Hash Tables

Hash Tables

- Hash tables combine the random access ability of an array with the dynamism of a linked list.

- This means (assuming we define our hash table well):
  - Insertion can start to tend toward $\theta(1)$
  - Deletion can start to tend toward $\theta(1)$
  - Lookup can start to tend toward $\theta(1)$

- We're gaining the advantages of both types of data structure, while mitigating the disadvantages.

Hash Tables

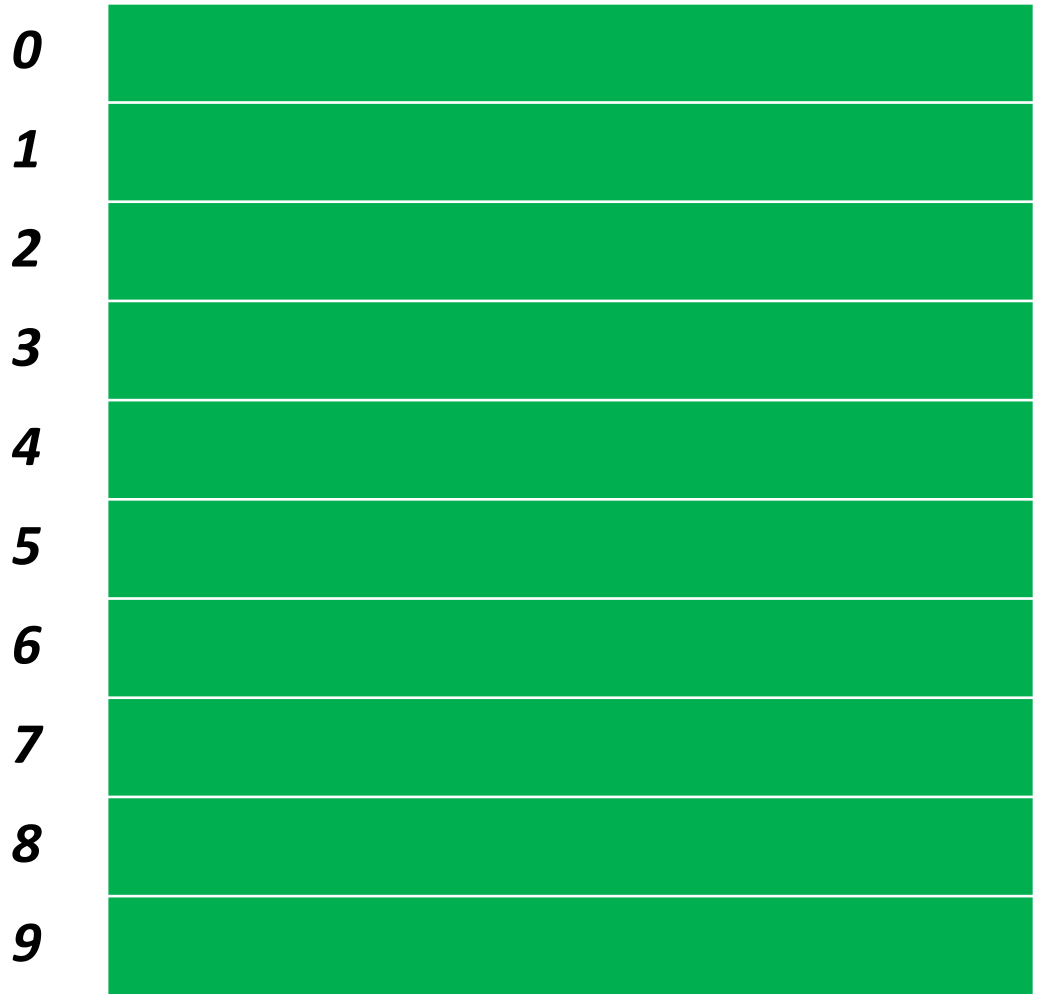
- To get this performance upgrade, we create a new structure whereby when we insert data into the structure, the data itself gives us a clue about where we will find the data, should we need to later look it up.

- The trade off is that hash tables are not great at ordering or sorting data, but if we don’t care about that, then we’re good to go!

Hash Tables

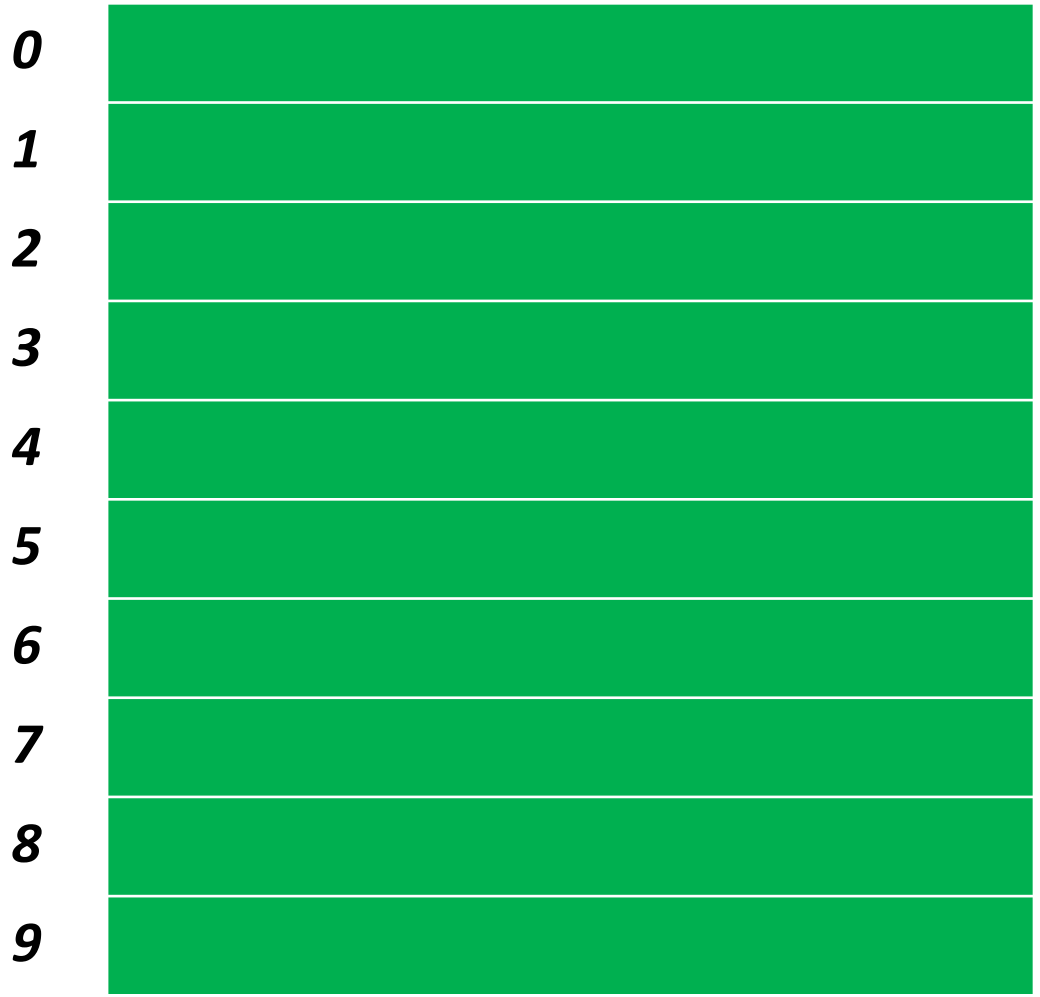
- A hash table amounts to a combination of two things with which we're quite familiar.
  - First, a **hash function**, which returns an nonnegative integer value called a *hash code*.
  - Second, an **array** capable of storing data of the type we wish to place into the data structure.
- The idea is that we run our data through the hash function, and then store the data in the element of the array represented by the returned hash code.

Hash Tables



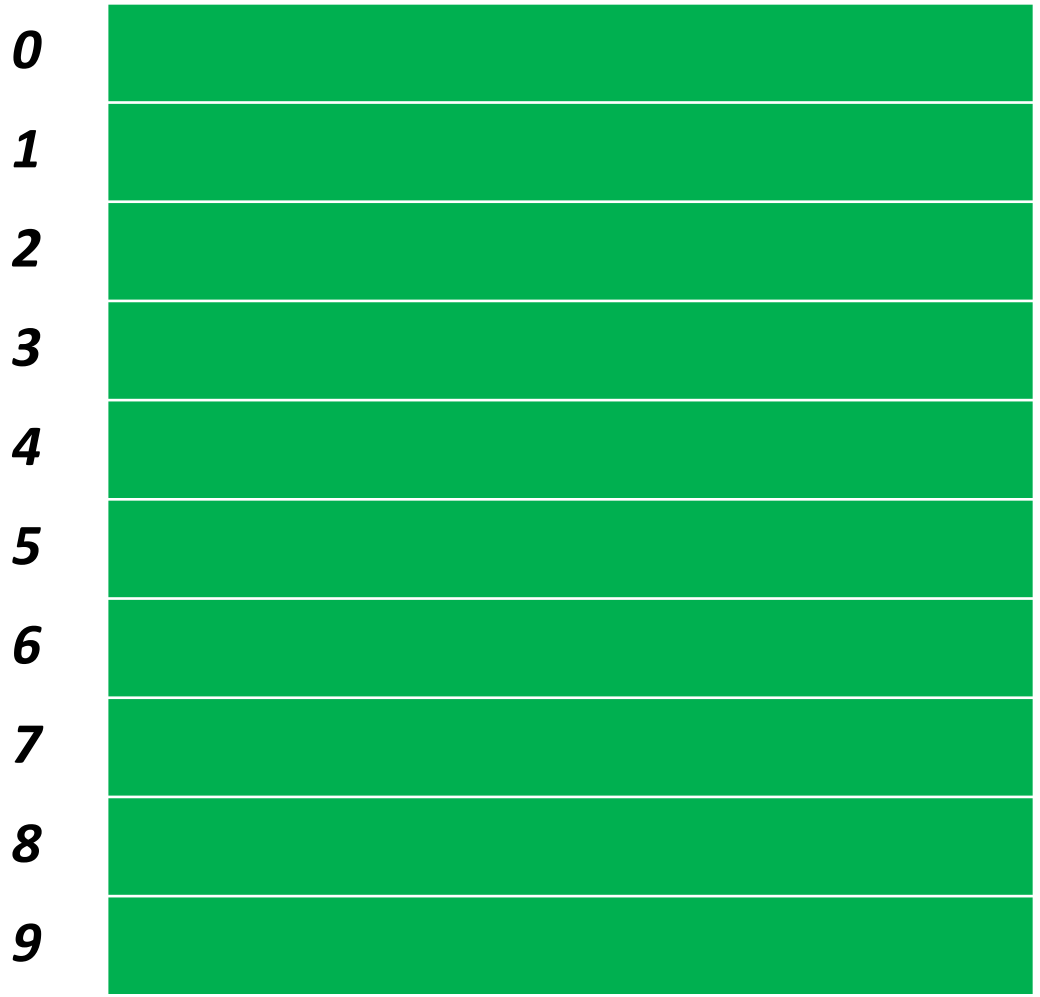

```
string hashtable[10];
```

Hash Tables



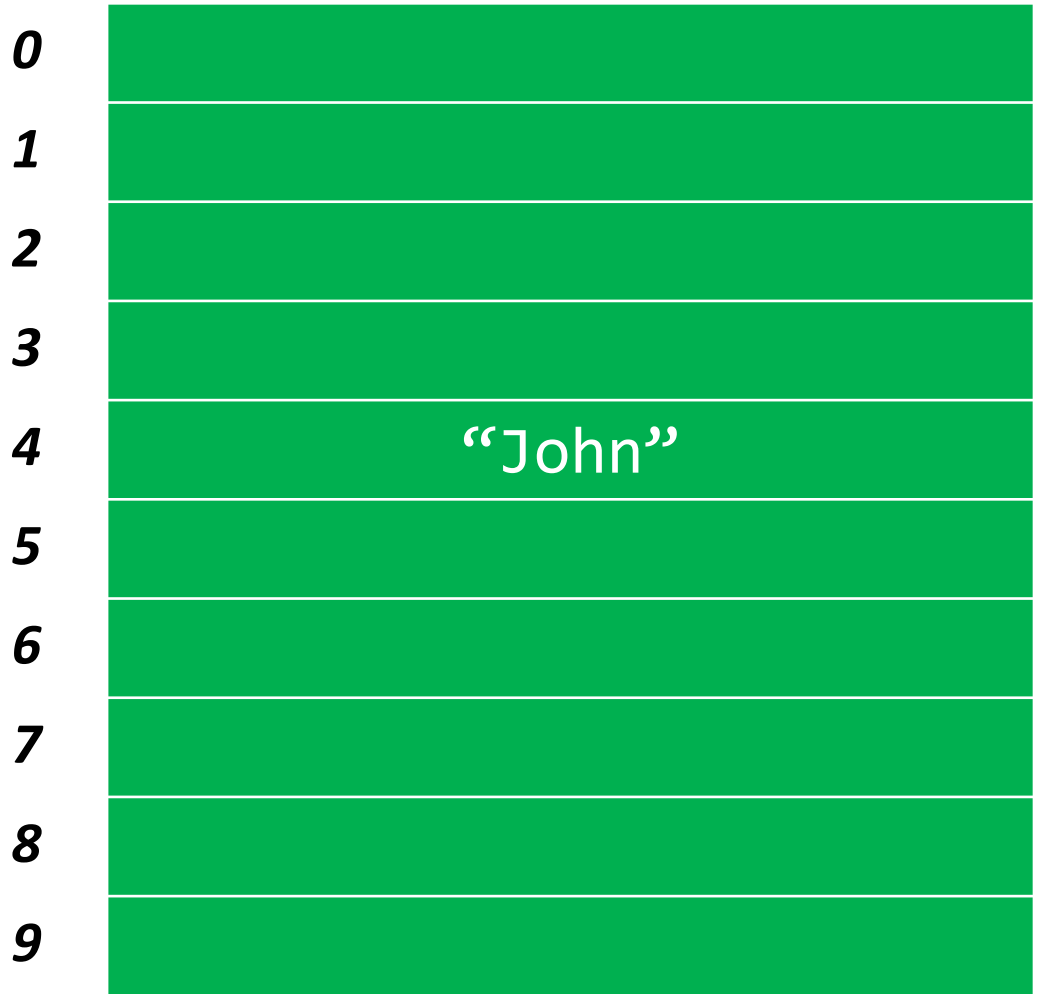

```
int x = hash(“John”);
```

Hash Tables



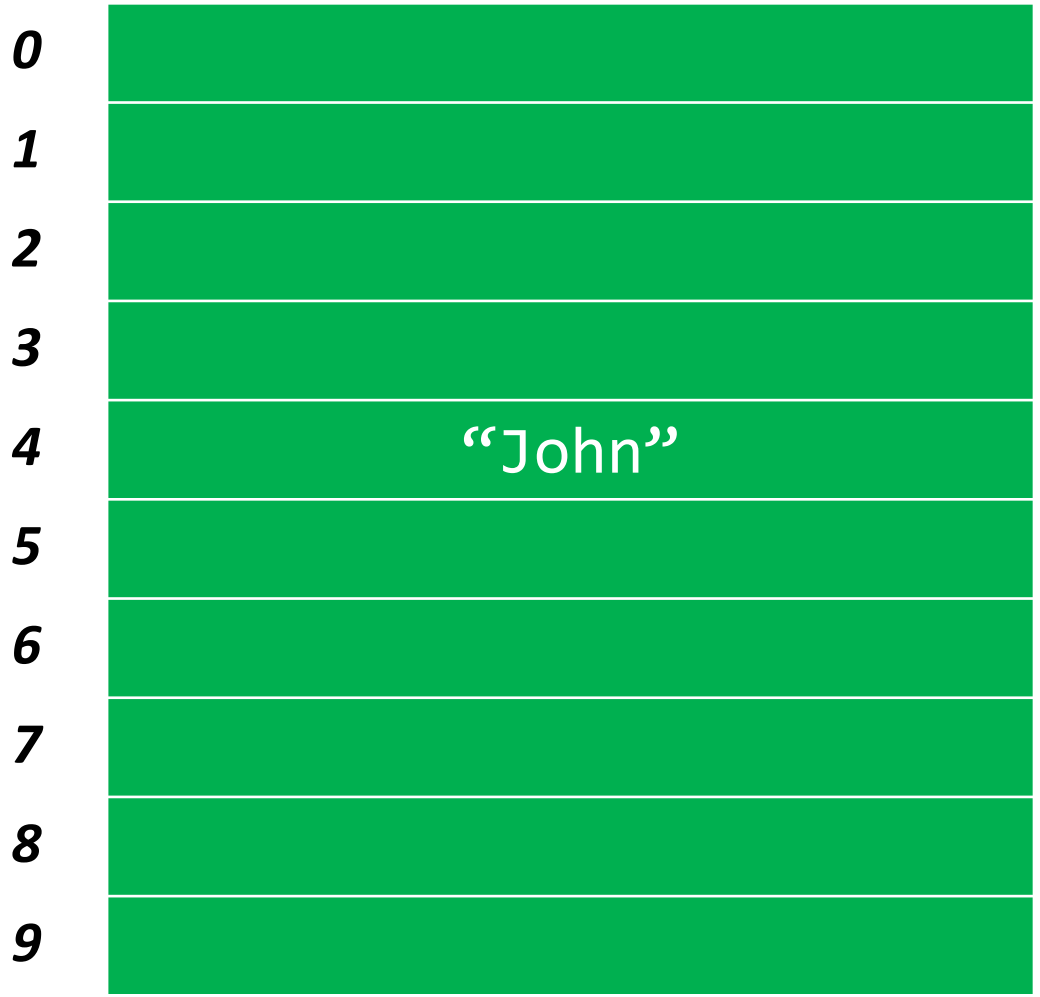

```
int x = hash(“John”);

// x is now 4
```

