HASHING

What is a hashtable?

- Hashtable : ADT
- The implementation of hash tables is frequently called hashing.
- Hashing is a technique used for performing finds, insertions and deletions in constant average time.

What is a hashtable?

- A ***hashtable*** is a collection of ⟨***key***, ***value***⟩ pairs.
- A ***hashtable*** is a *symbol table* that applies a *hash function* on a given key to look up the corresponding value in the table

key	**value**
s061234	Chan Tai Man, Male, 91234567
s976543	Mo Ming Si, Male, 90807060
s051351	Ng Chi Ming, Female, 33445566
s042465	Yum Ho Yan, Female, 98765432
…	…

Without Hashing

Searching a person by matching each SID

We have to delete or insert items from time to time.

SID	value
s061234	Chan Tai Man, Male, 91234567
s976543	Mo Ming Si, Male, 90807060
s051351	Ng Chi Ming, Female, 33445566
s042465	Yum Ho Yan, Female, 98765432
…	…

Hash function

- The key is mapped into a position in the hash table.
- The mapping is called hash function

The size of the table is fixed.

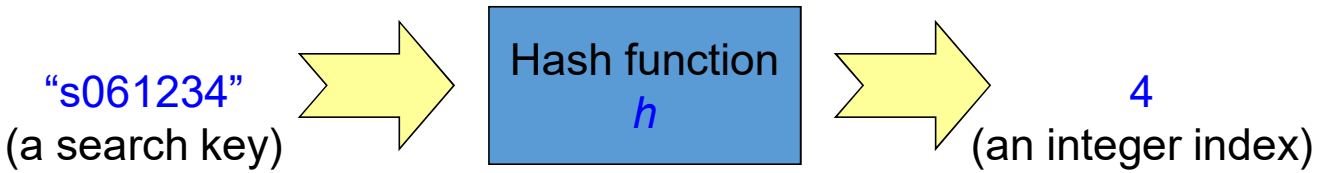
key	address
s061234	4
s976543	3
s051351	1
s042465	5
…	…

address	value
0	
1	Ng Chi Ming, Female, 33445566
2	
3	Mo Ming Si, Male, 90807060
4	Chan Tai Man, Male, 91234567
5	Yum Ho Yan, Female, 98765432
…	
9	

Hash function

- ***Hash function*** is the function that maps the key into a value that indicates the location of the value.

- Example
  - h("s061234") → 4
  - h("s976543") → 3



With Hash

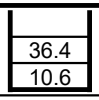
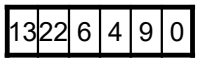
- Searching is done by finding h(key).
- It is easy to do insertion and deletion of a record.

key	h(key)
s061234	4
s976543	3
s051351	1
s042465	5
…	…

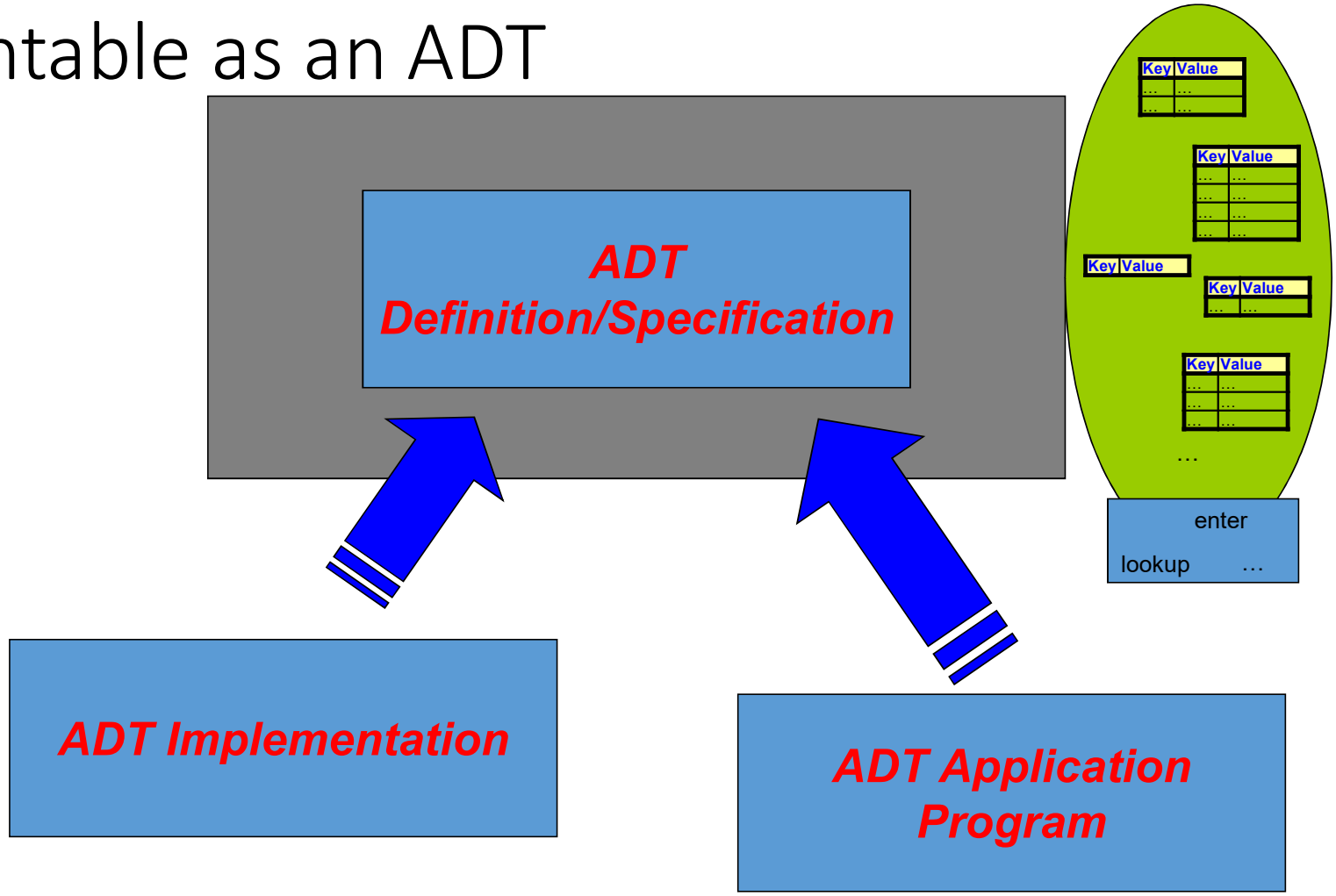
address	value
0	
1	Ng Chi Ming, Female, 33445566
2	
3	Mo Ming Si, Male, 90807060
4	Chan Tai Man, Male, 91234567
5	Yum Ho Yan, Female, 98765432
…	
9	

