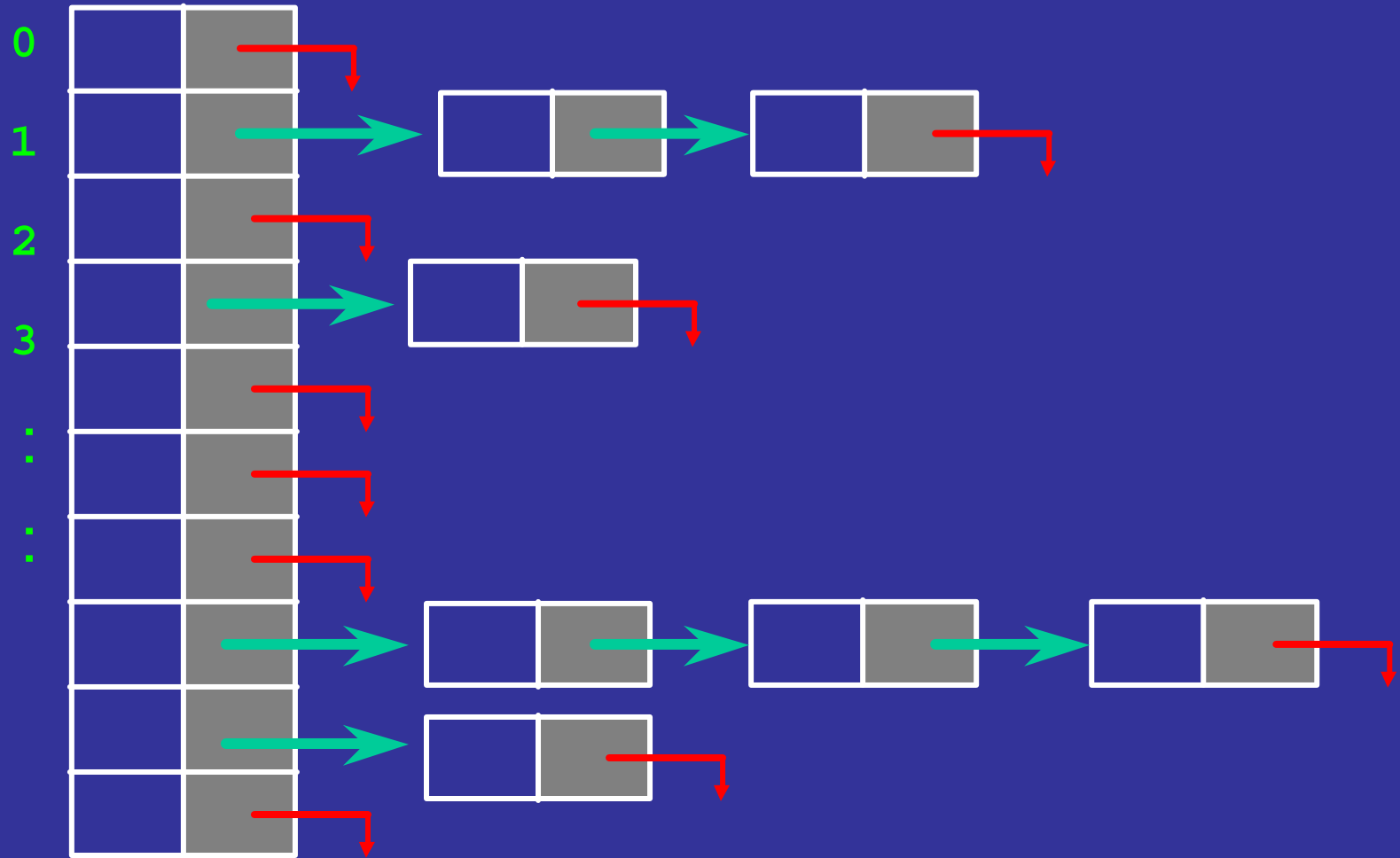
Linear Probing: Problems

- Speed.
- Tendency for **clustering** to occur as the table becomes half full.
- **Deletion** of records is very difficult.