Hash Tables



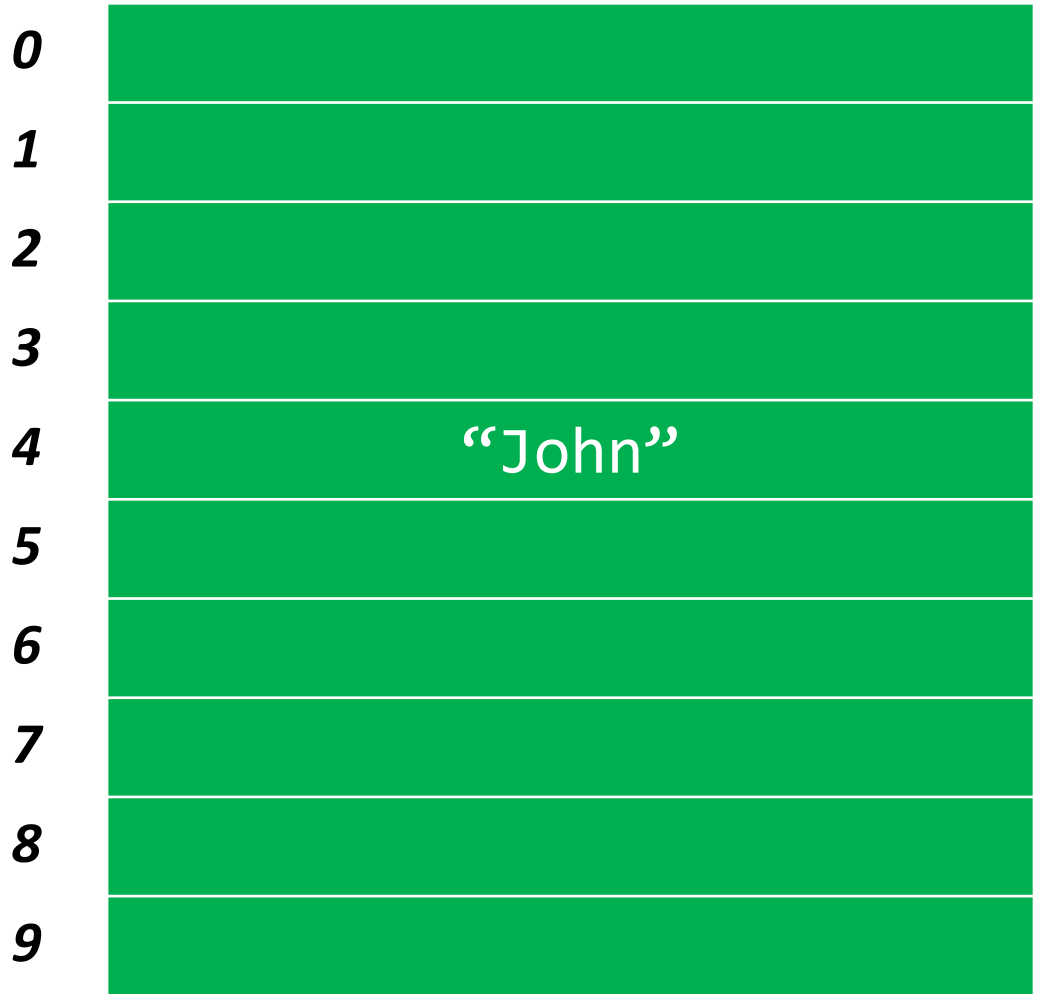

```
int x = hash(“John”);

// x is now 4

hashtable[x] = “John”;
```

Hash Tables



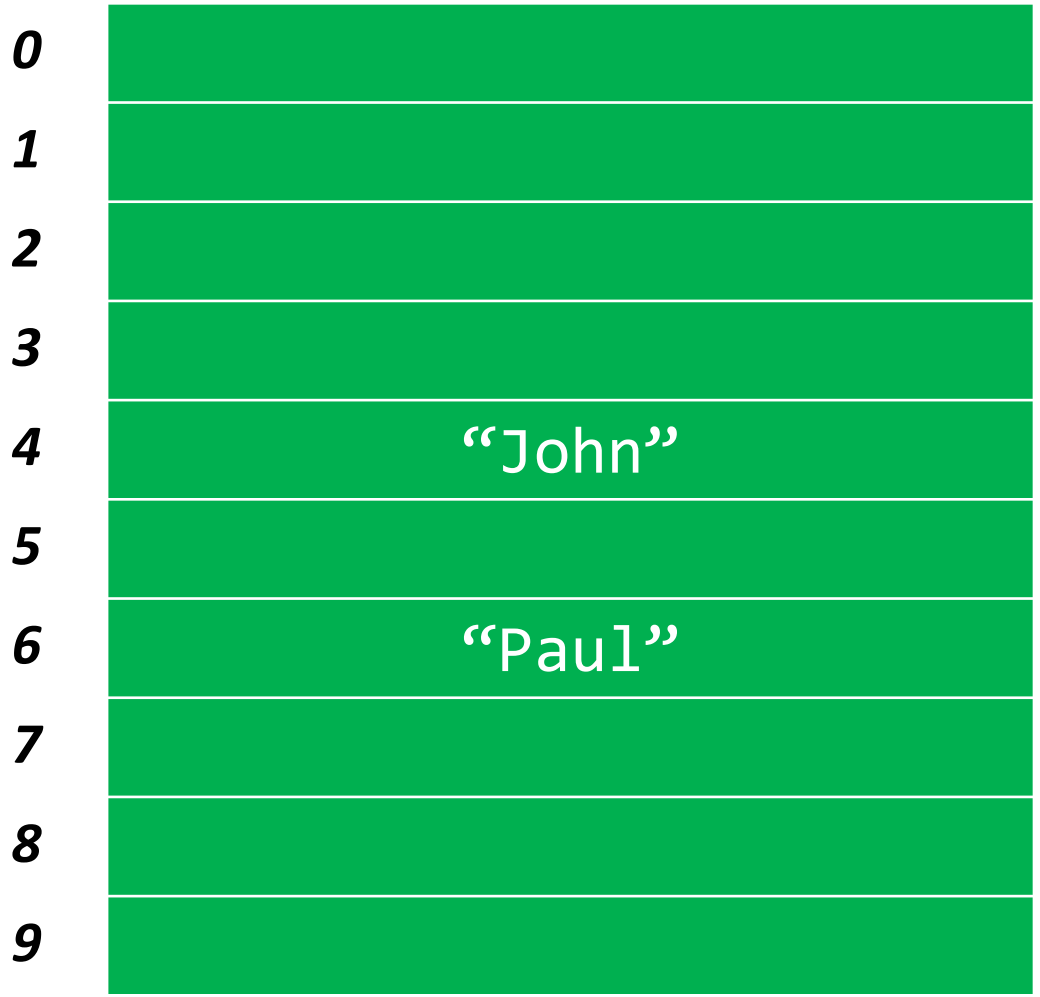

```
int y = hash("Paul");
```

Hash Tables




```
int y = hash(“Paul”);

// y is now 6
```

Hash Tables




```
int y = hash(“Paul”);

// y is now 6

hashtable[y] = “Paul”;
```

Hash Tables

- How to define a hash function? Really no limit to the number of possible hash functions.

- A good hash function should:
  - Use only the data being hashed
  - Use all of the data being hashed
  - Be deterministic
  - Uniformly distribute data
  - Generate very different hash codes for very similar data

Hash Tables

```
unsigned int hash(char* str)
{
    int sum = 0;
    for (int j = 0; str[j] != '\0'; j++)
    {
        sum += str[j];
    }
    return sum % HASH_MAX;
}
```

Hash Tables

```
unsigned int hash(char* str)
{
    int sum = 0;
    for (int j = 0; str[j] != '\0'; j++)
    {
        sum += str[j];
    }
    return sum % HASH_MAX;
}
```

Hash Tables

```
unsigned int hash(char* str)
{
    int sum = 0;
    for (int j = 0; str[j] != '\0'; j++)
    {
        sum += str[j];
    }
    return sum % HASH_MAX;
}
```

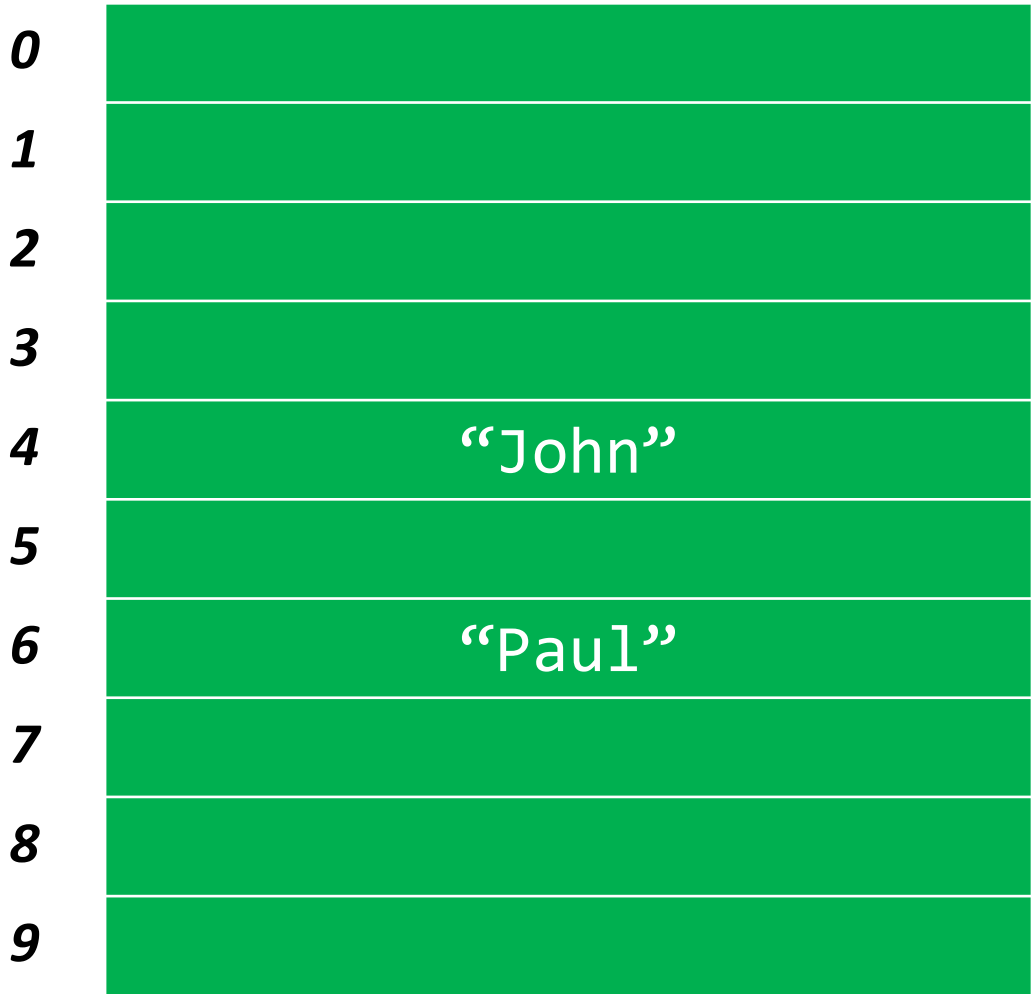
Hash Tables

```
unsigned int hash(char* str)
{
    int sum = 0;
    for (int j = 0; str[j] != '\0'; j++)
    {
        sum += str[j];
    }
    return sum % HASH_MAX;
}
```

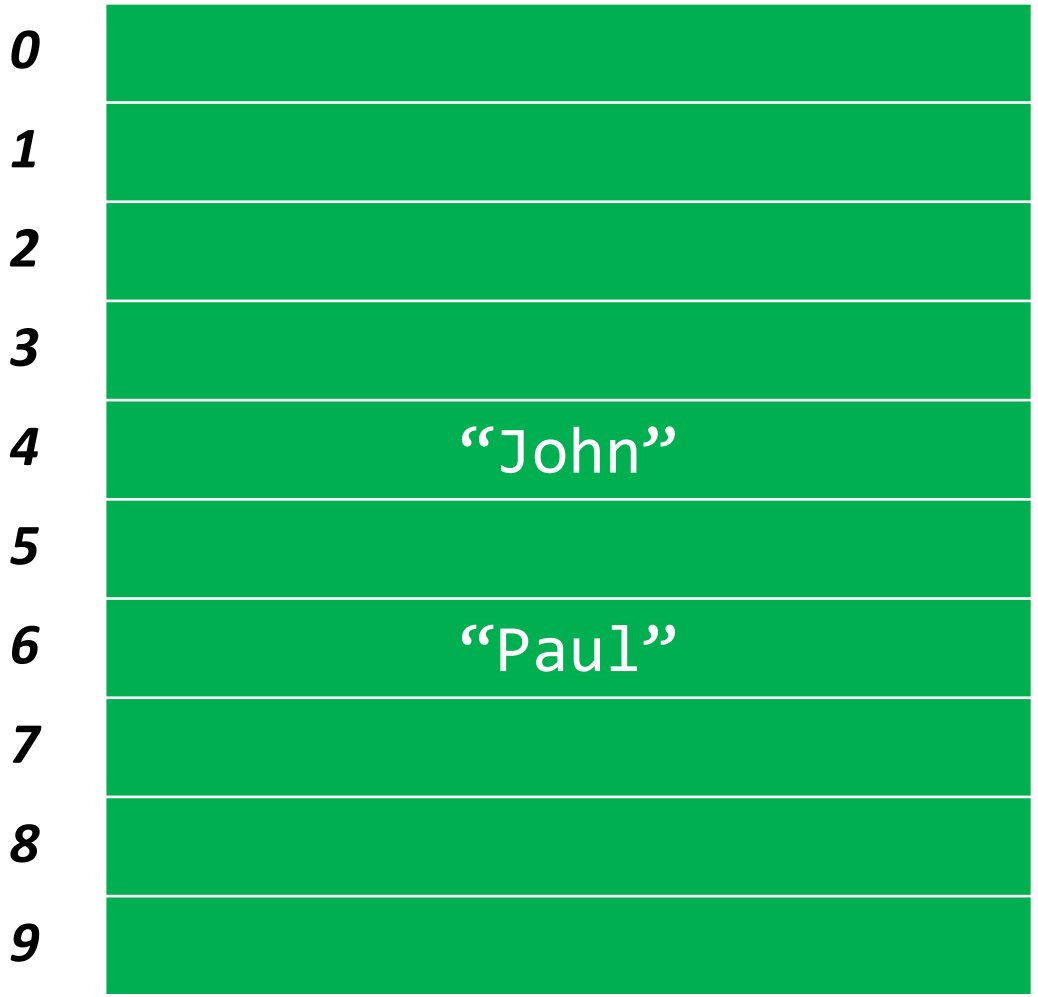
Hash Tables

```
unsigned int hash(char* str)
{
    int sum = 0;
    for (int j = 0; str[j] != '\0'; j++)
    {
        sum += str[j];
    }
    return sum % HASH_MAX;
}
```

Hash Tables



Hash Tables




```
int z = hash(“Ringo”);

// z is now 6
```

Hash Tables

- A **collision** occurs when two pieces of data, when run through the hash function, yield the same hash code.

- Presumably we want to store *both* pieces of data in the hash table, so we shouldn't simply overwrite the data that happened to be placed in there first.

- We need to find a way to get both elements into the hash table while trying to preserve quick insertion and lookup.

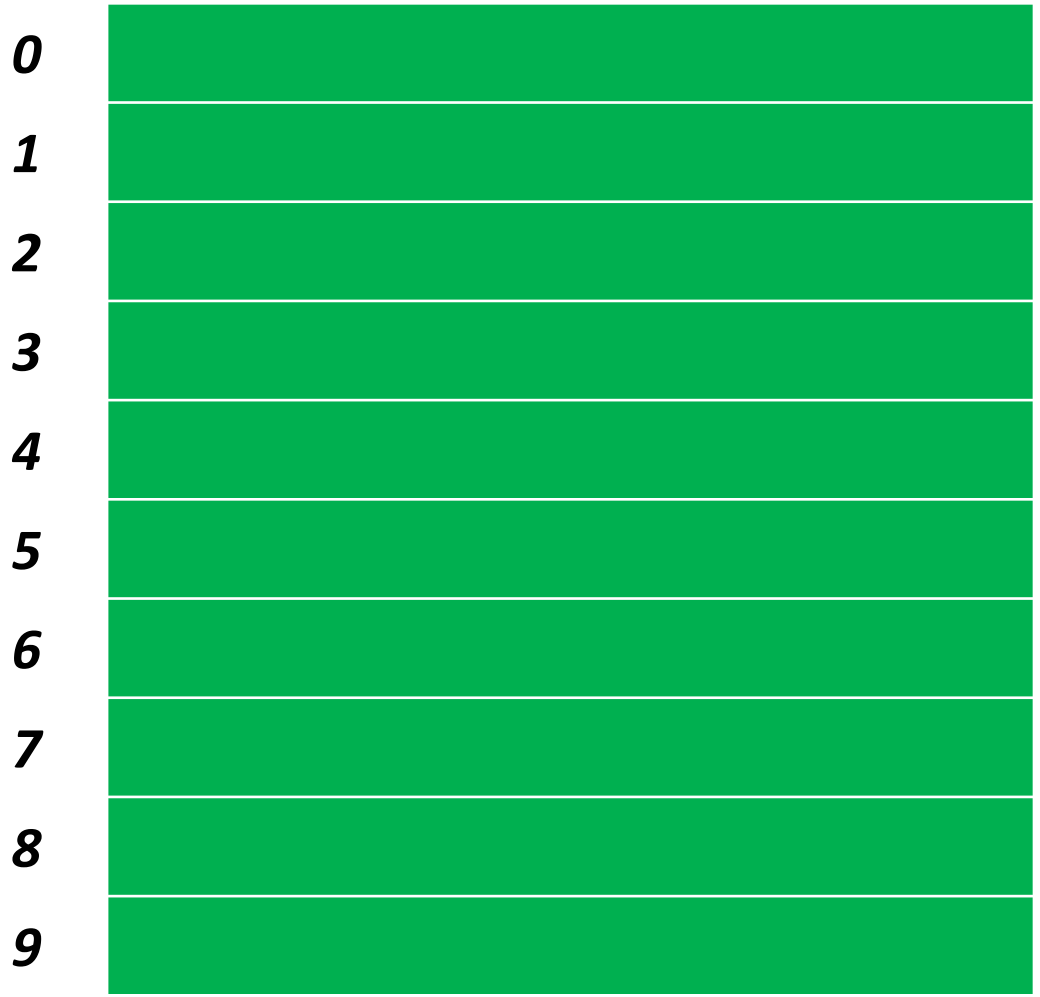
Hash Tables

- Resolving collisions: *Linear probing*

- In this method, if we have a collision, we try to place the data in the next consecutive element in the array (wrapping around to the beginning if necessary) until we find a vacancy.