What is a hashtable?

- "key" and "value" can be anything.
- "value" can be composed of multiple parts.

key	value
	stack, double, 2
	queue, integer, 5
	array, integer, 6
…	…

Hashtable as an ADT



Defining the hashtable ADT

- Before giving the function prototypes, we need to decide the *data types* of “key” and “value”.

- Since “value” can be *anything*, we define its type as void * (generic pointer).

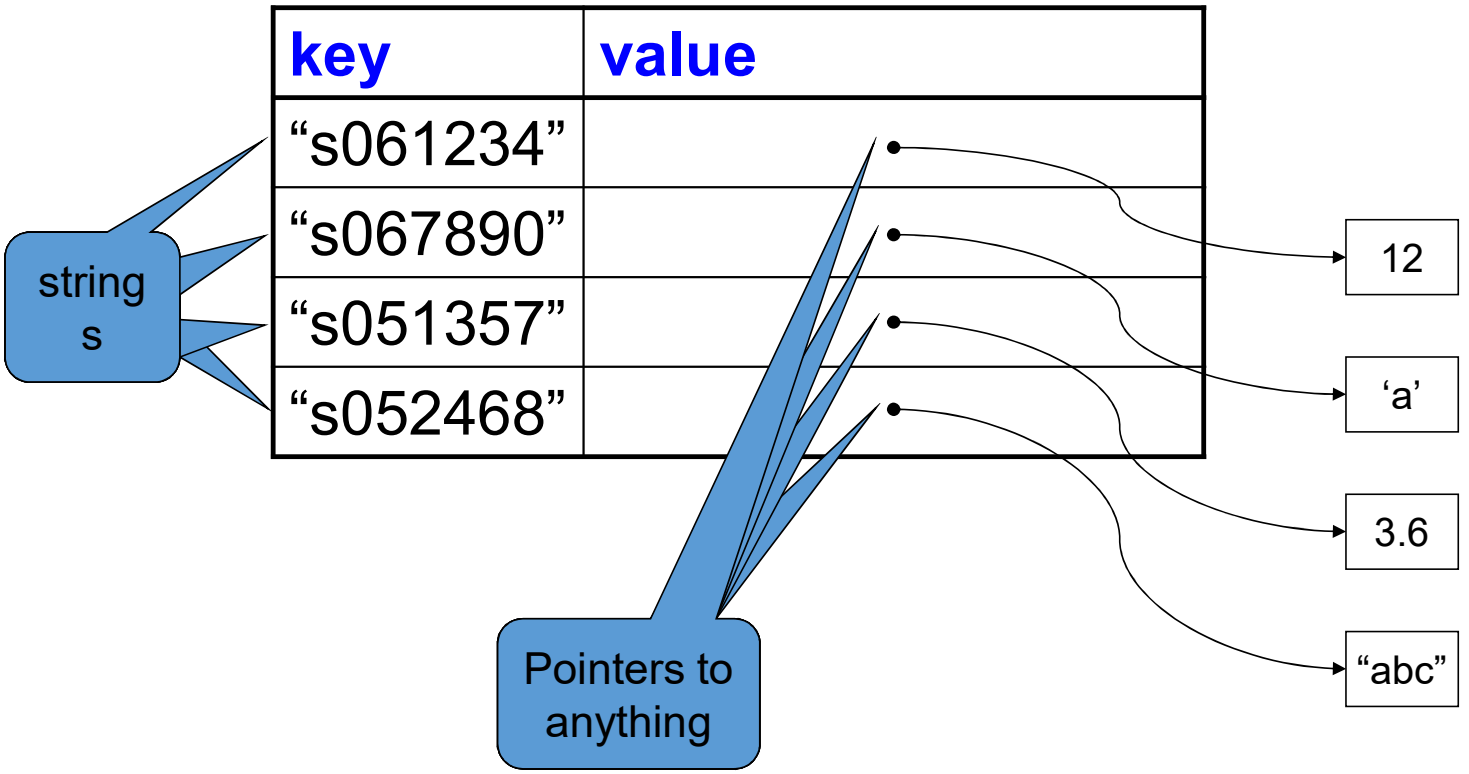
- For simplicity, we define the type of “key” as string.
- (In C, char * is used.)