Chaining

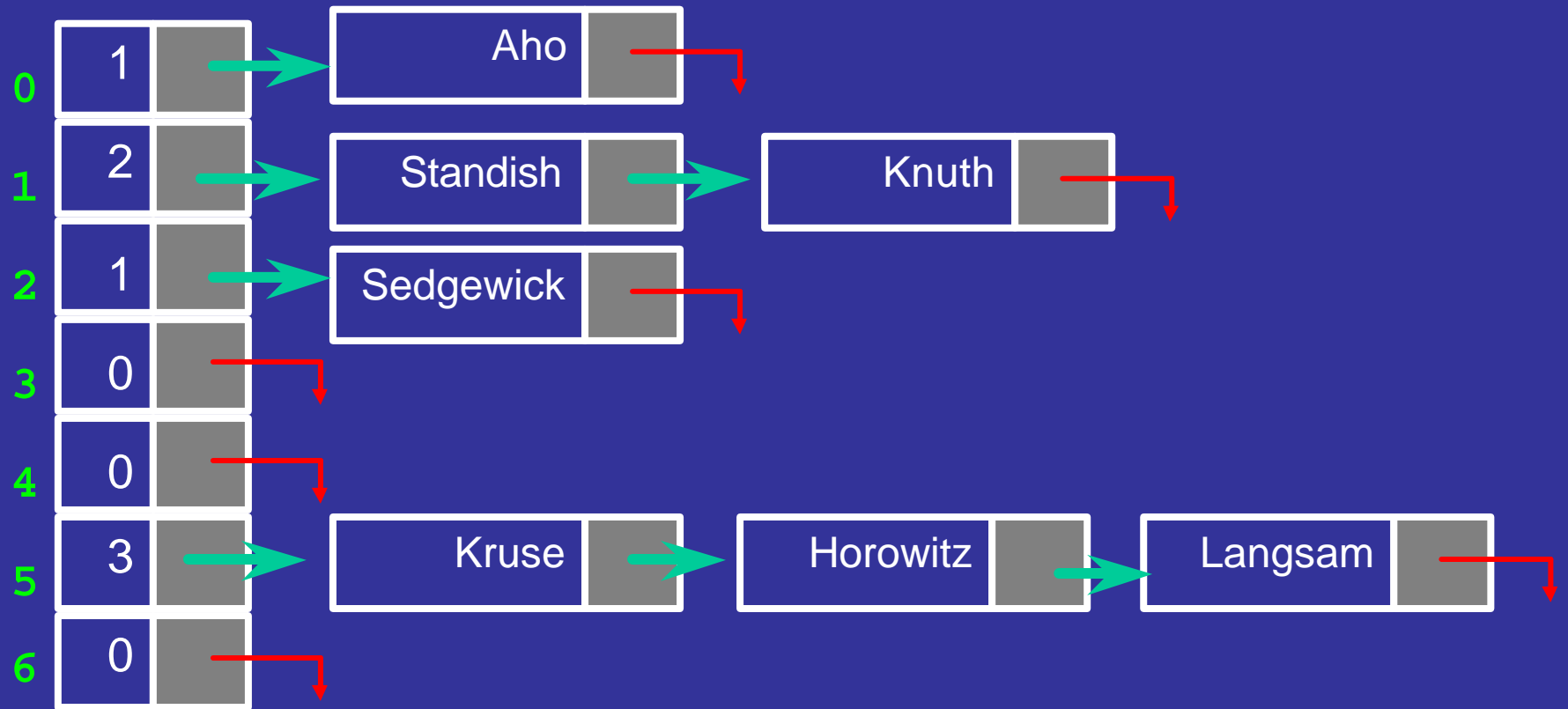
- Uses a Linked List at each position in the Hash Table.