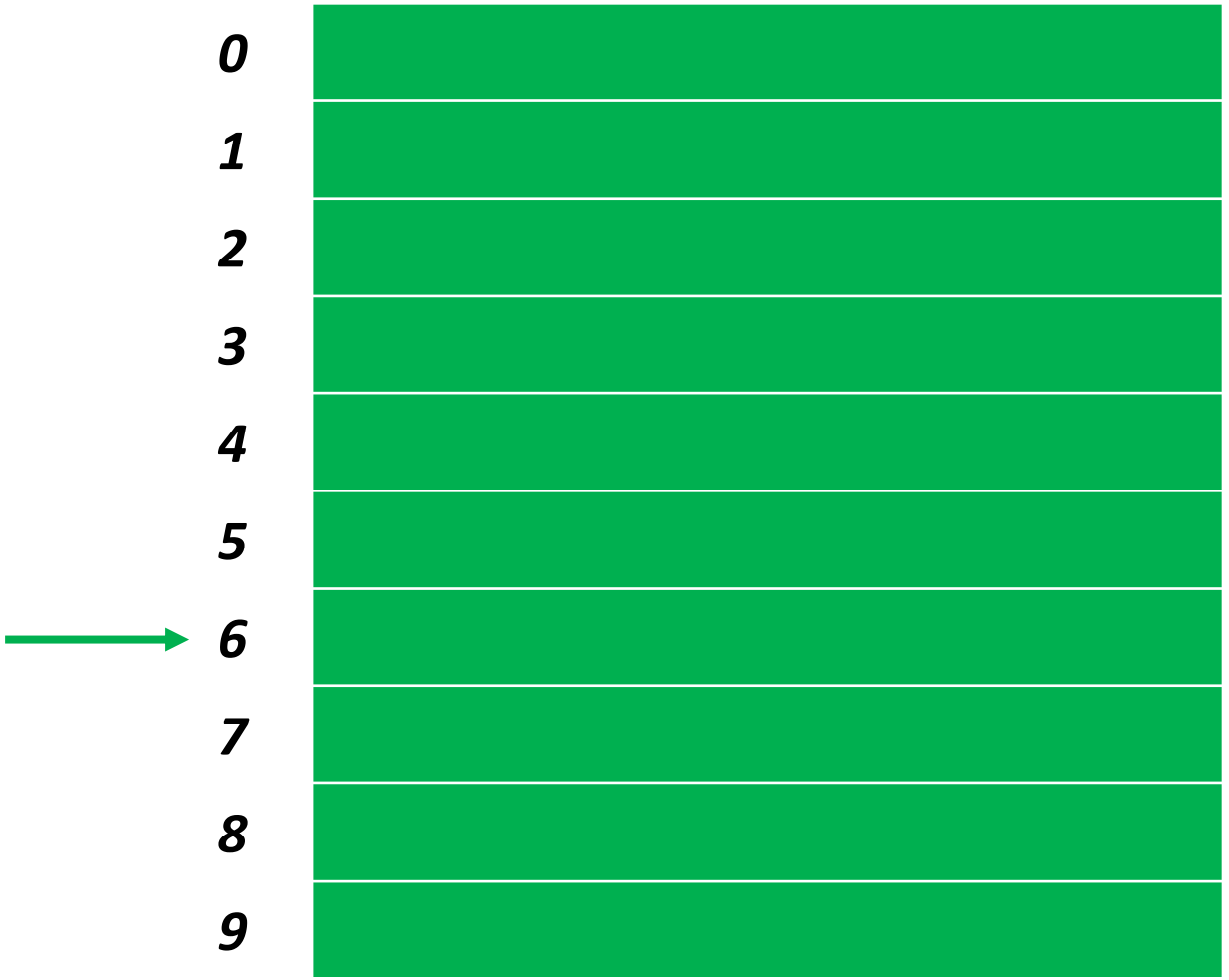
- That way, if we don’t find what we’re looking for in the first location, at least hopefully the element is somewhere nearby.