Generic Pointer

- A generic pointer (type void *) is compatible to pointers of *any* type.

```
double d, *dptr;
int i, *iptr;
void *vptr;

iptr = &i;
dptr = &d;
iptr = dptr;   // invalid
vptr = iptr;   // valid
vptr = dptr;   // valid
iptr = vptr;   // valid, but problematic
```

Defining the hashtable ADT



Defining the hashtable ADT

hashtable.h

```
typedef struct hashtableCDT *hashtableADT;

hashtableADT EmptyHashtable();
void Enter(hashtableADT table, char *key, void *value);
void *Lookup(hashtableADT table, char *key);
```

`hashtableADT EmptyHashtable();`

- Creates and returns a new hashtable with no entries.

Defining the hashtable ADT

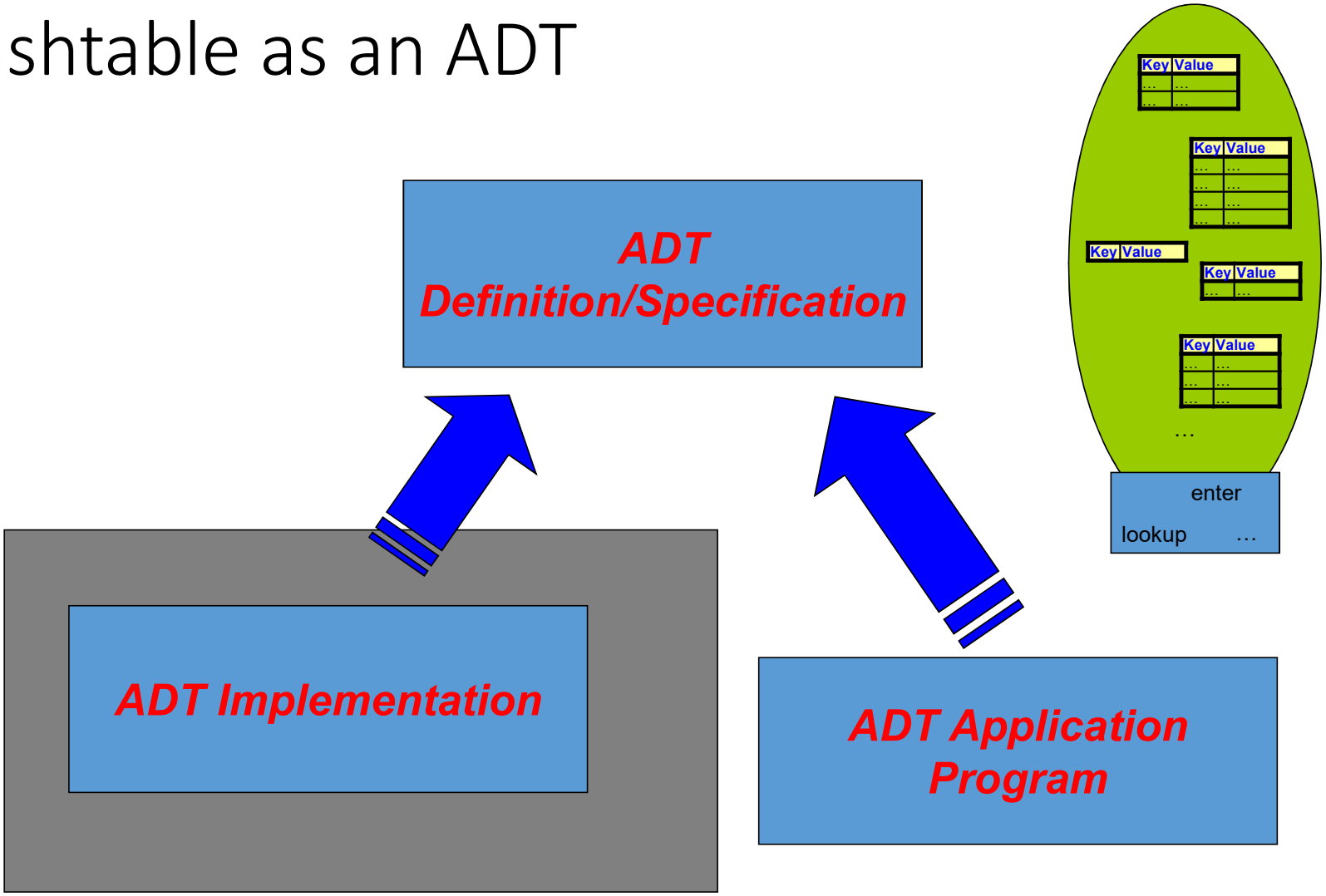
```
void Enter(hashtableADT table,
           char *key, void *value);
```

- Inserts the entry ⟨**key**, **value**⟩ to the hashtable **table**. Nothing is returned.

```
void *Lookup(hashtableADT table, char *key);
```

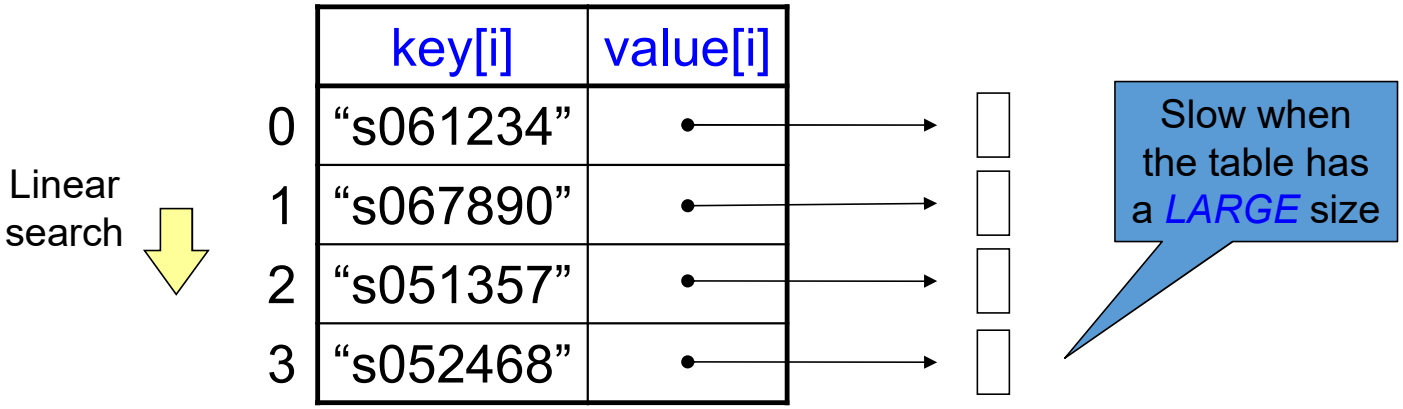
- Looks for and returns the value (which is a pointer) of the key **key** in the hashtable **table**.
- Returns NULL if **key** is not found in **table**.