hash table



Example: Chaining

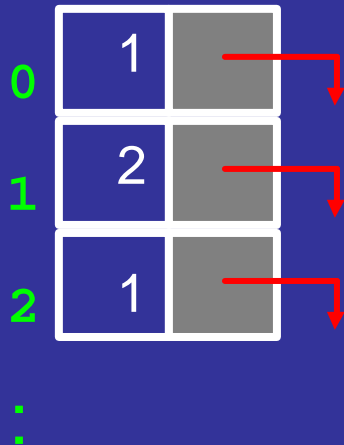
Aha, Kruse, Standish, Horowitz, Langsam, Sedgwick, Knuth
0, 5, 1, 0, 5, 5, 2, 1



Hashtable with Chaining

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

List hashTable[MAXTABLE];

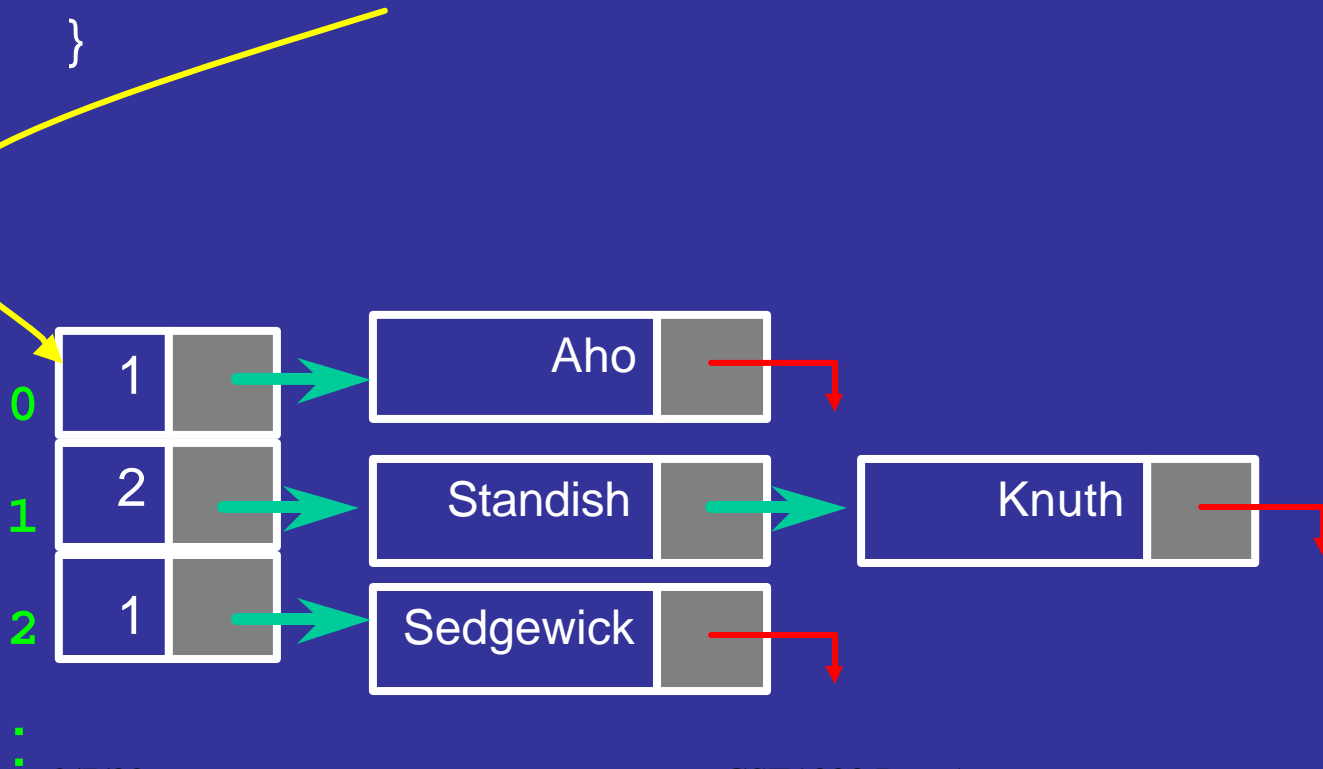


Insert with Chaining

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

```
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)
```