Hash Tables



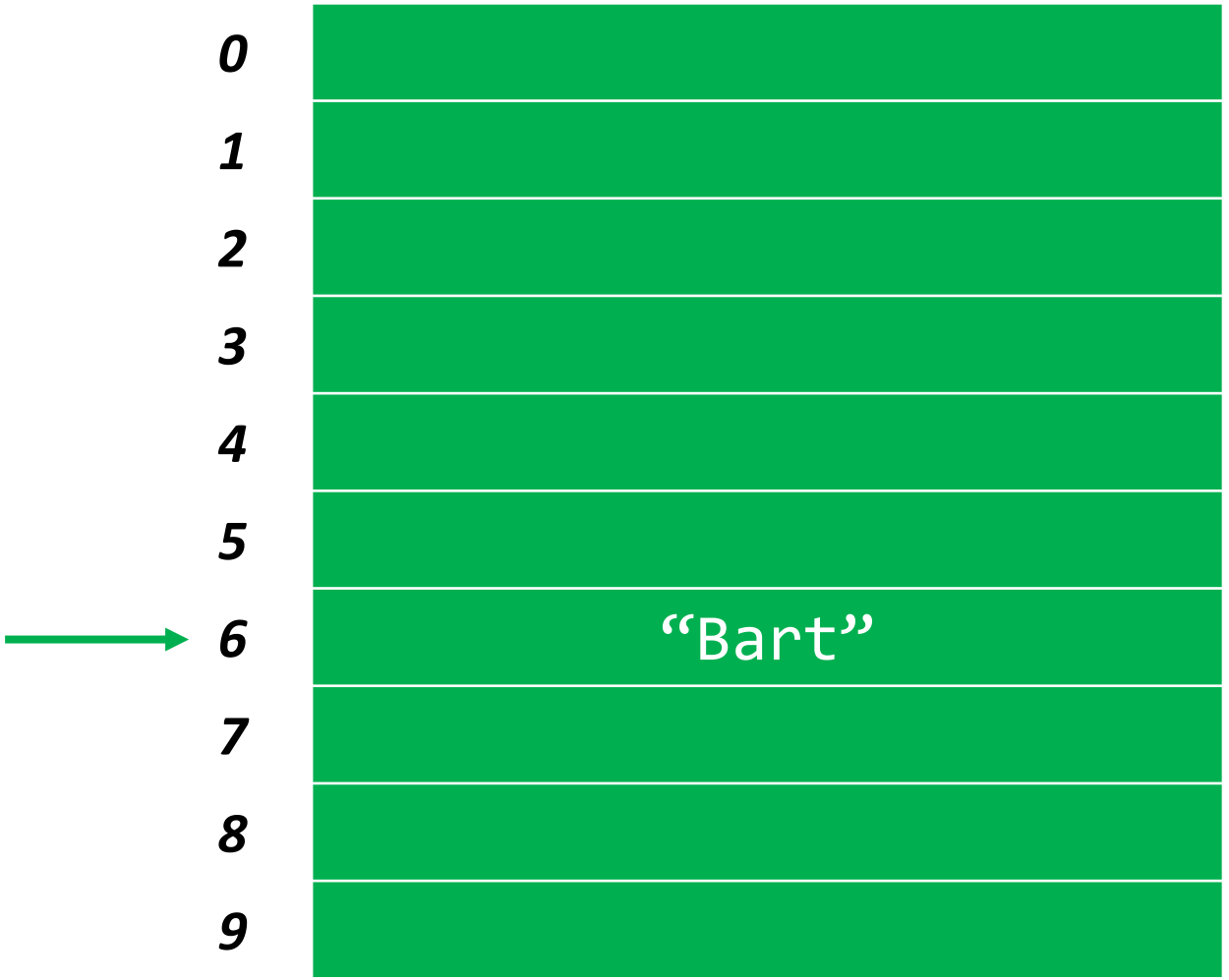
`hash("Bart");` returns **6**

Hash Tables



`hash("Bart");` returns **6**

Hash Tables



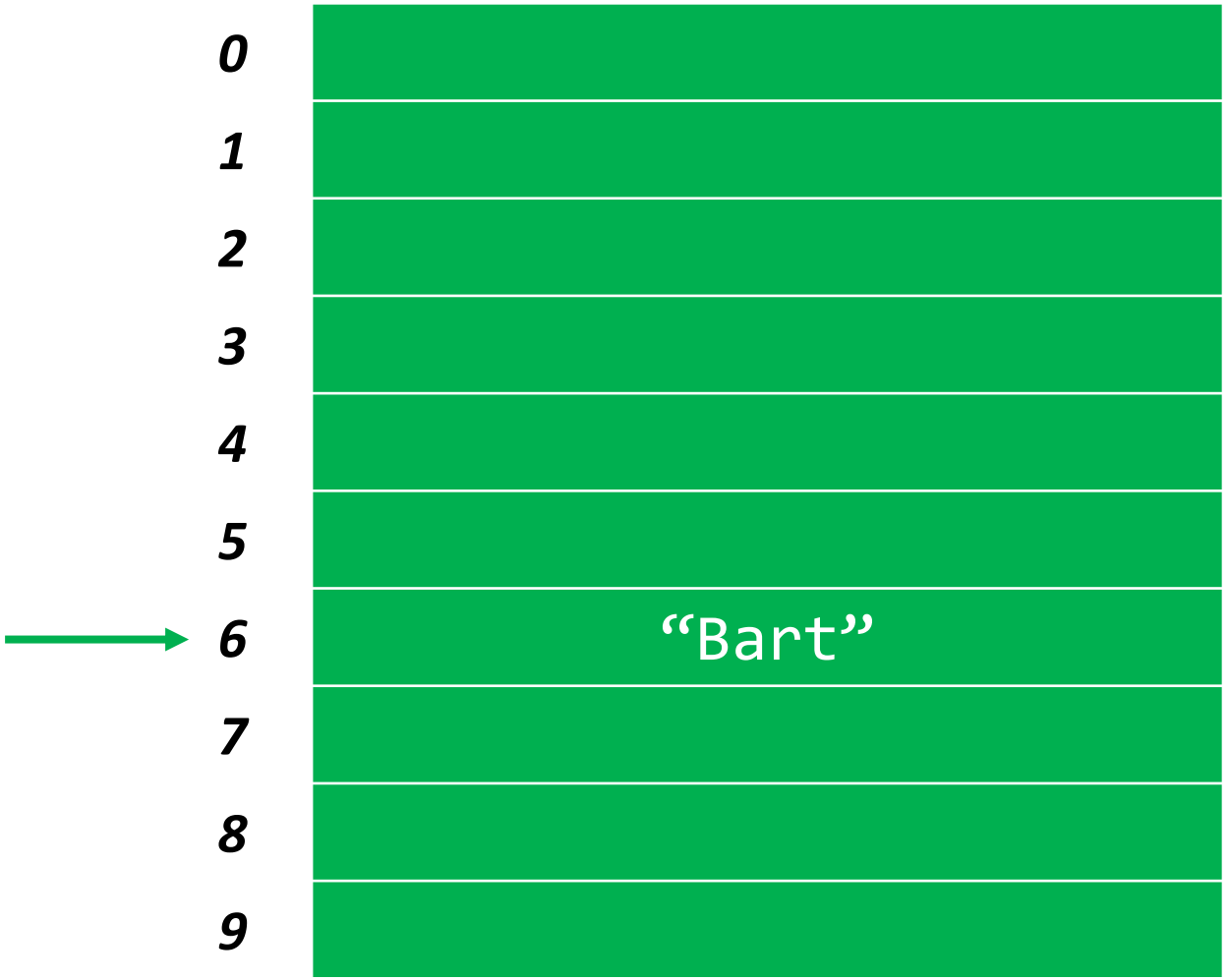
`hash(“Bart”);` returns **6**

Hash Tables

`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

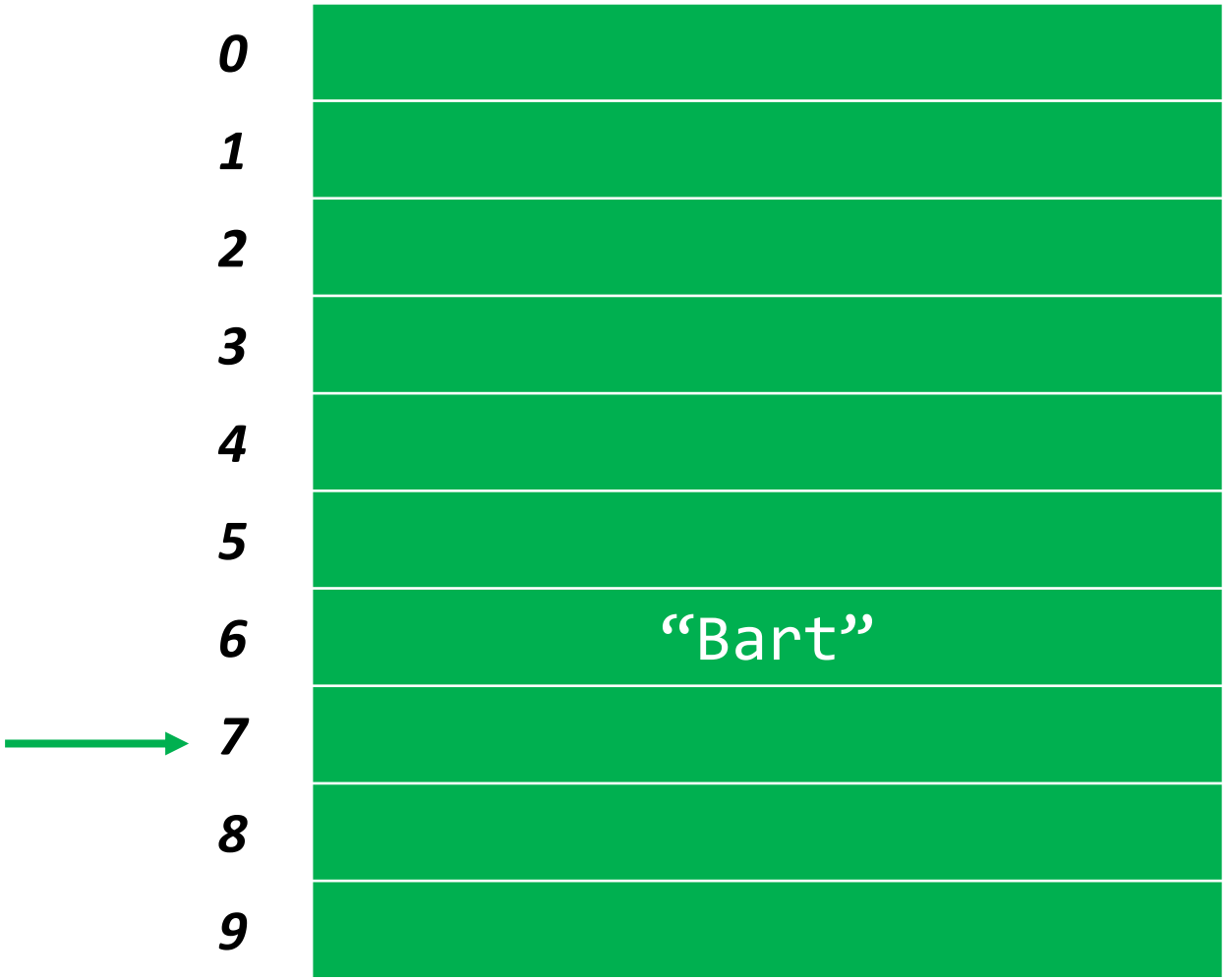
Hash Tables



`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

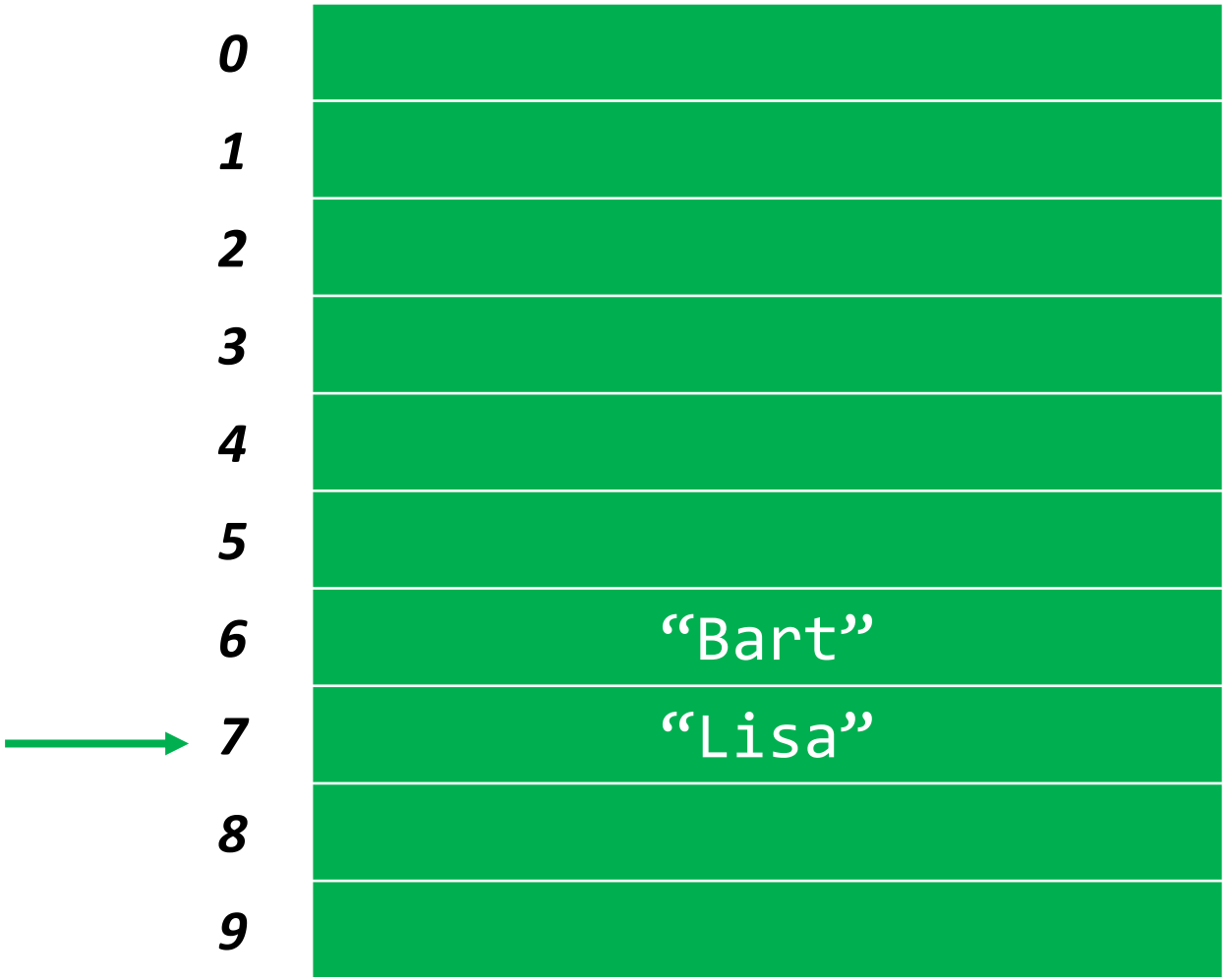
Hash Tables



`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

Hash Tables



`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

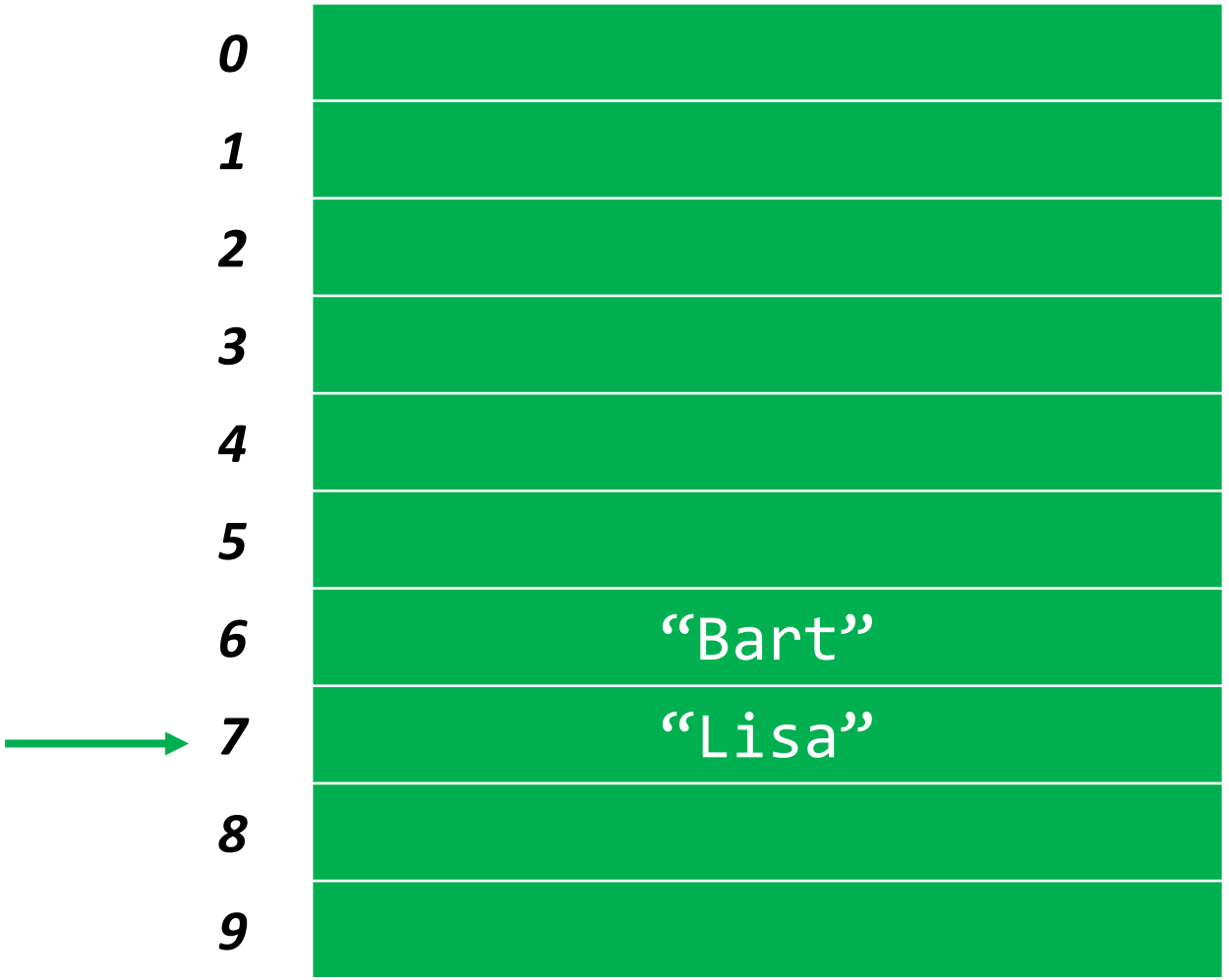
Hash Tables

`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

Hash Tables

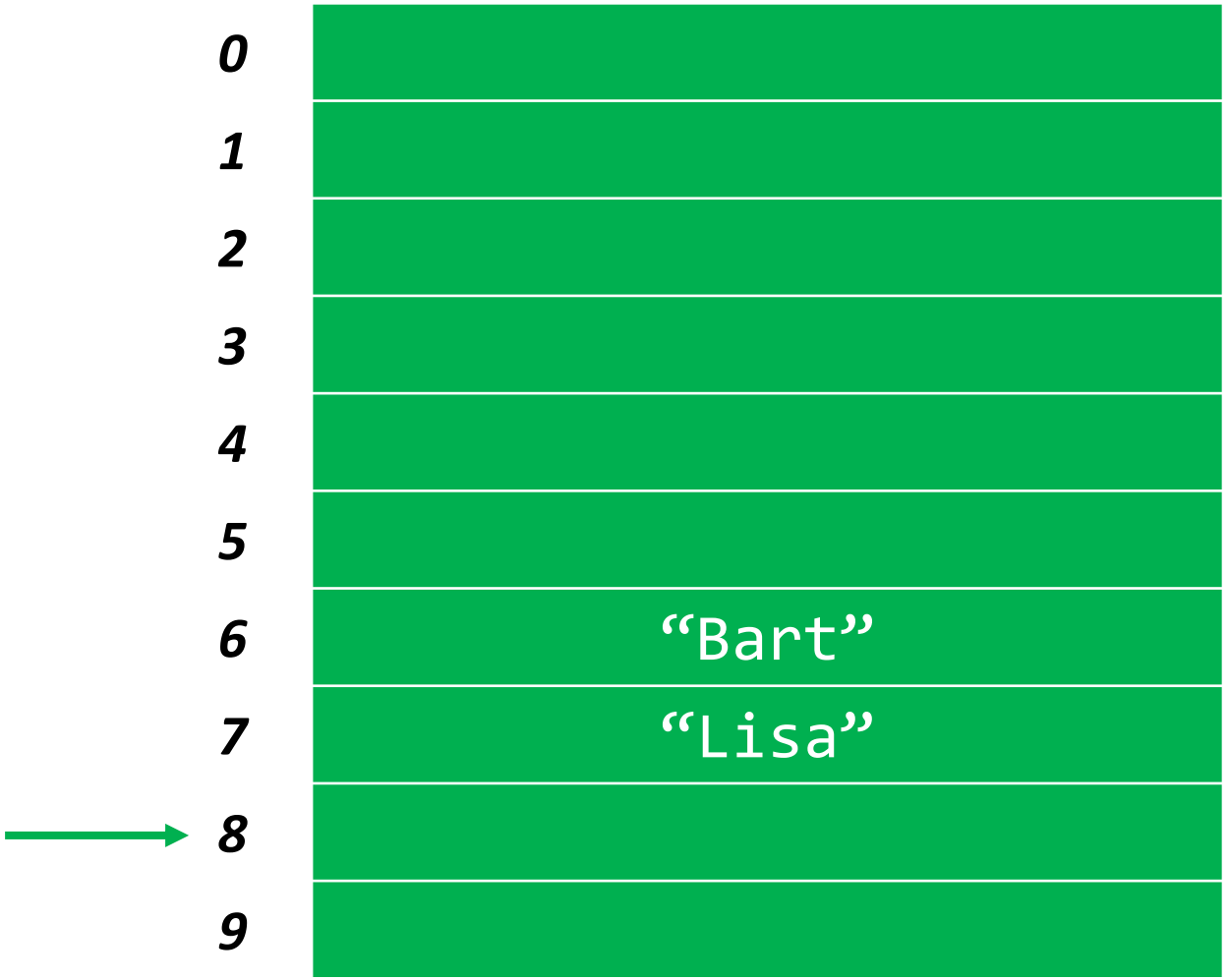


`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

Hash Tables

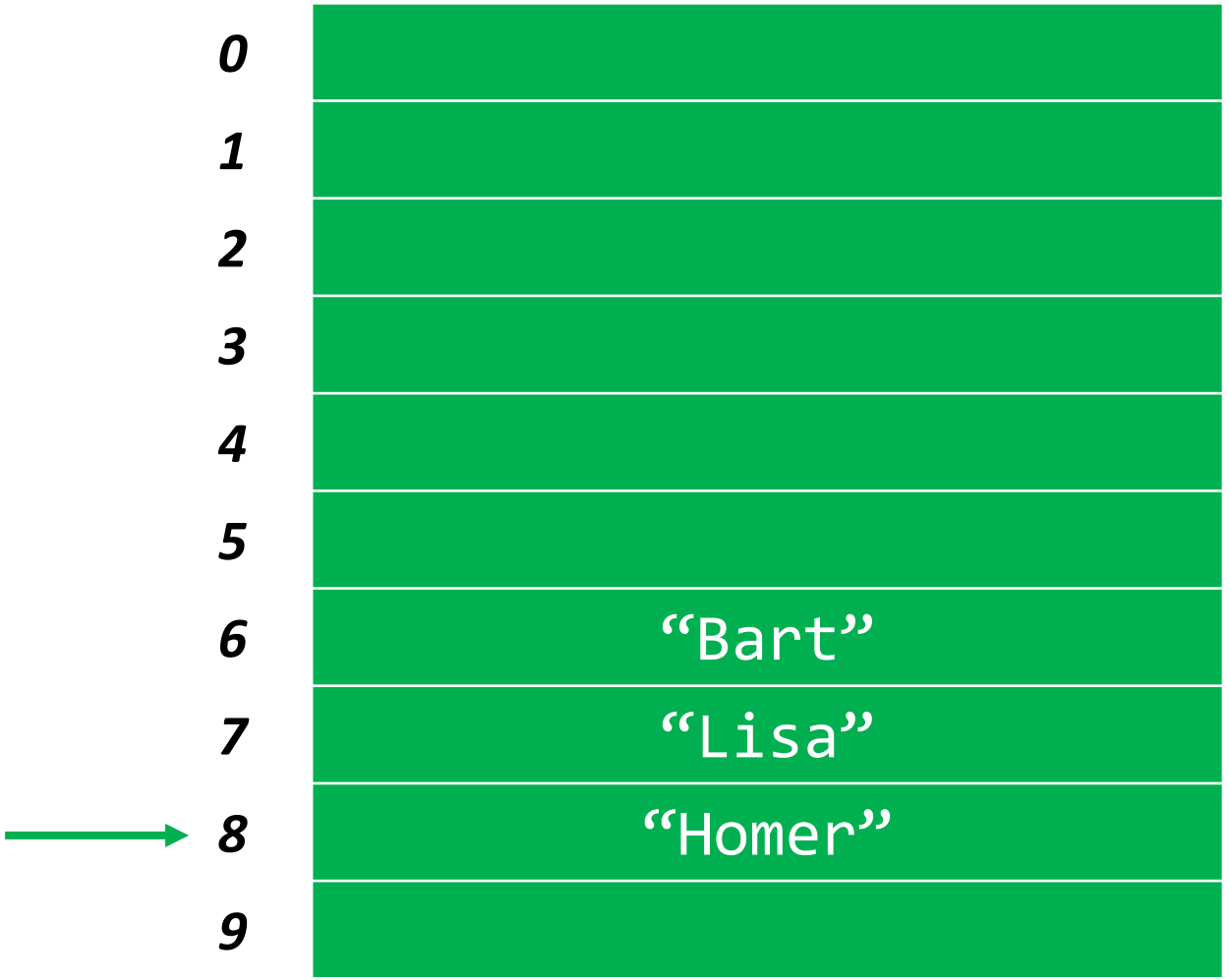


`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

Hash Tables

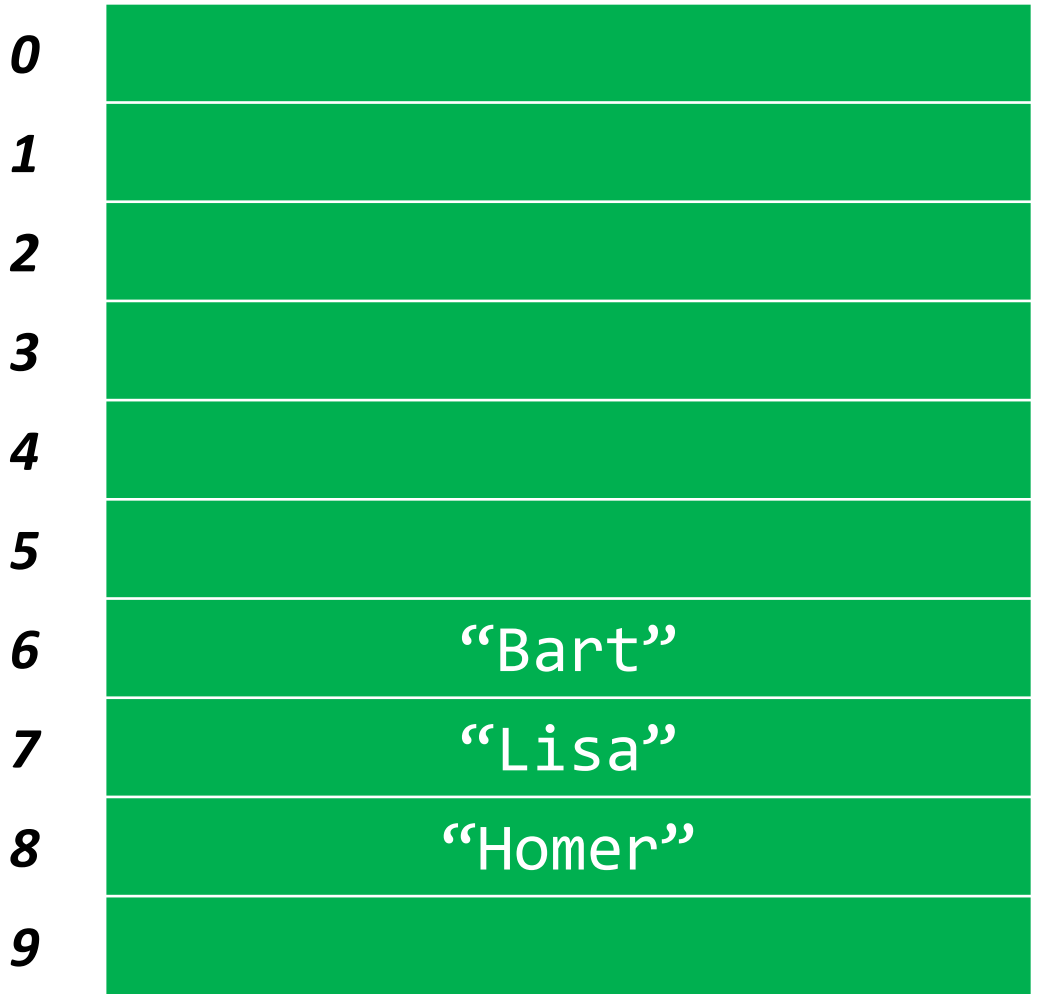


`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

Hash Tables



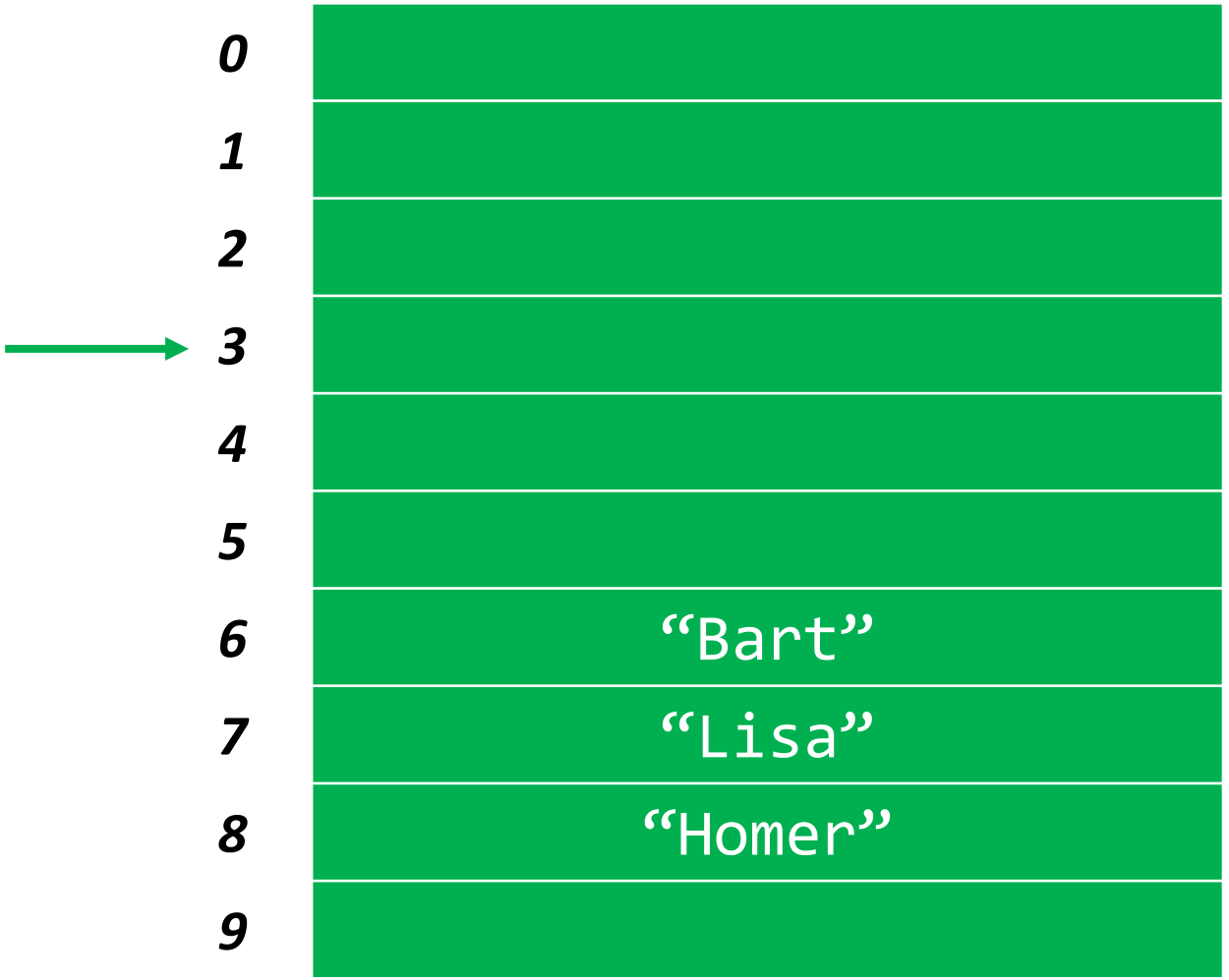
`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