Hashtable as an ADT



Implementing the Hashtable ADT

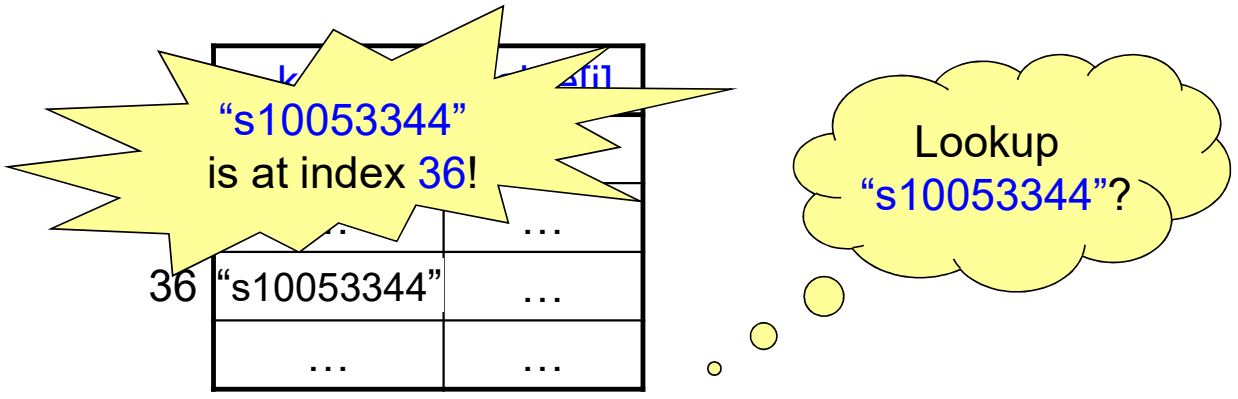
- It is easy to implement a hashtable using arrays.



- Enter and Lookup can be done by searching the array elements one by one.

Implementing the Hashtable ADT

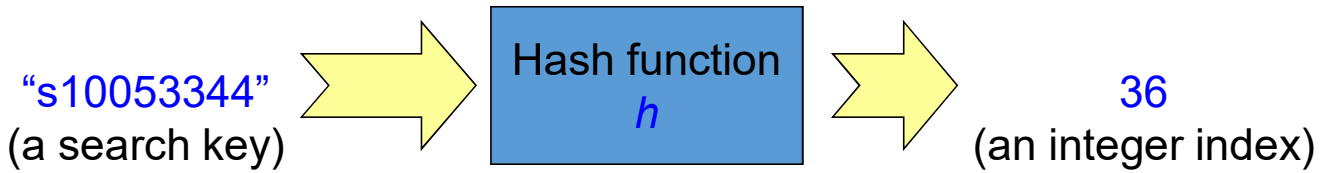
- We would prefer a strategy that allows *efficient* implementation of the operations.
- Ideally, we can know *right away* the array index where a key is positioned.



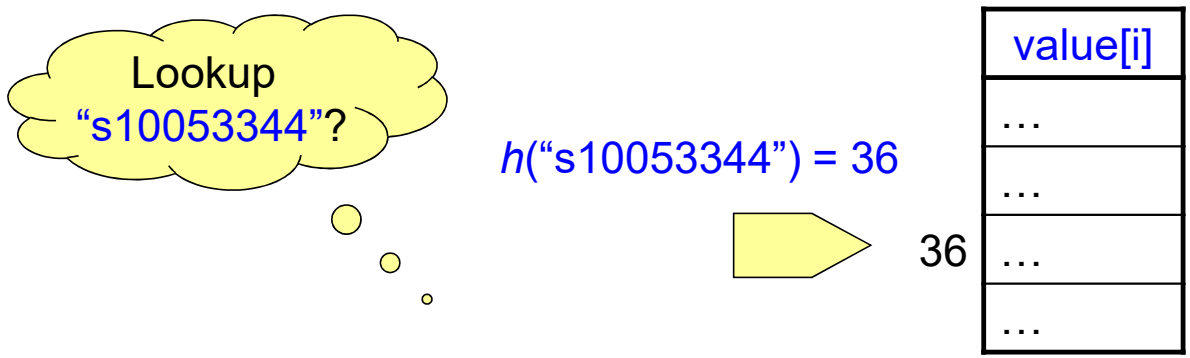
Hashing

- We assume we have a "magic" function *h* that maps a search key to an integer index.
  - E.g., *h*("s10053344") = 36

- Such a magic function is called a ***hash function***.