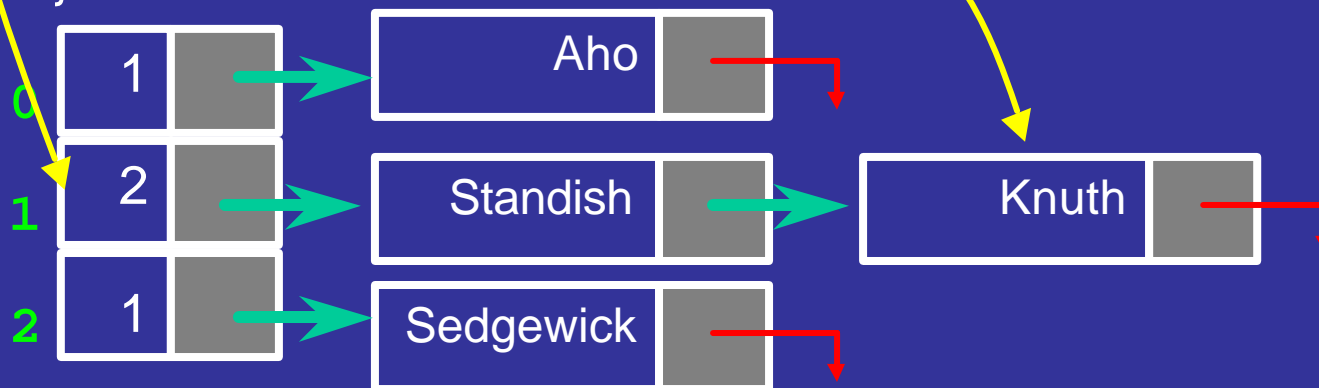
```
{
```

```
  posHash = hash(key of item)
  Node* found;
```

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

```
  return found;
```

```
}
```



⋮

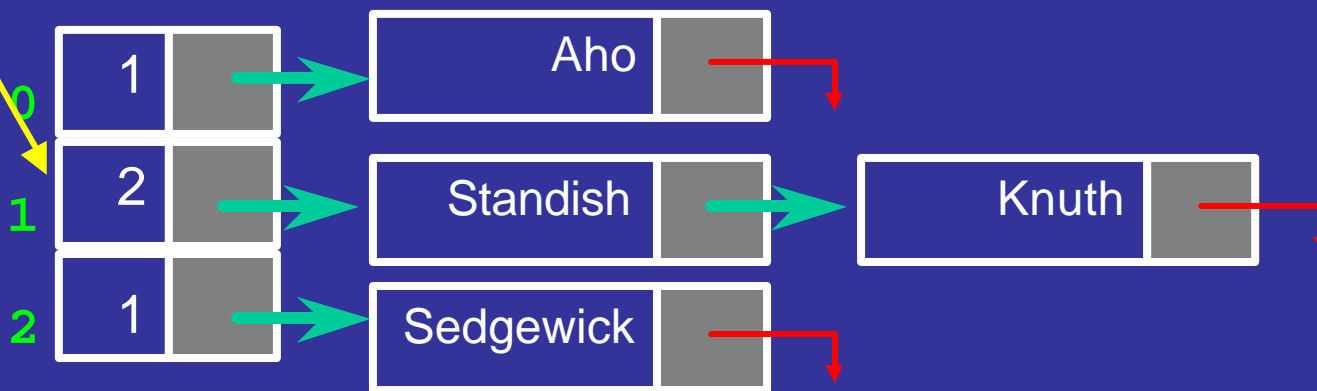
Delete with Chaining

- 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.
 - A linked list at every element in the array

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.

Revision

- Hash Tables
 - Hash Functions
 - Insert, Search
- Collision Resolution
 - Linear Probing, Chaining

Preparation

- Read Chapter 7.6-7.8 in Kruse et al.