Hash Tables



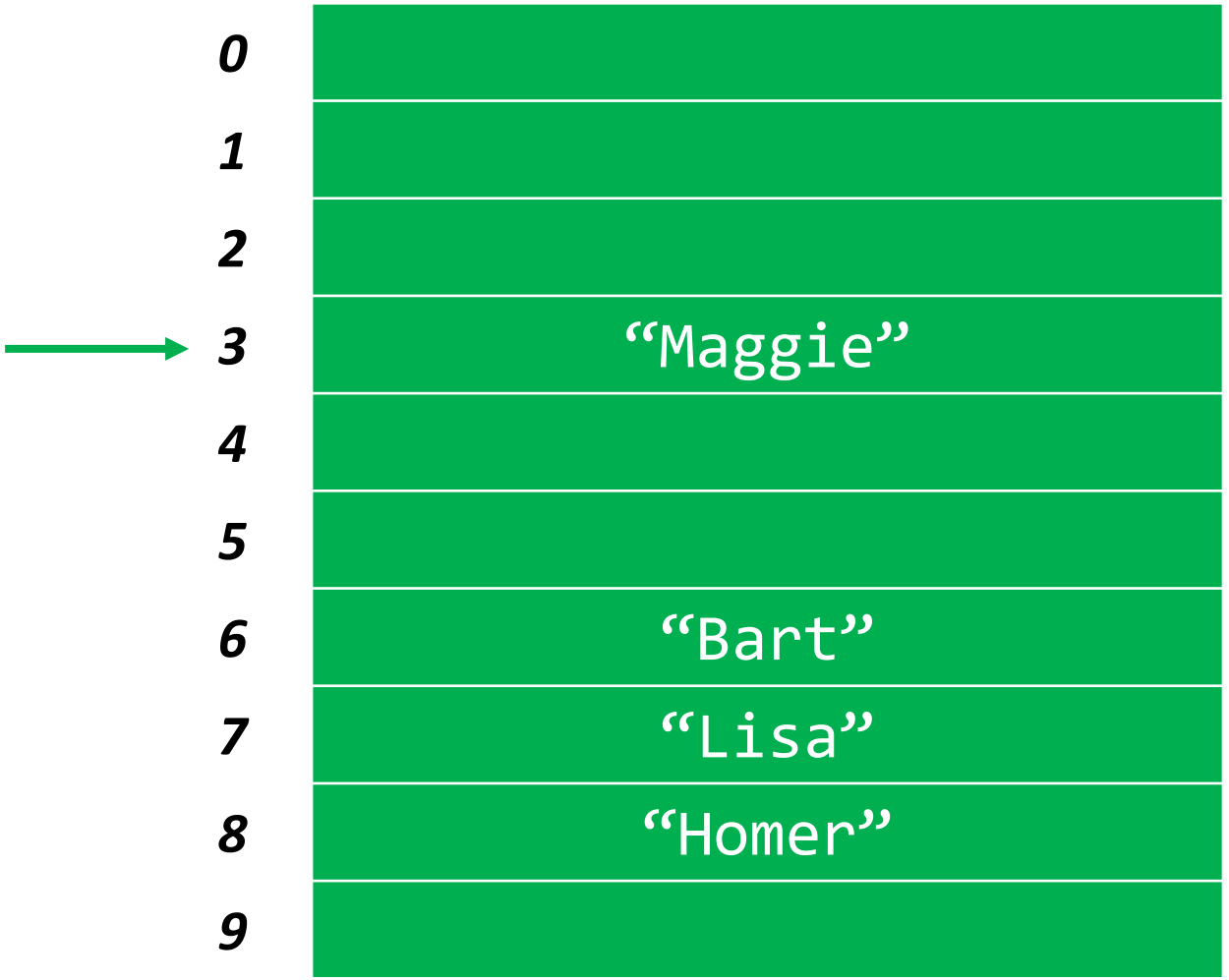
`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

Hash Tables



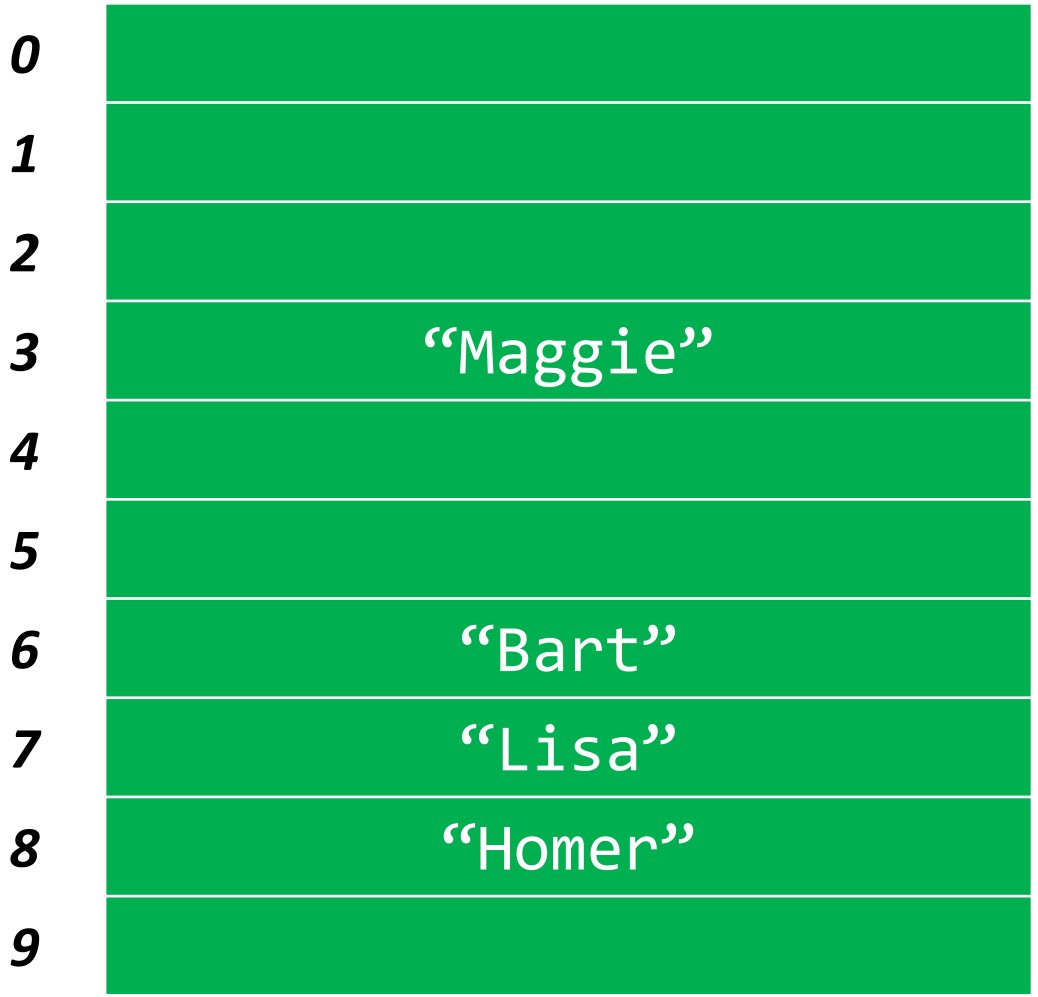
`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

Hash Tables



`hash(“Bart”);` returns **6**

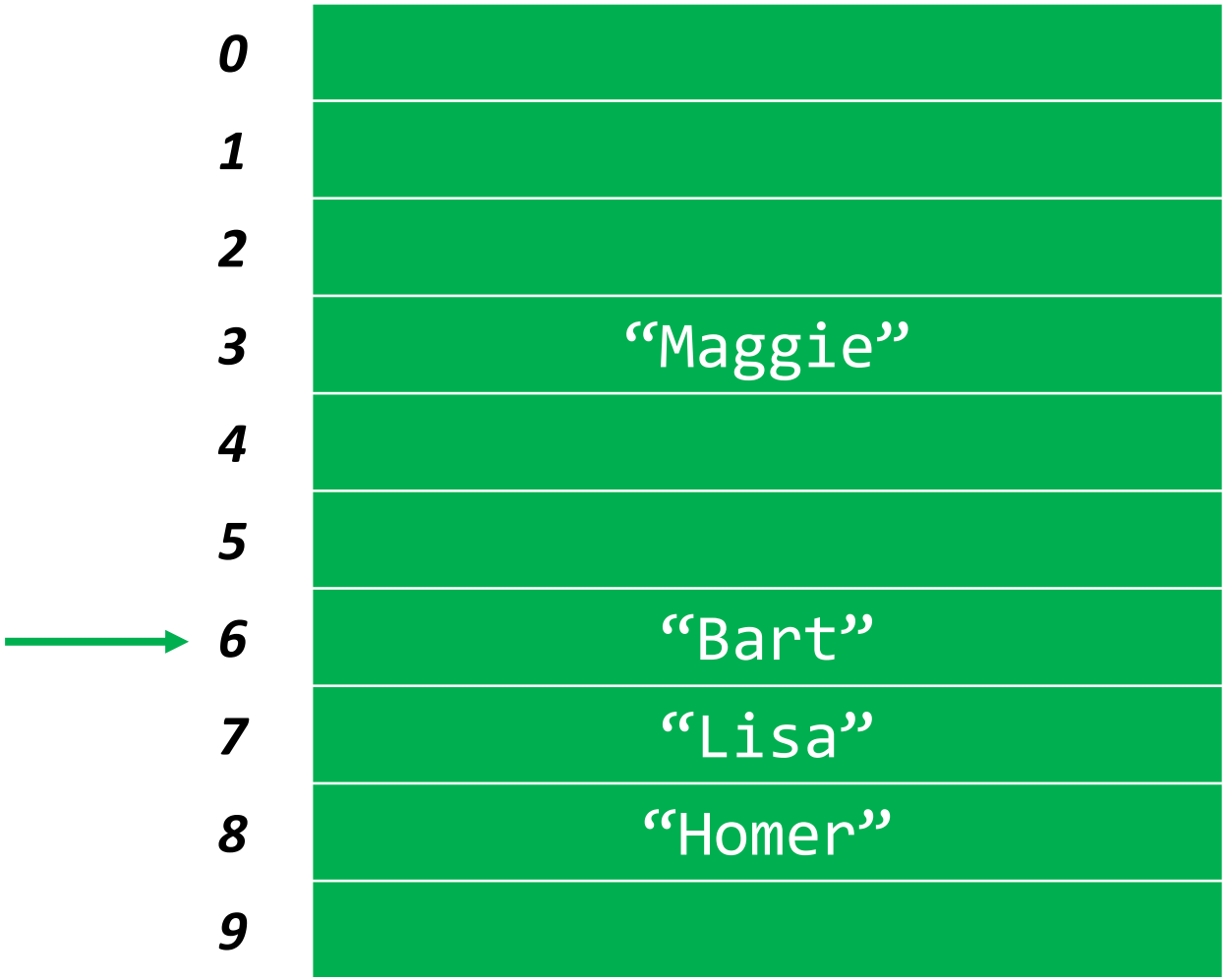
`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

`hash(“Marge”);` returns **6**

Hash Tables



`hash(“Bart”);` returns **6**

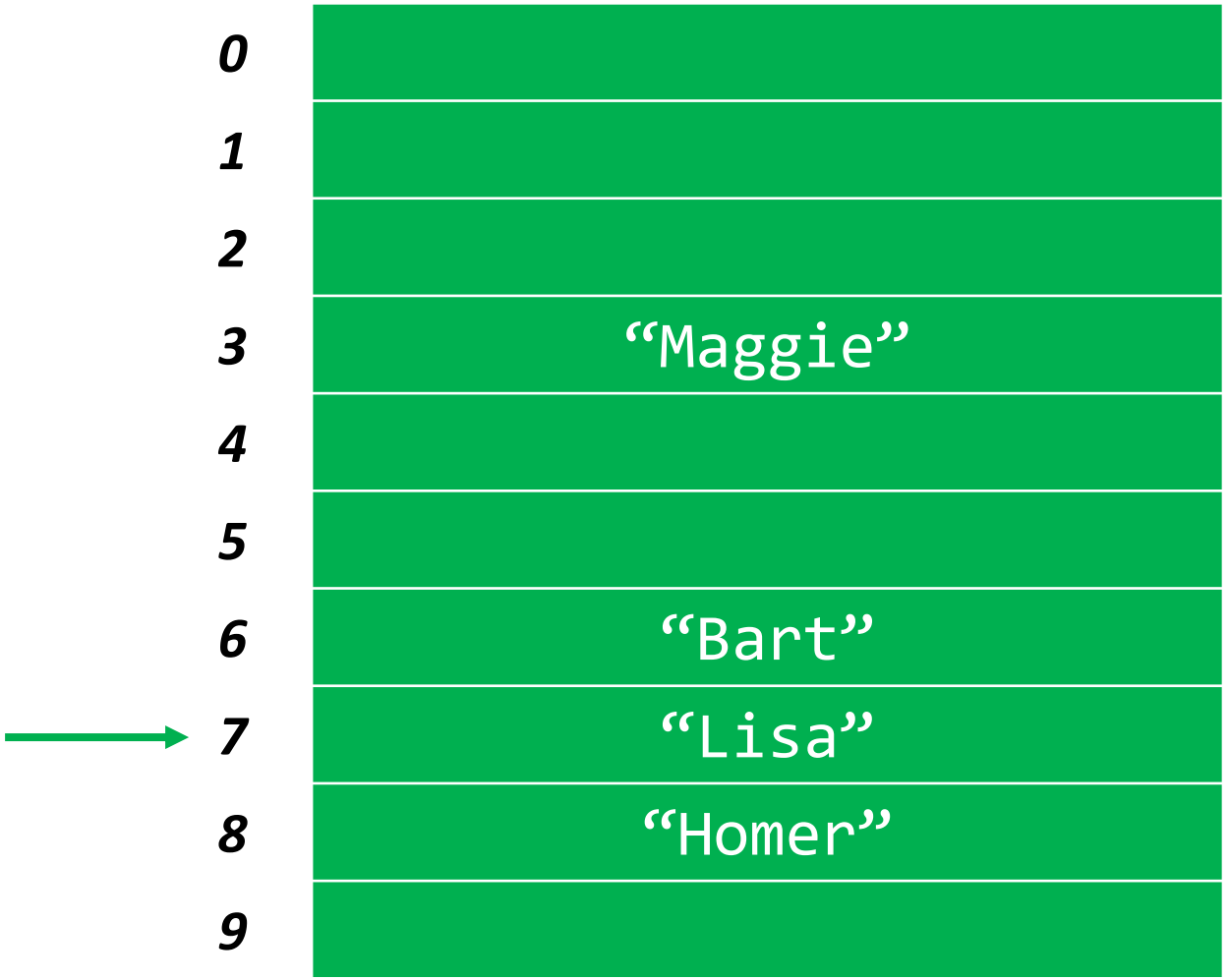
`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

`hash(“Marge”);` returns **6**

Hash Tables



`hash(“Bart”);` returns **6**

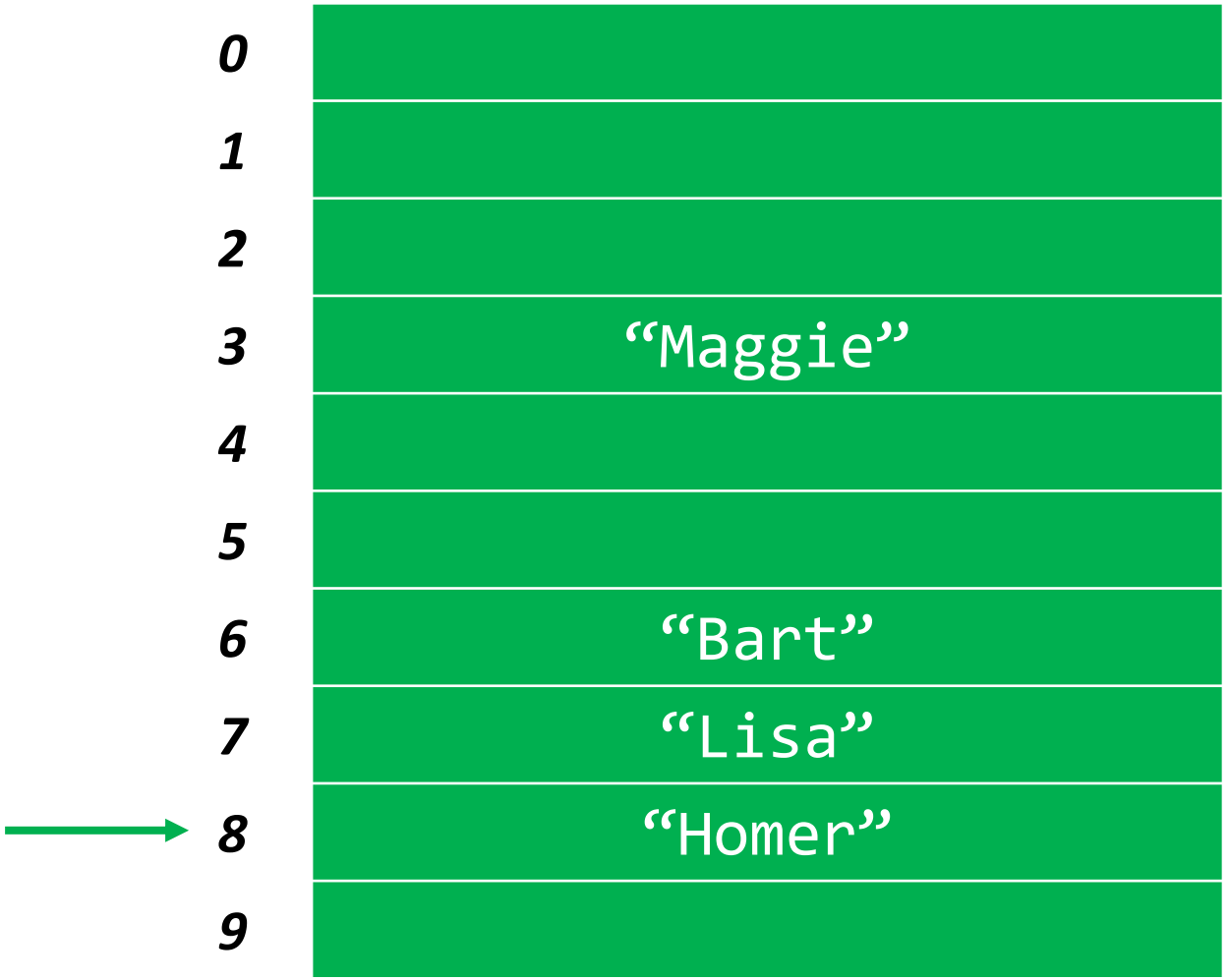
`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