Hashtable lookup



Collision in Hashing

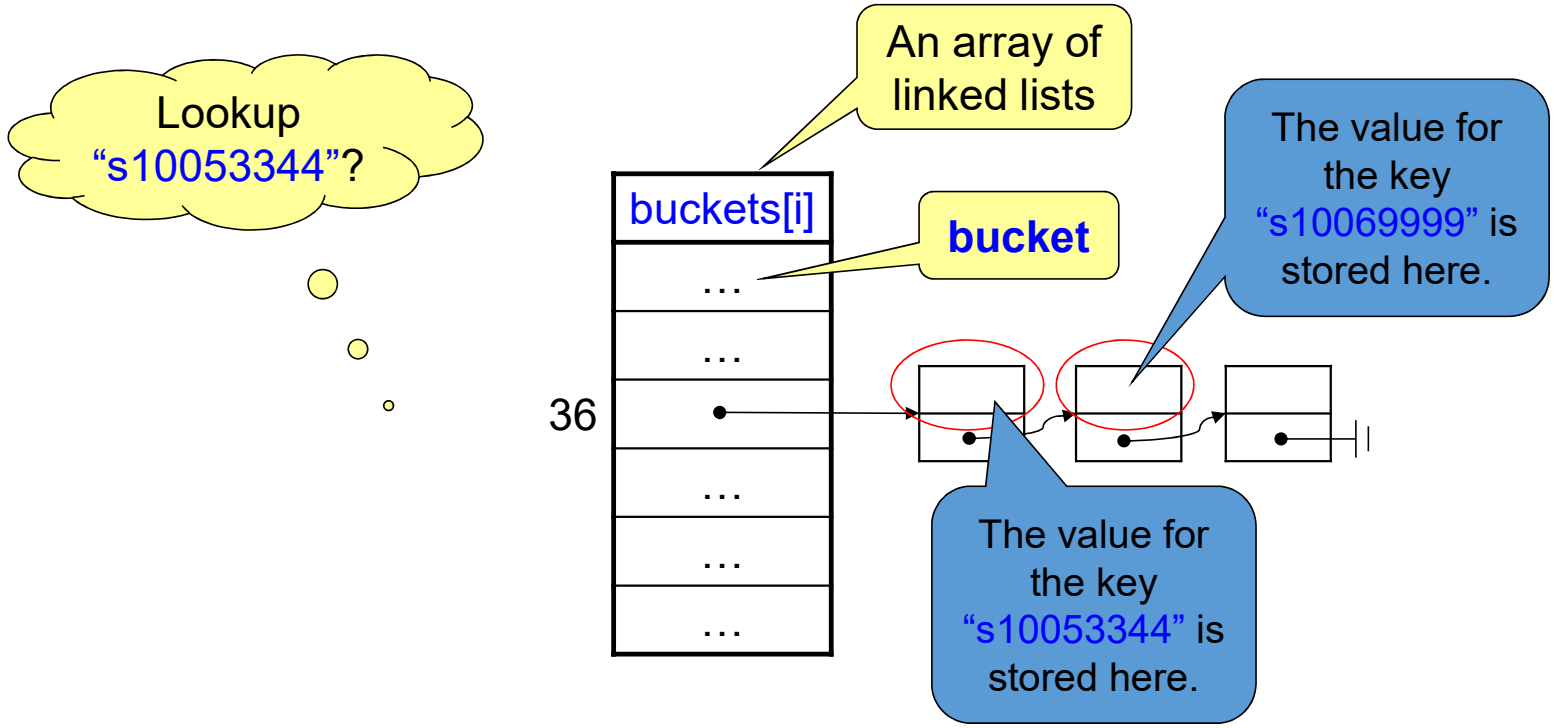
- Perfection rarely exists. A hash function can sometimes return the same index (called ***hash code***) for two different keys.

h("s10053344") = 36
h("s10069999") = 36
…

- Having two or more keys hashed to the same index is called ***collision***.

Chaining

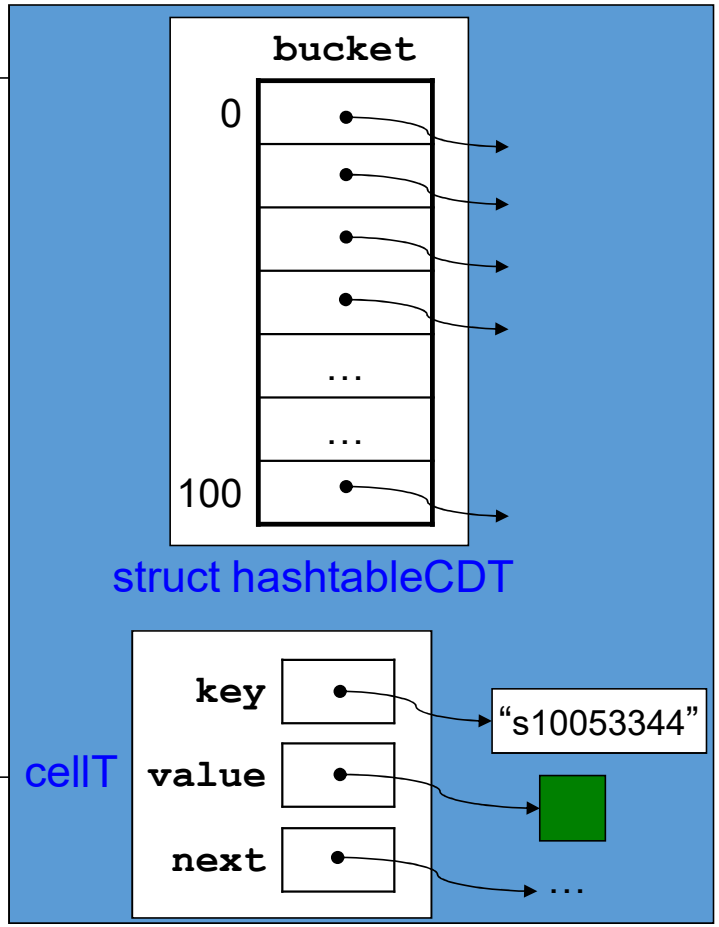
- Collision can be resolved by ***chaining***, i.e., using a linked list for each bucket.