`hash(“Marge”);` returns **6**

Hash Tables



`hash(“Bart”);` returns **6**

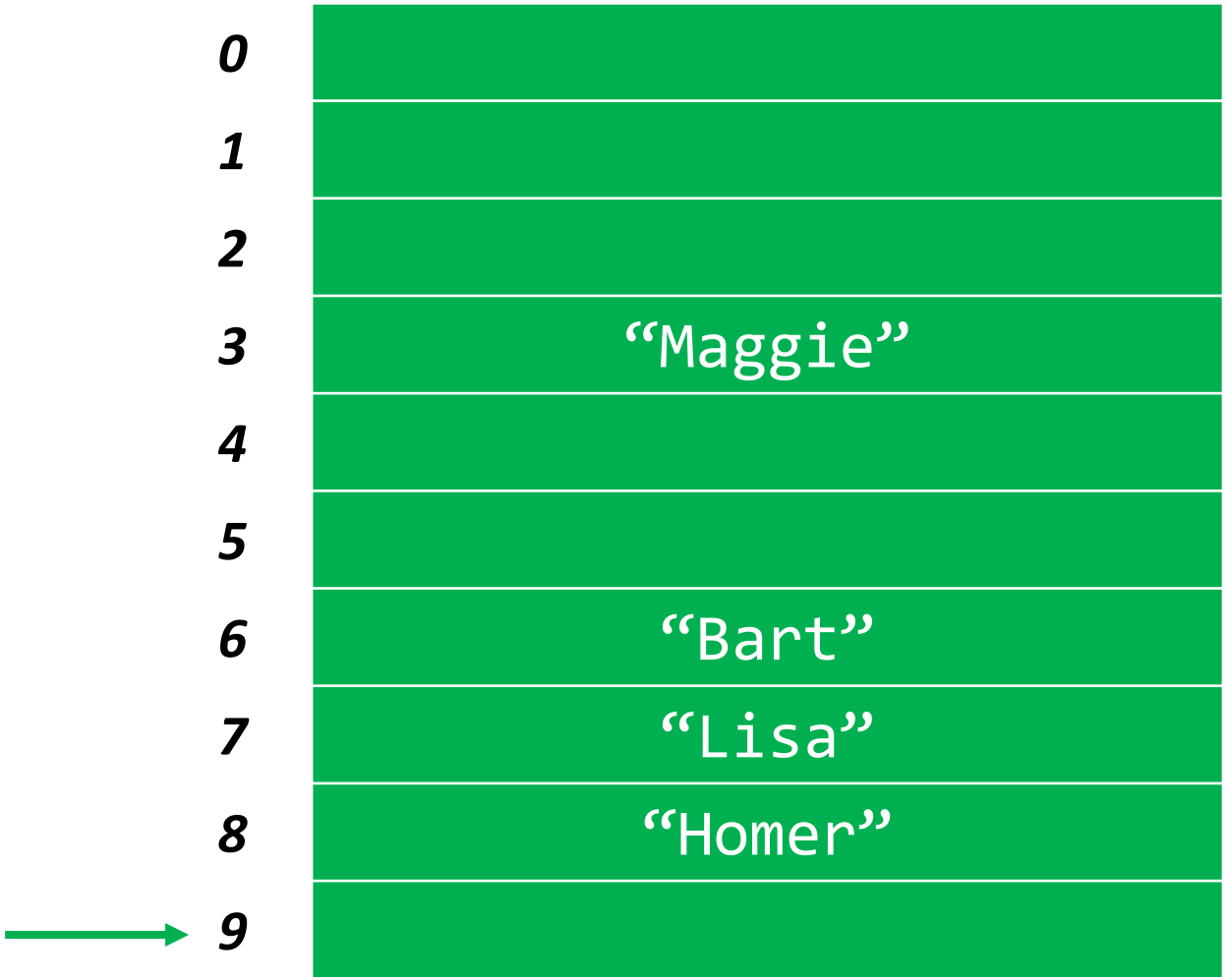
`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

`hash(“Marge”);` returns **6**

Hash Tables



`hash(“Bart”);` returns **6**

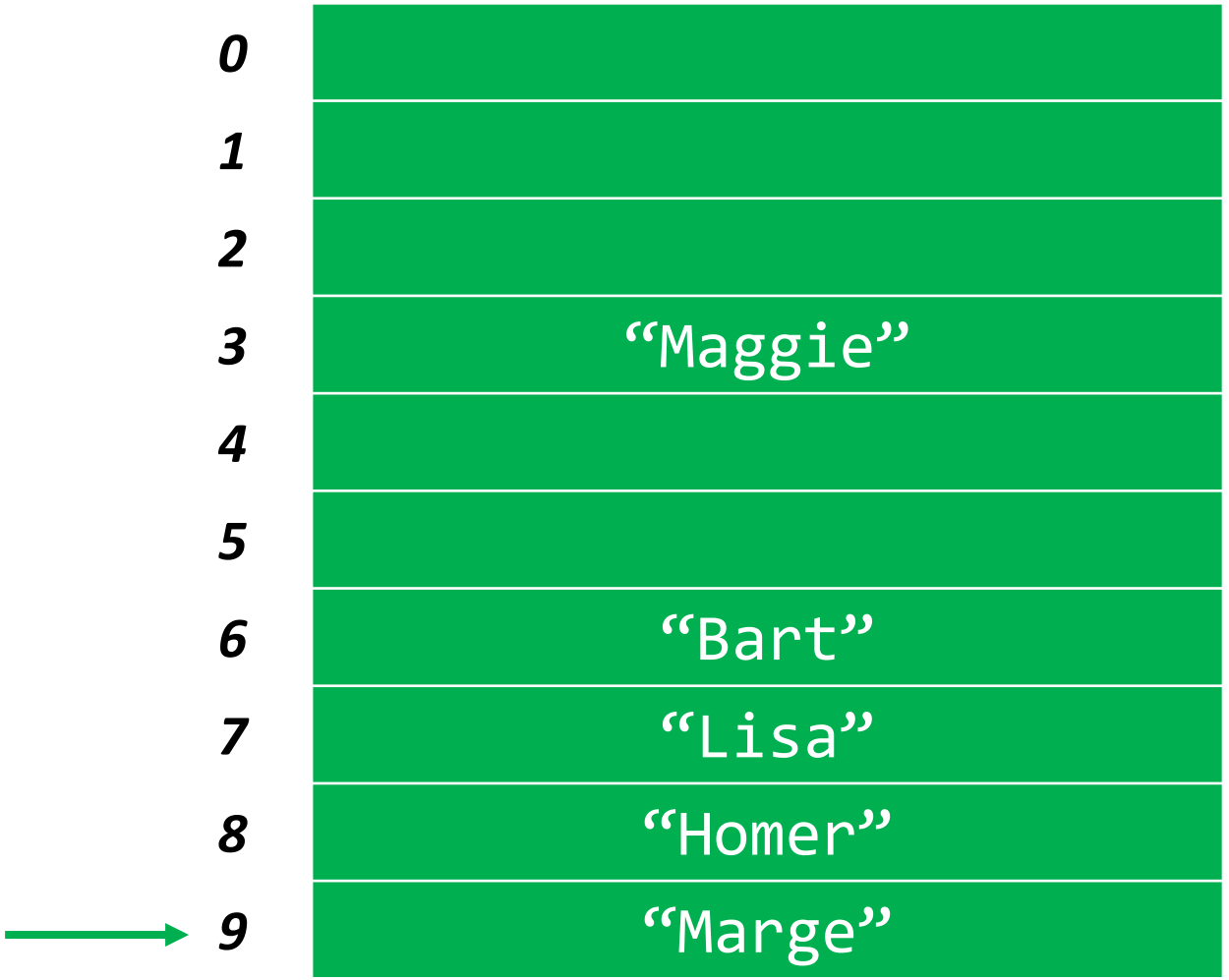
`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

`hash(“Marge”);` returns **6**

Hash Tables



`hash(“Bart”);` returns **6**

`hash(“Lisa”);` returns **6**

`hash(“Homer”);` returns **7**

`hash(“Maggie”);` returns **3**

`hash(“Marge”);` returns **6**

Hash Tables

- Resolving collisions: *Linear probing*

- Linear probing is subject to a problem called **clustering**. Once there's a miss, two adjacent cells will contain data, making it more likely in the future that the cluster will grow.

- Even if we switch to another probing technique, we're still limited. We can only store as much data as we have locations in our array.

Hash Tables

- Resolving collisions: *Chaining*

- Let’s start to pull it all together.

- What if instead of each element of the array holding just one piece of data, it held multiple pieces of data?

- If each element of the array is a pointer to the head of a linked list, then multiple pieces of data can yield the same hash code and we’ll be able to store it all!

Hash Tables

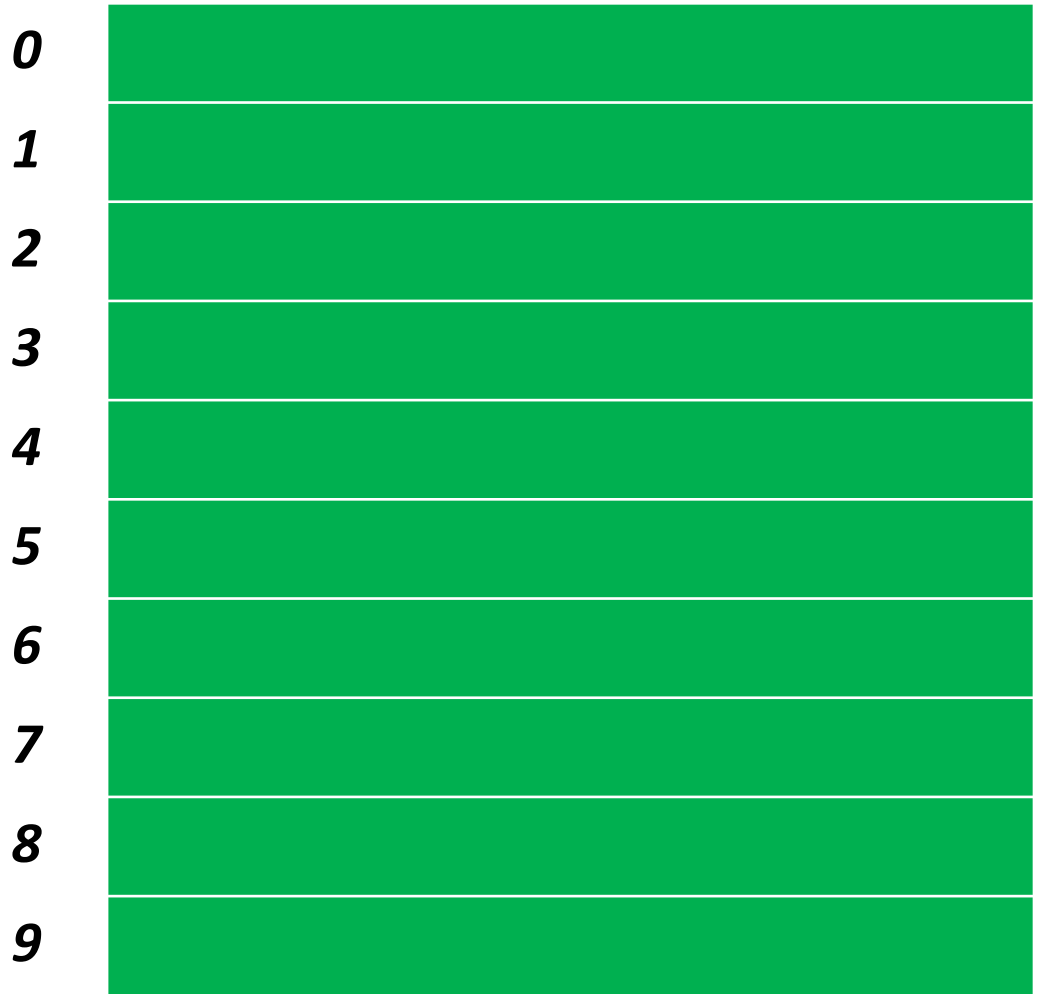
- Resolving collisions: *Chaining*

- We've eliminated clustering.

- We know from experience with linked lists that insertion (and creation, if necessary) into a linked list is an O(1) operation.

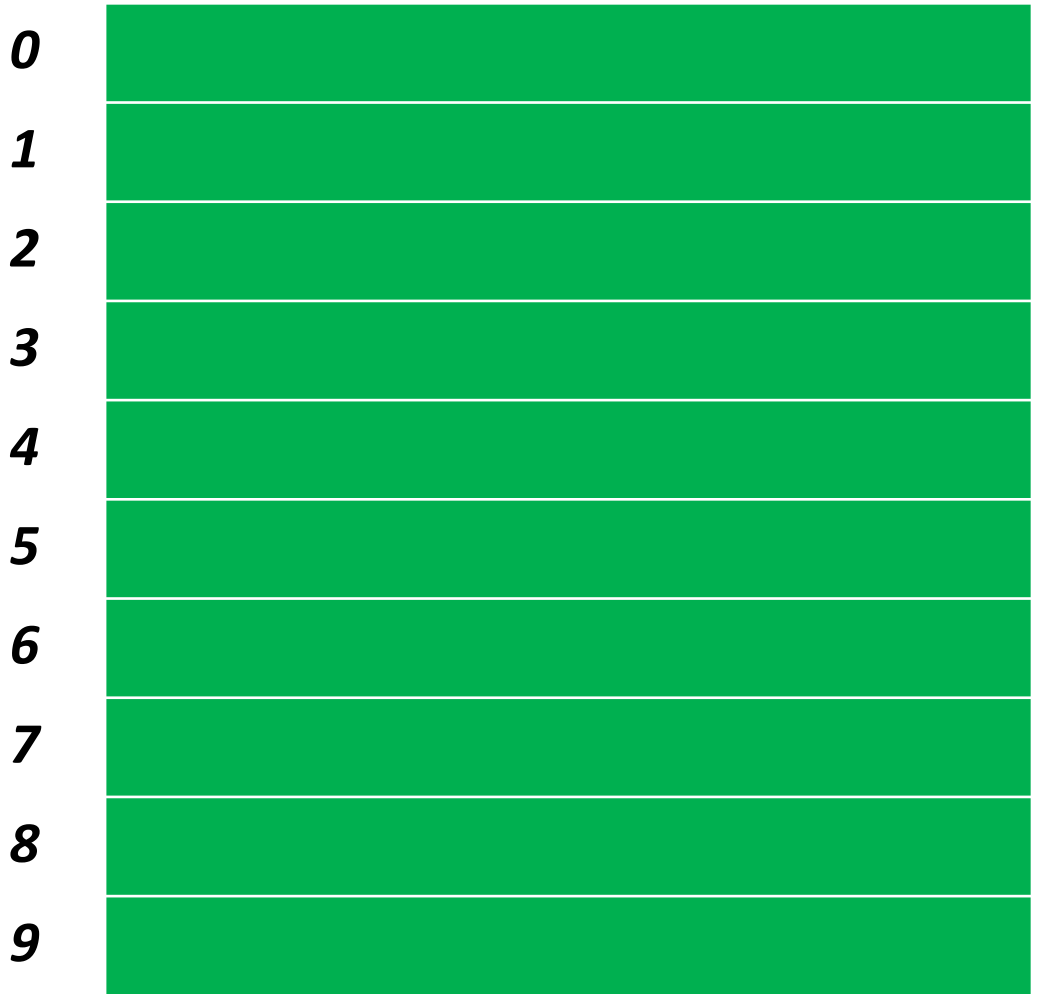
- For lookup, we only need to search through what is hopefully a small list, since we're distributing what would otherwise be one huge list across *n* lists.

Hash Tables



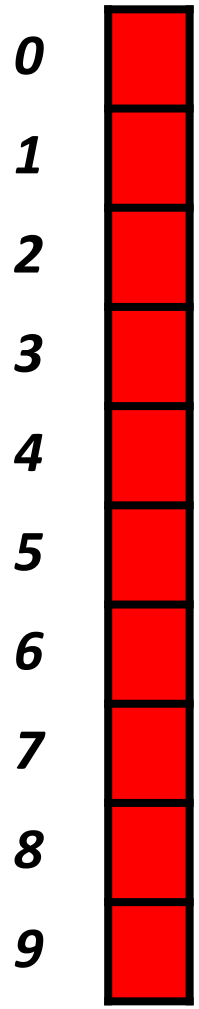

```
string hashtable[10];
```

Hash Tables



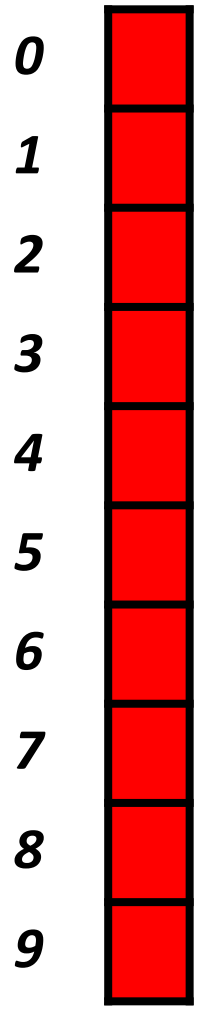
~~string hashtable[10];~~

Hash Tables



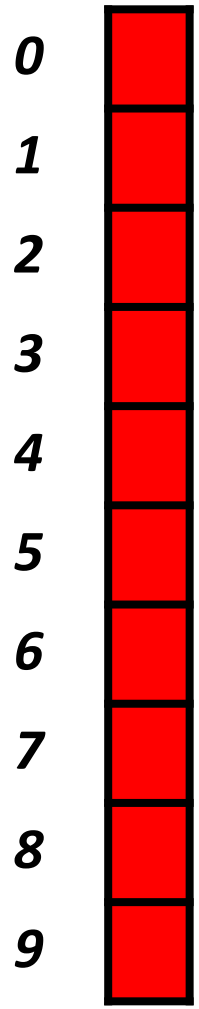

```
node* hashtable[10];
```

Hash Tables

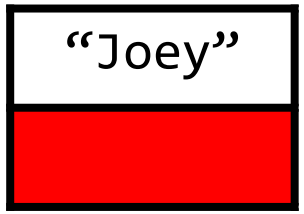


`hash("Joey");` returns **6**

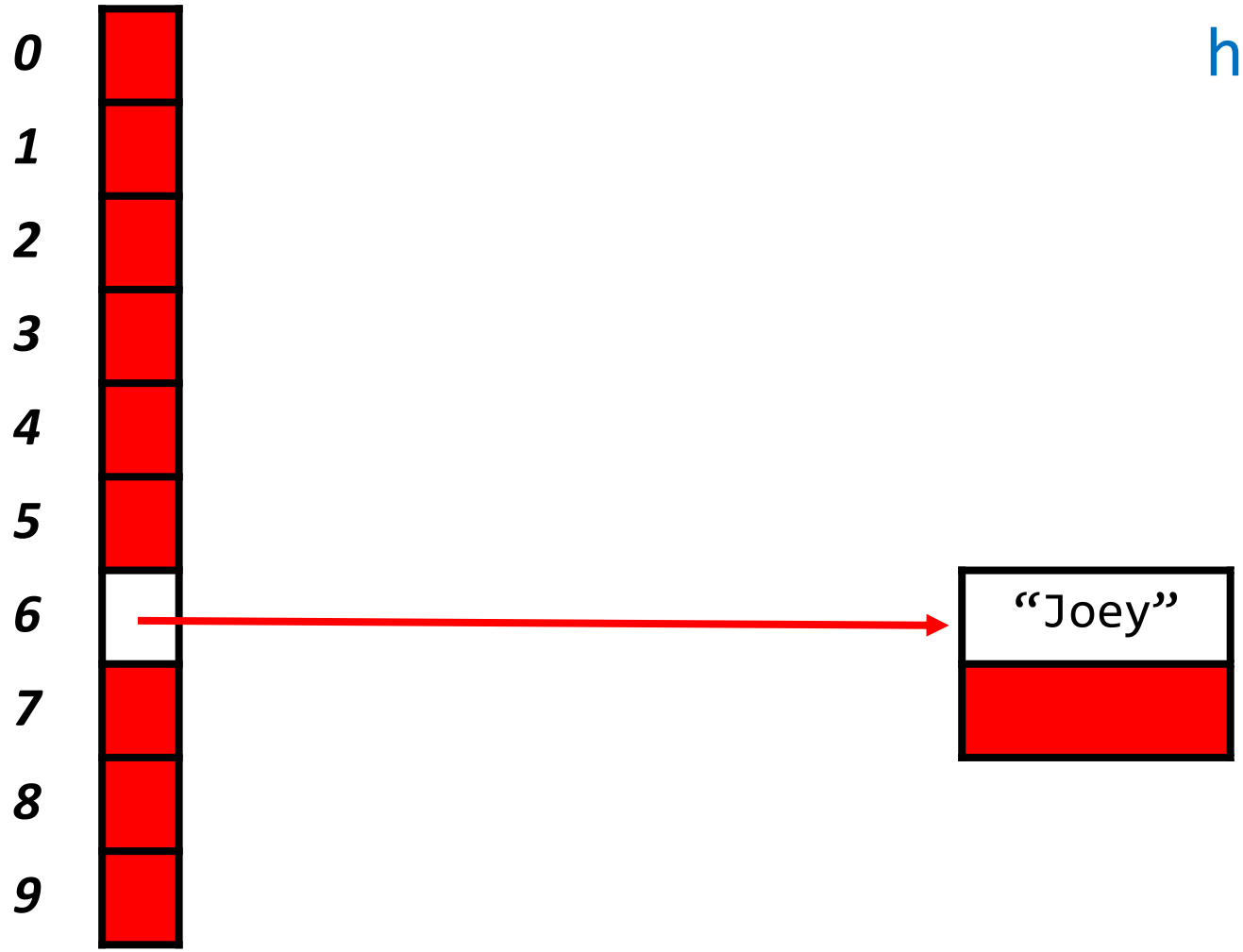
Hash Tables



`hash("Joey");` returns **6**

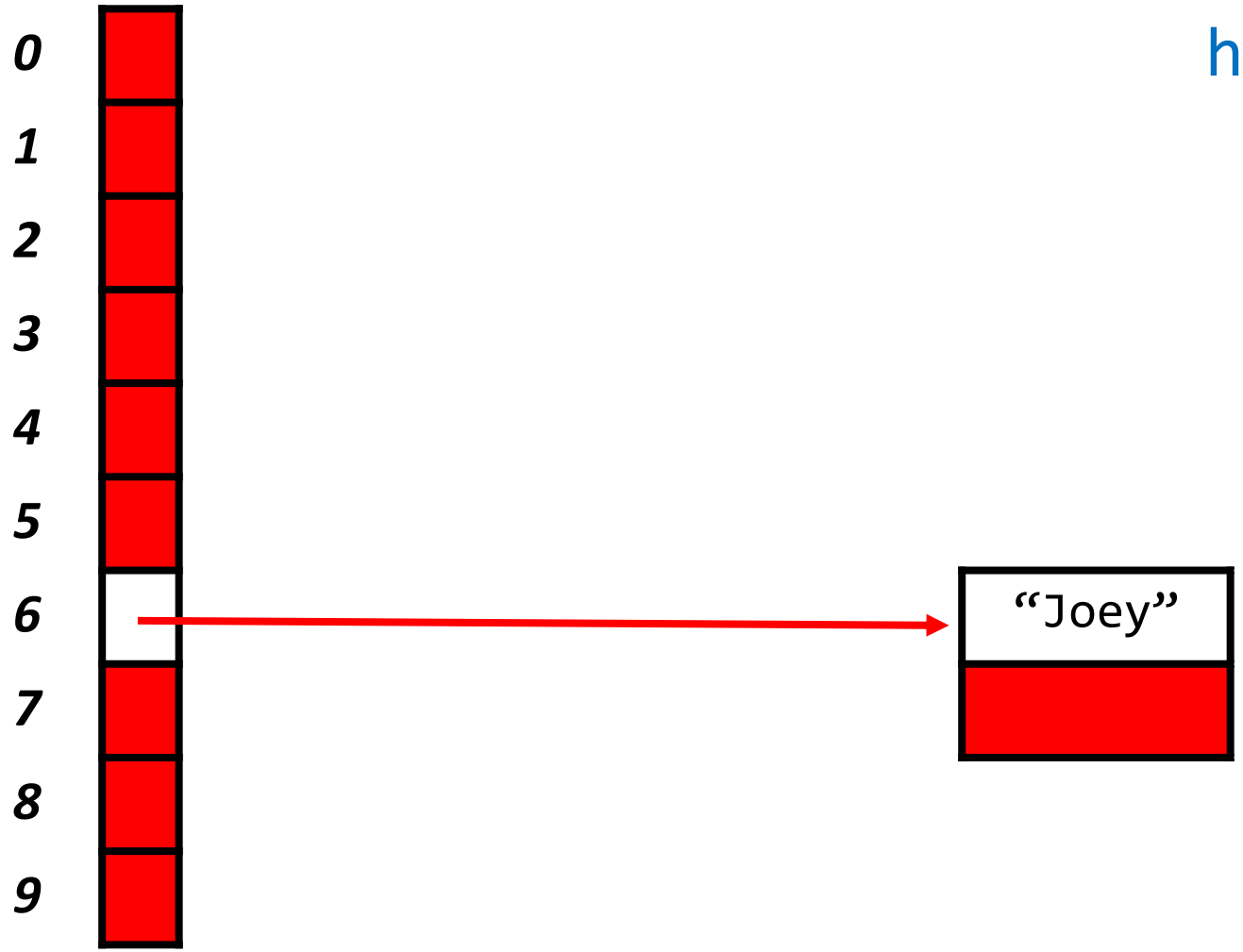


Hash Tables



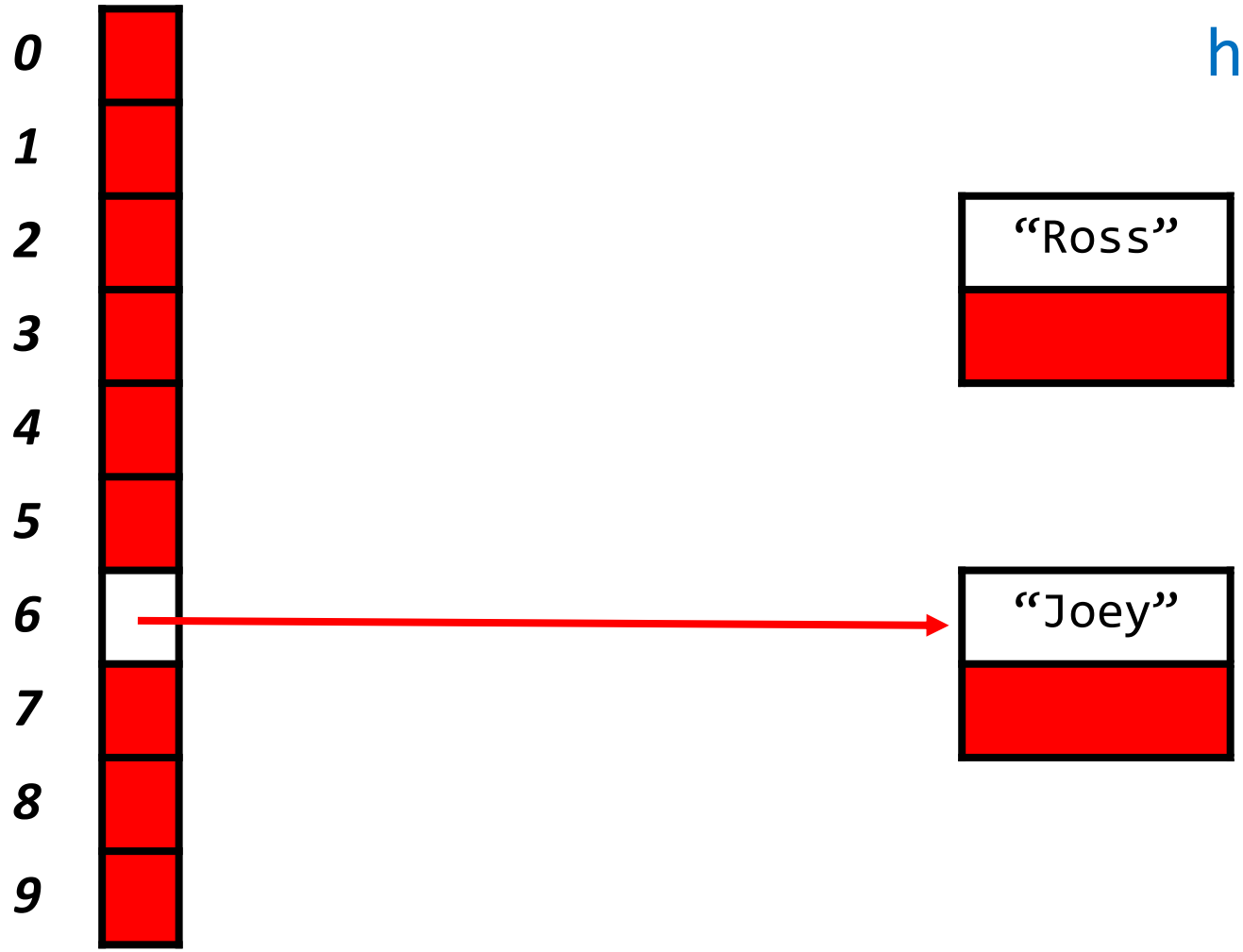
`hash(“Joey”);` returns **6**

Hash Tables