Hashtable *Implementation* (Version *1.0*)

hashtable.c

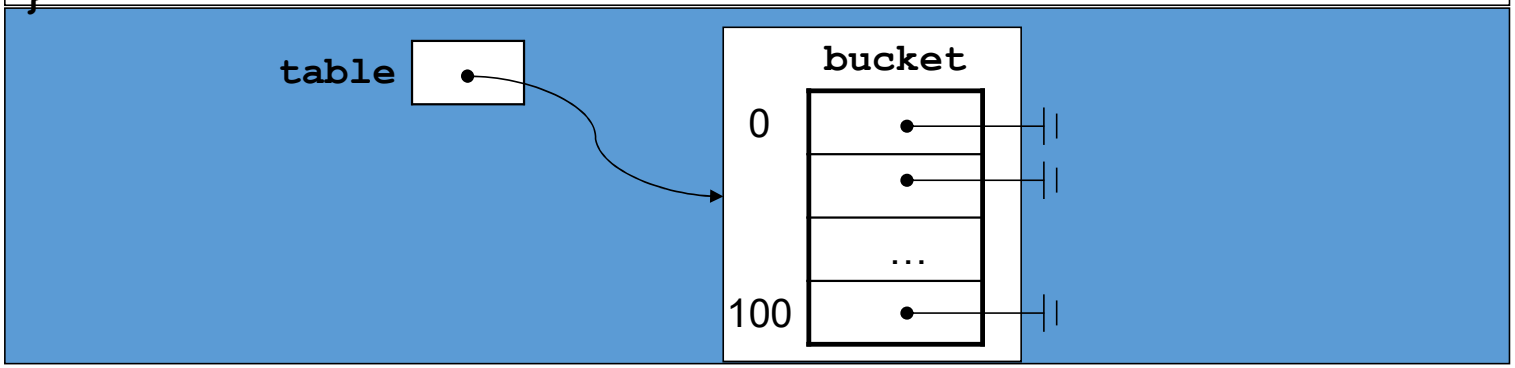
```
#include "hashtable.h"
#include <stdlib.h>
#include <string.h>

typedef struct cellT {
   char *key;
   void *value;
   struct cellT *next;
} cellT;

struct hashtableCDT {
   cellT *buckets[101];
};
```



Hashtable *Implementation* (Version *1.0*)

hashtable.c (continue)

```
hashtableADT EmptyHashtable() {
   int i;
   hashtableADT table;
   table = (hashtableADT)malloc(sizeof(struct
hashtableCDT));
   for (i = 0; i < 101; i++)
      table->buckets[i] = NULL;
   return table;
}
```



Hashtable *Implementation* (Version *1.0*)

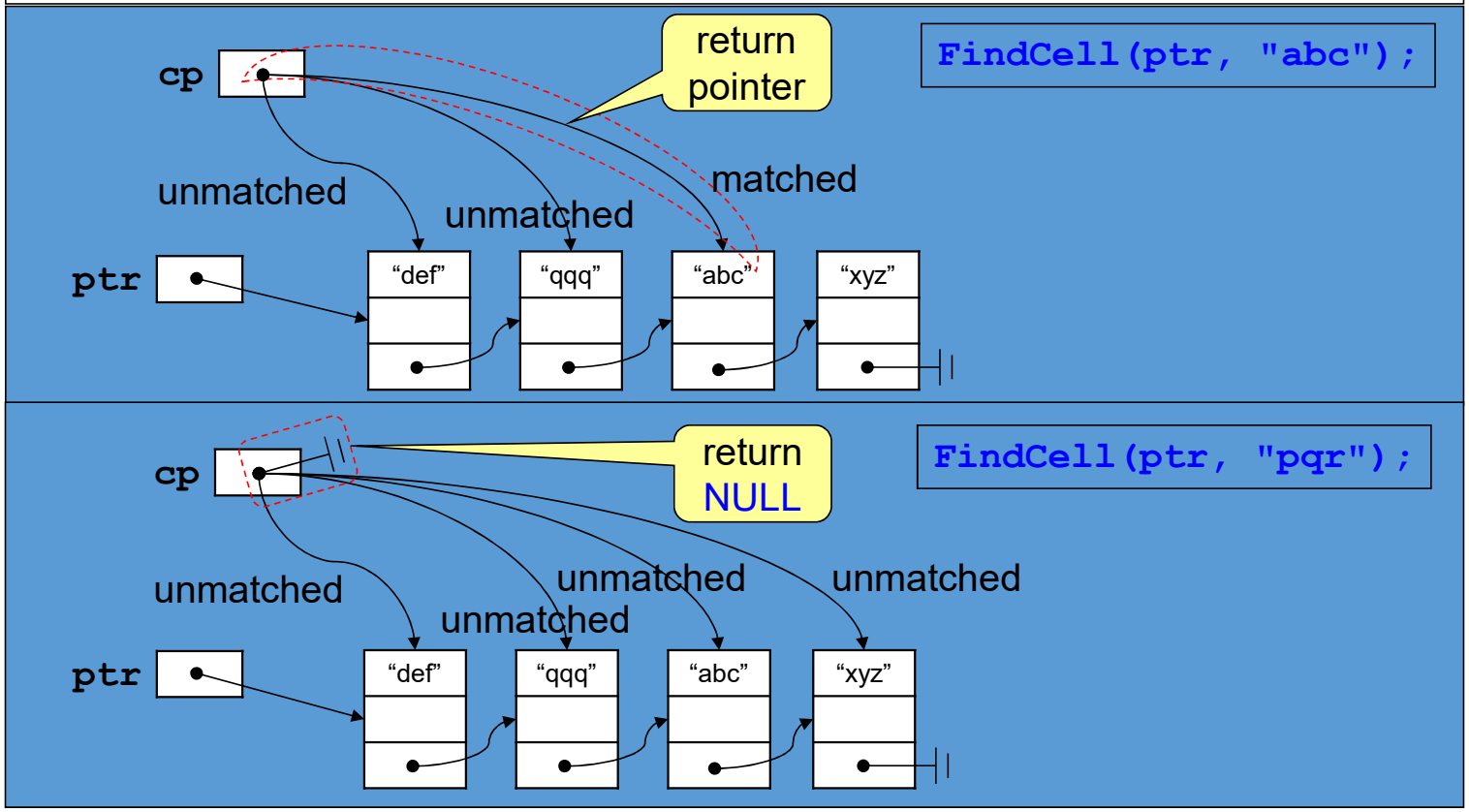
- Before writing the Enter and Lookup functions, we write a helper function FindCell.

```
cellT *FindCell(cellT *cp, char *key);
```

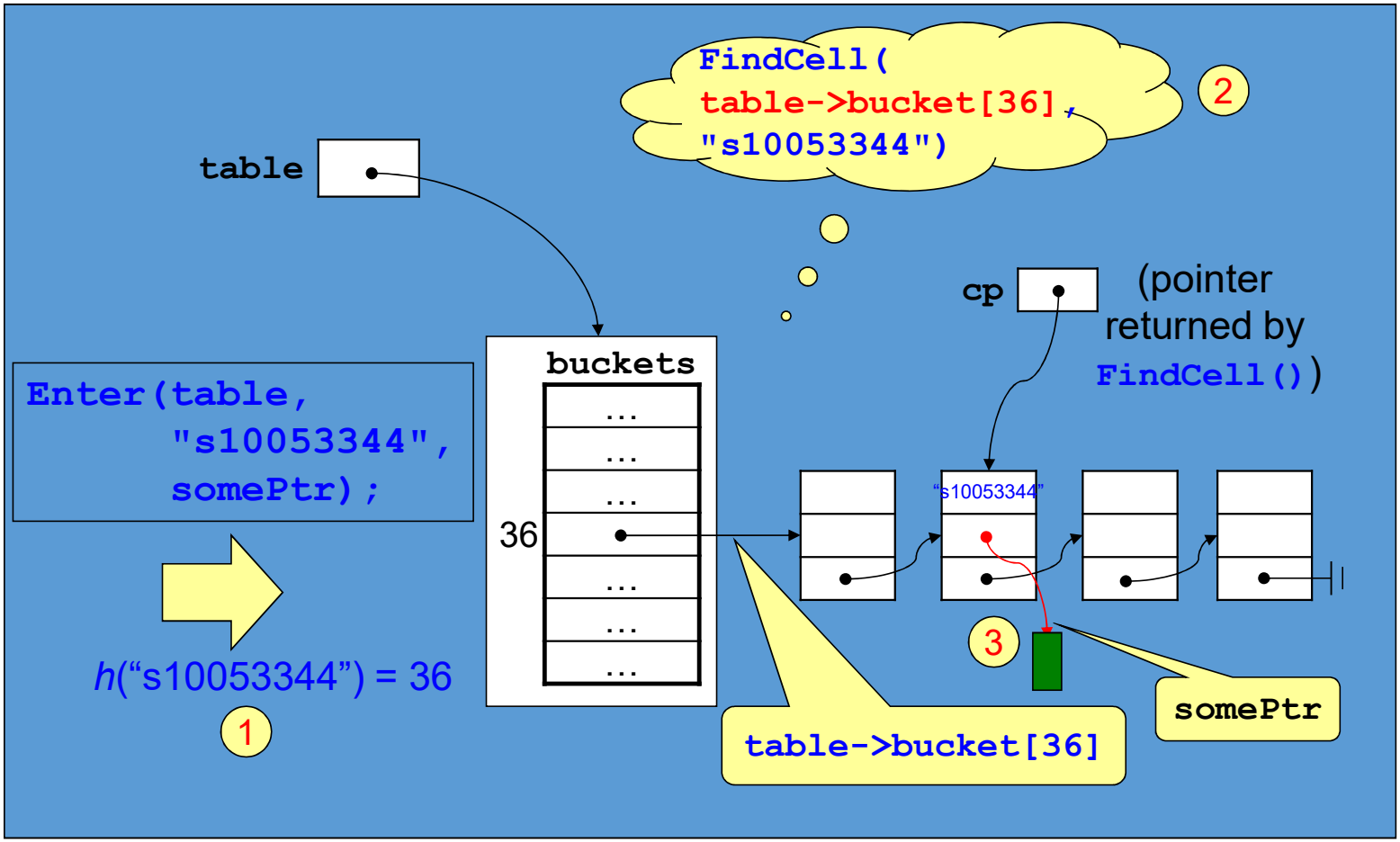
- Traverses the linked list starting from the cell pointed to by **`cp`**.
- Returns a pointer to a cell whose key matches the string pointed to by **`key`**.
- Returns NULL if **`key`** is unmatched for all cells.

hashtable.c (continue)