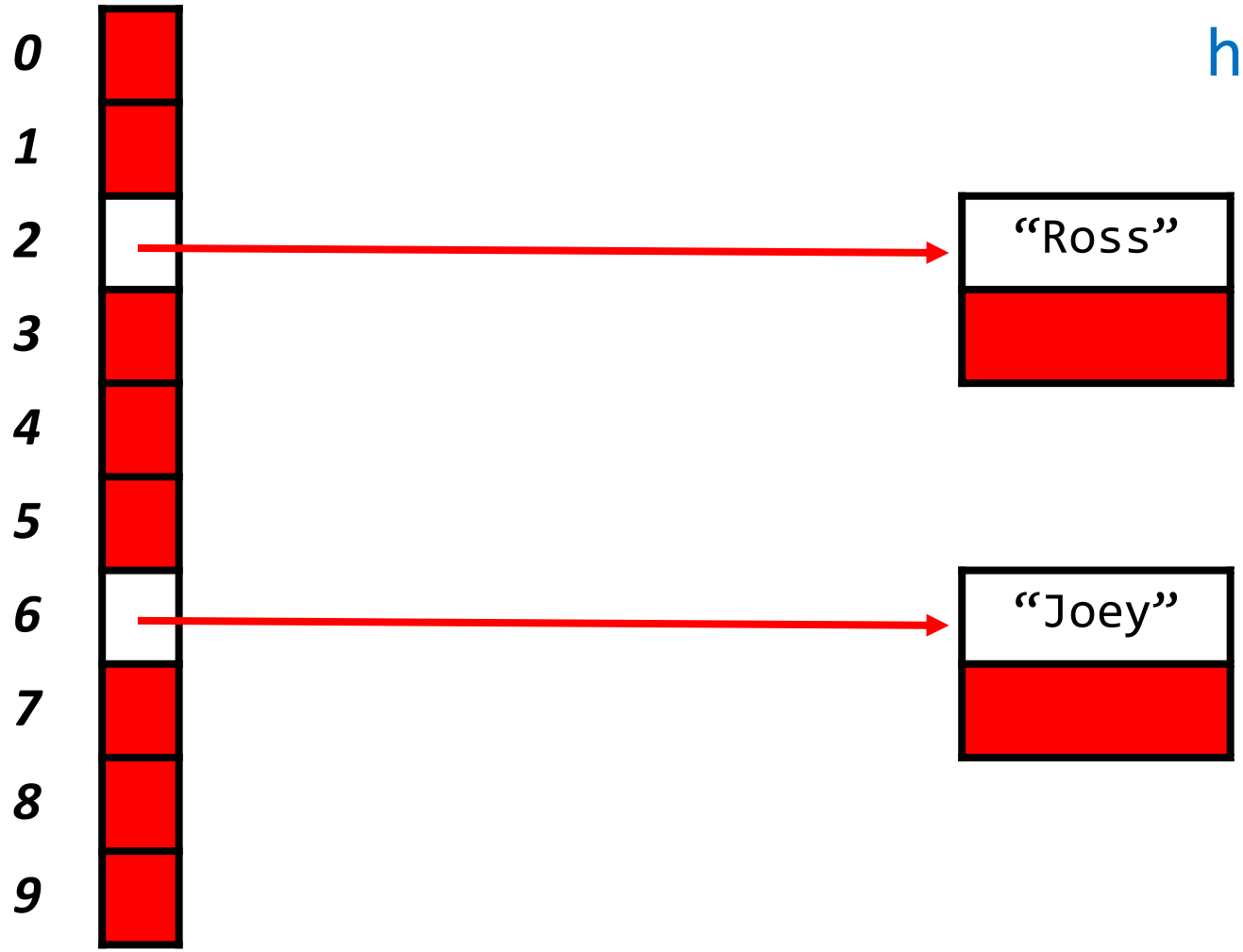
`hash(“Ross”);` returns **2**

Hash Tables



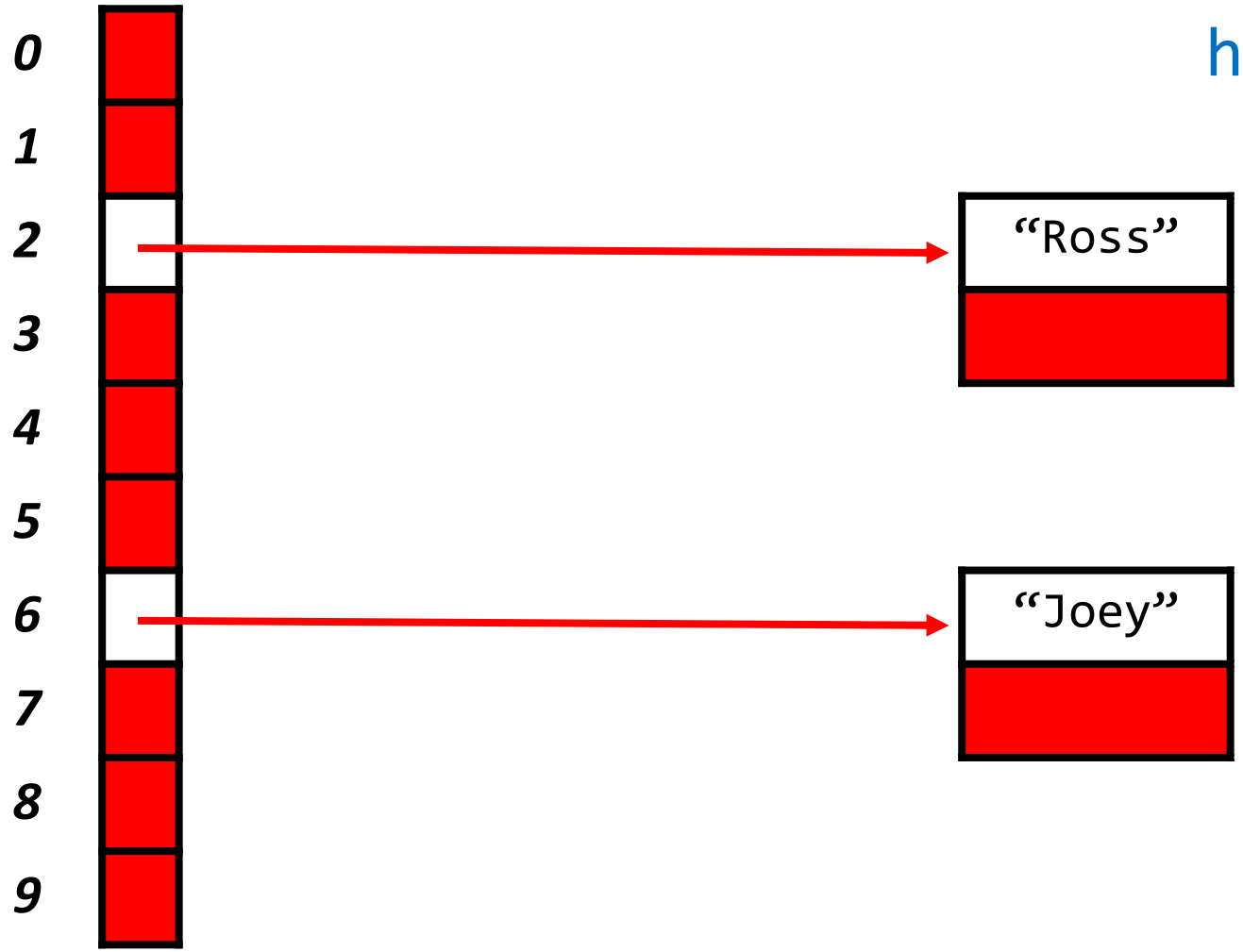
hash(“Ross”); returns **2**

Hash Tables



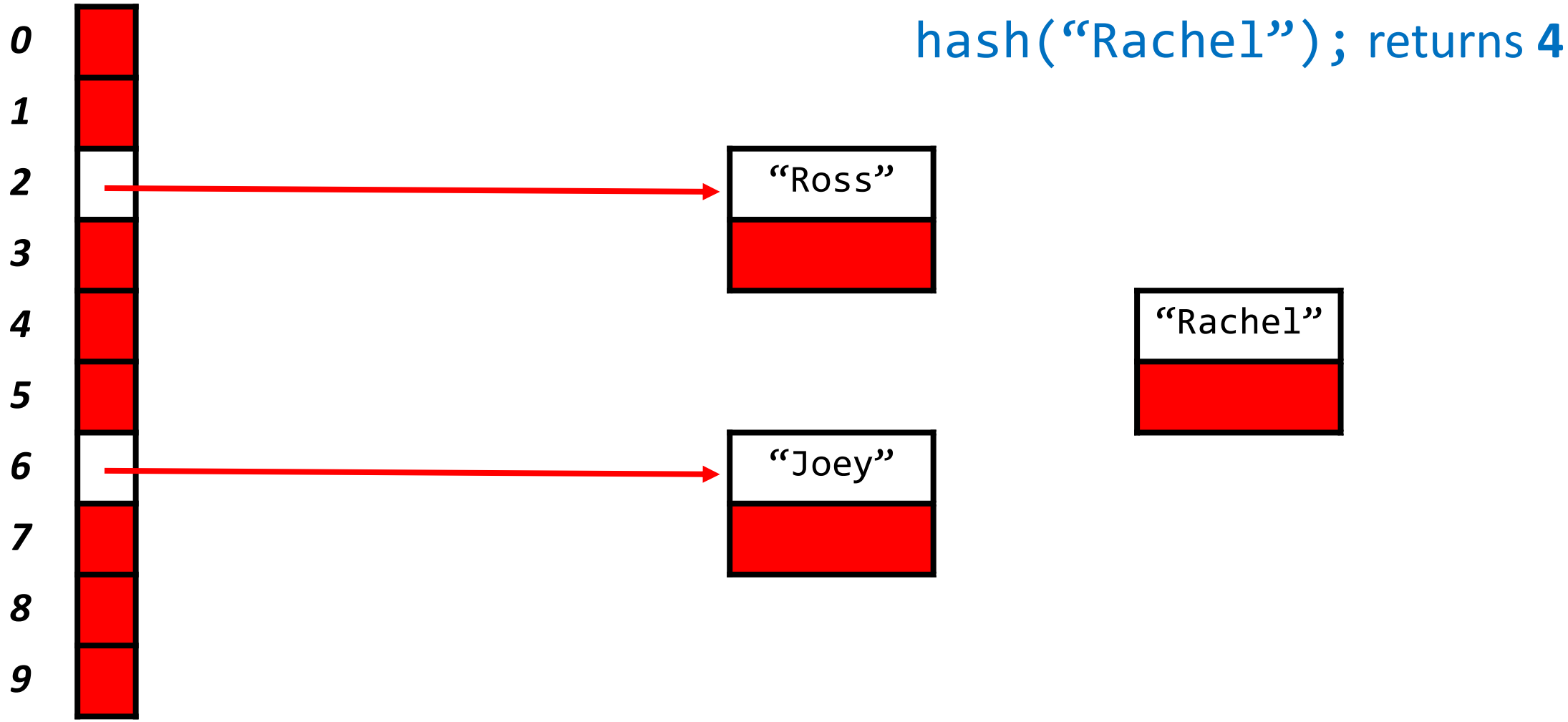
hash(“Ross”); returns **2**

Hash Tables

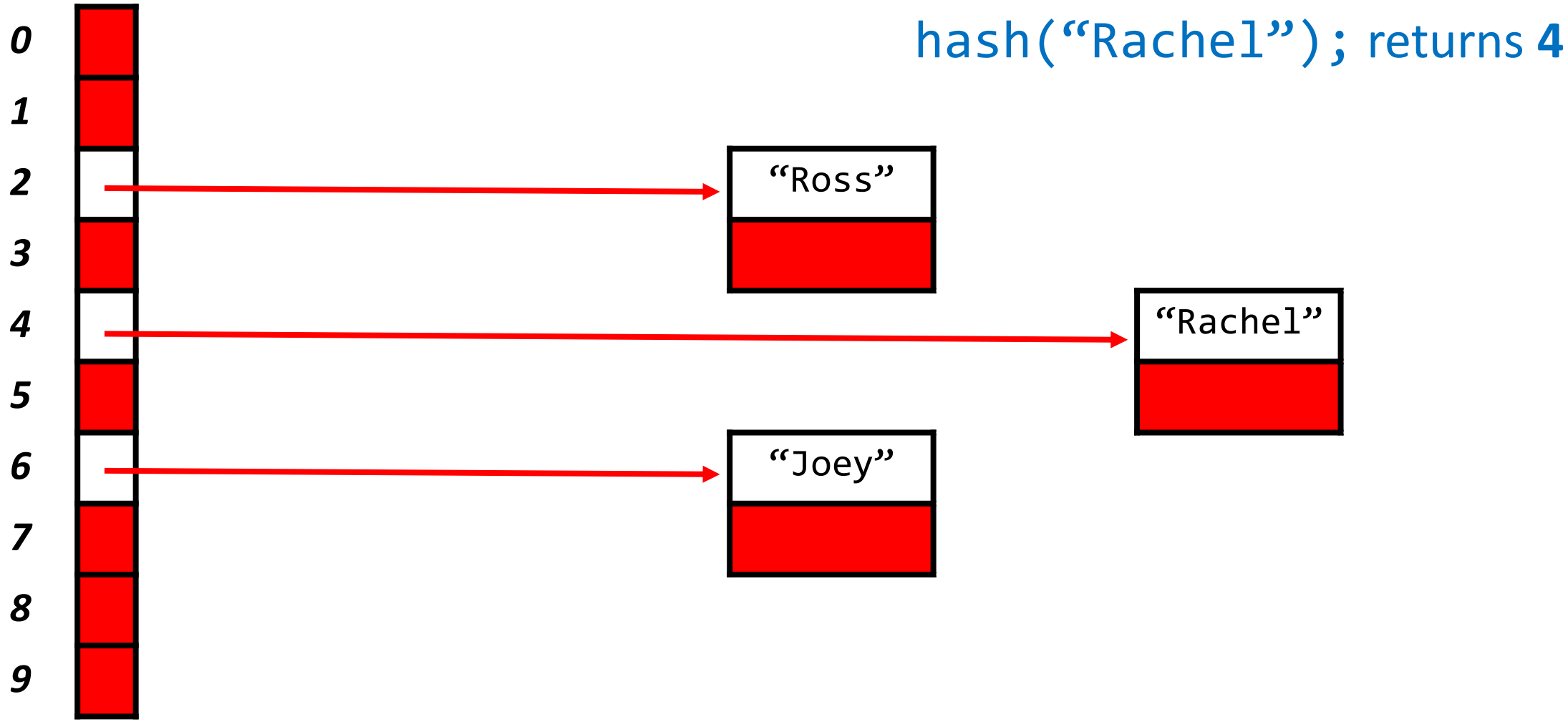


`hash(“Rachel”);` returns **4**

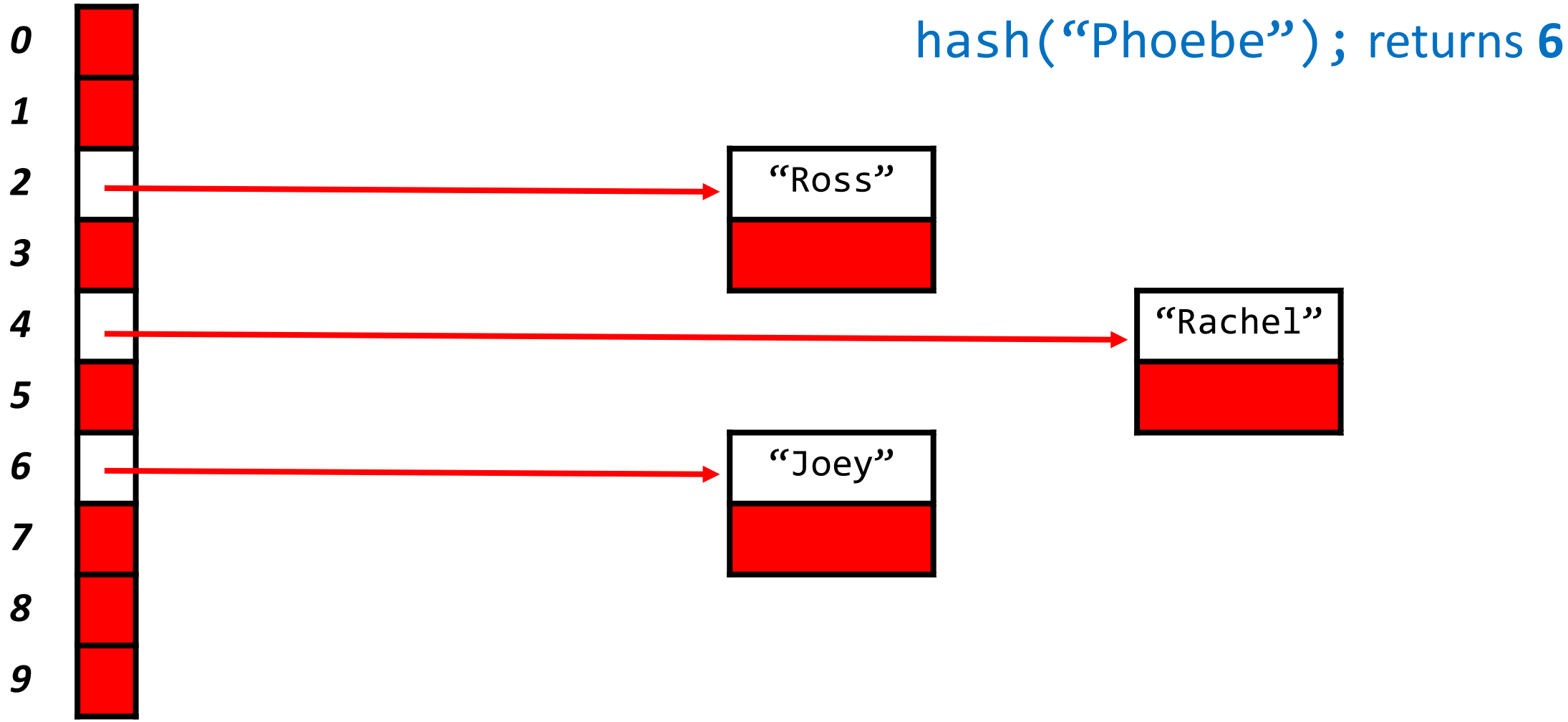
Hash Tables



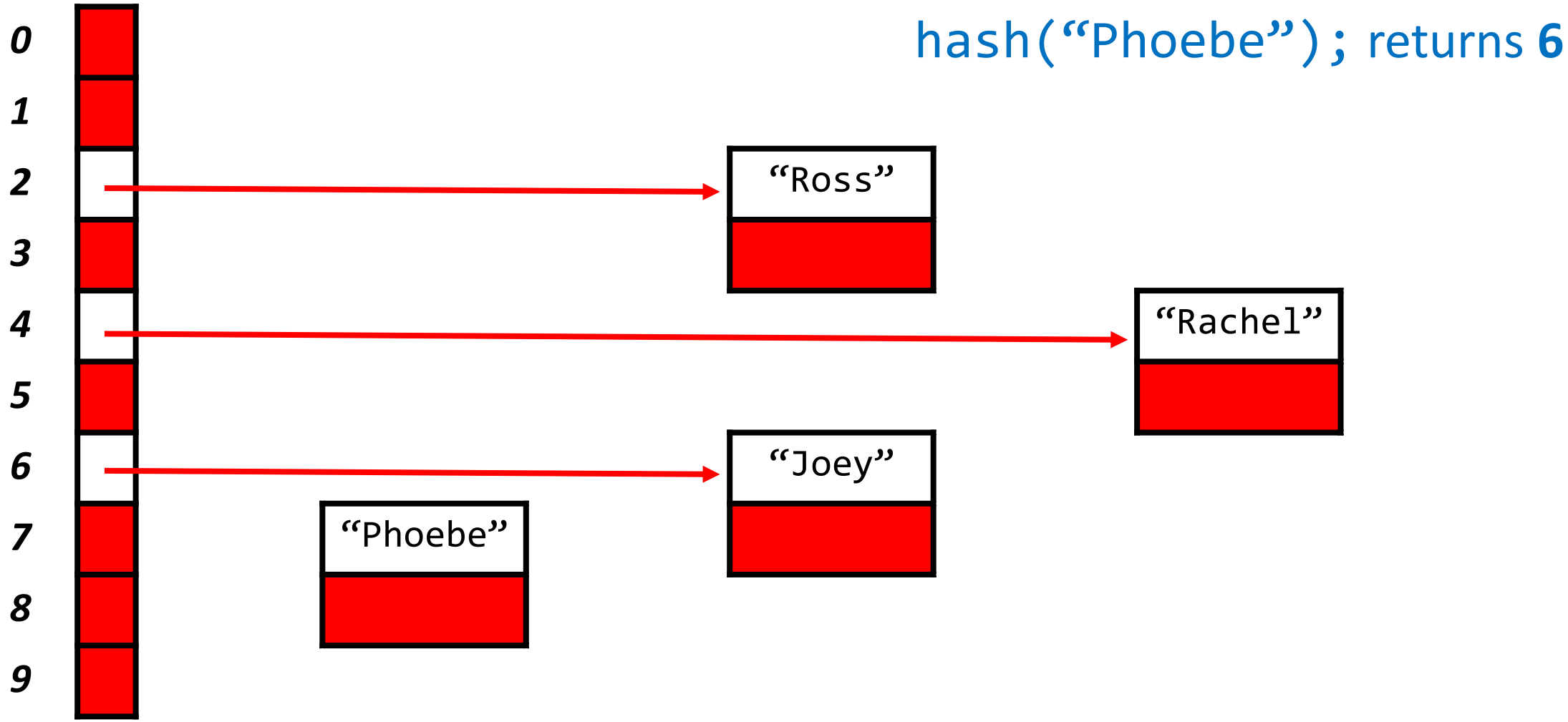
Hash Tables



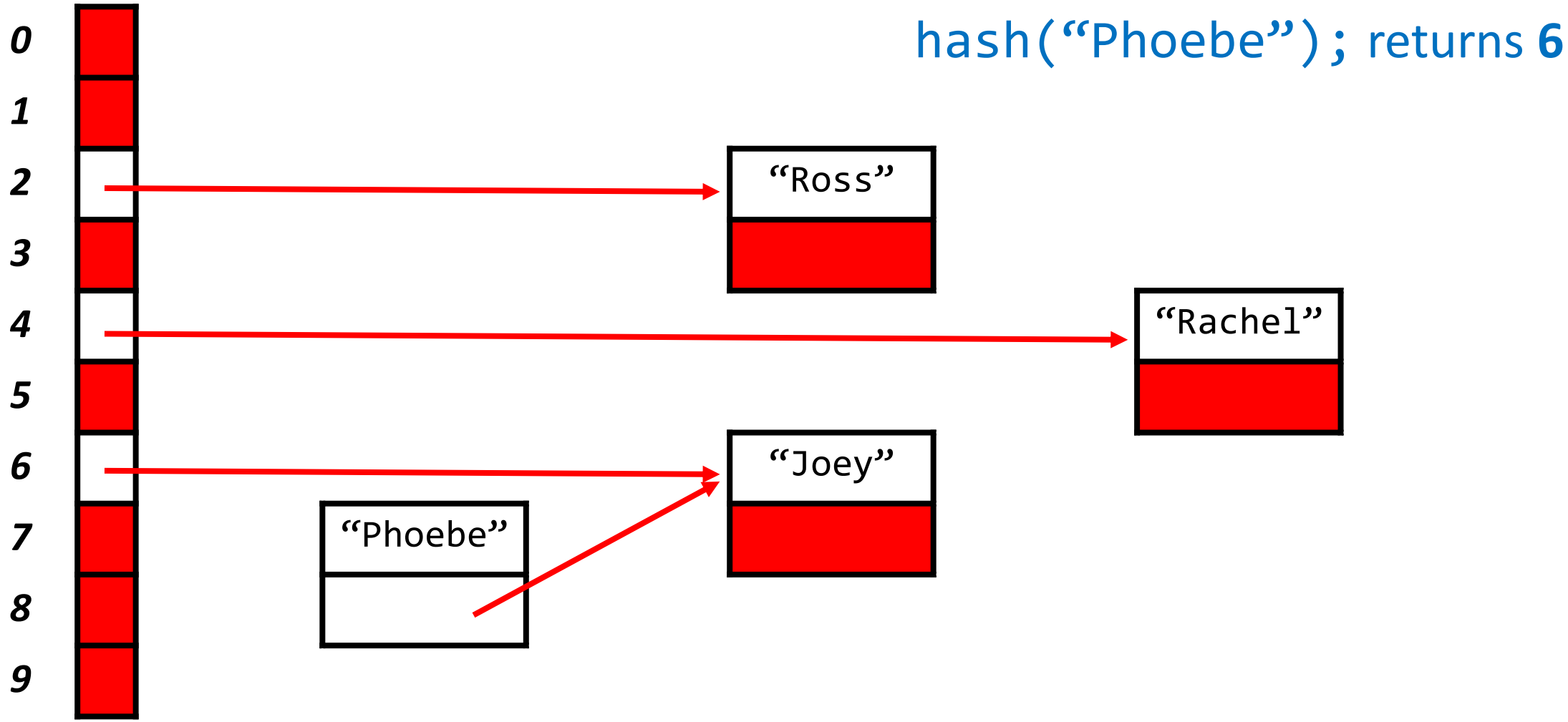
Hash Tables



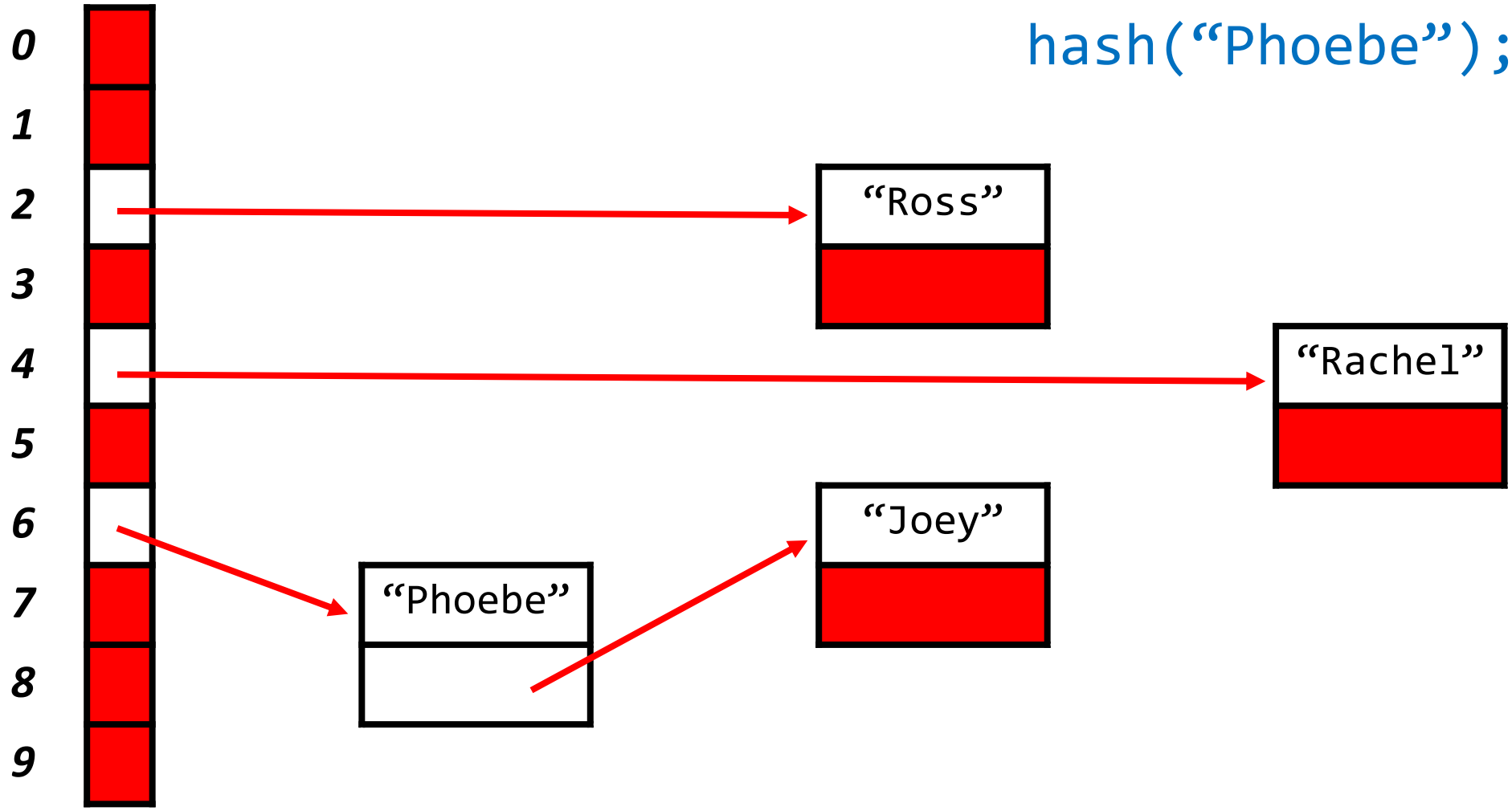
Hash Tables



Hash Tables



Hash Tables



hash("Phoebe"); returns **6**