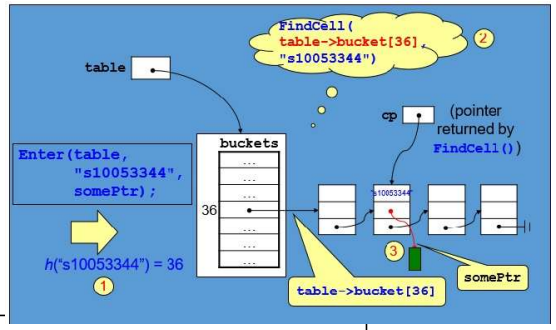
```
cellT *FindCell(cellT *cp, char *key) {
   while (cp != NULL && strcmp(cp->key, key) != 0)
      cp = cp->next;
   return cp;
}
```



The Enter Operation



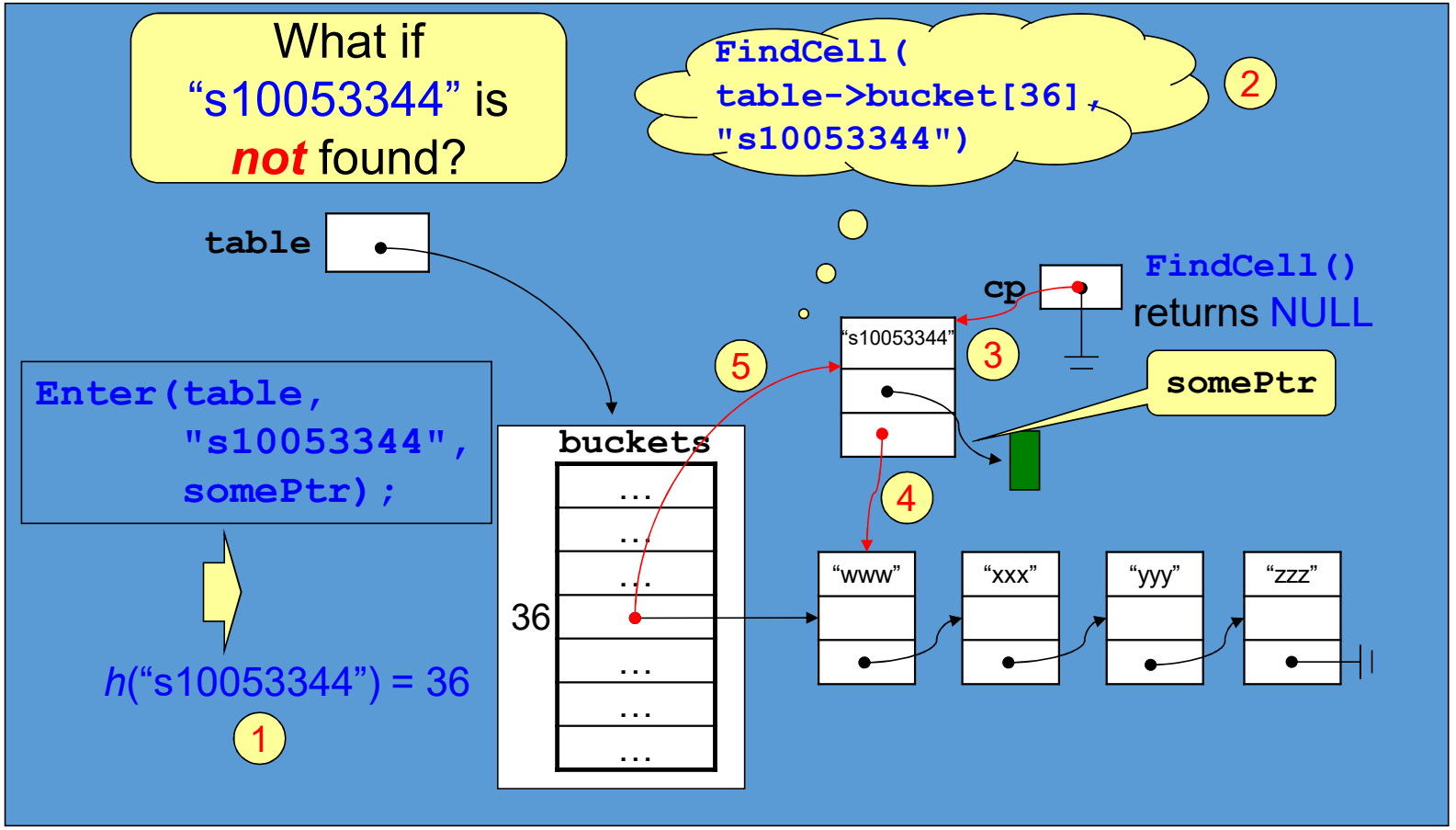
The Enter Operation



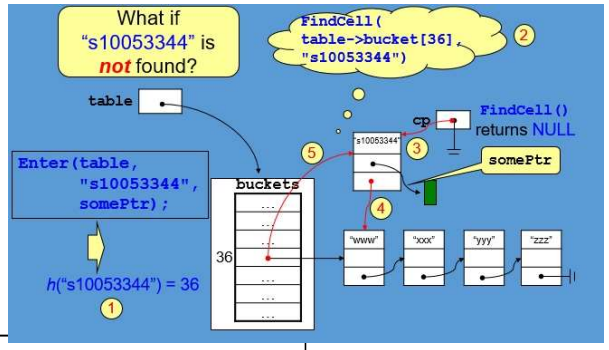
hashtable.c (continue)

```
void Enter(hashtableADT table, char *key, void *value)
{
   cellT *cp;
   int i;
   i = Hash(key, 101);                                    1
   cp = FindCell(table->buckets[i], key);                 2
   if (cp == NULL) {
      cp = (cellT *)malloc(sizeof(cellT));
      cp->key = (char *)malloc((strlen(key) + 1)
                               * sizeof(char));
      strcpy(cp->key, key);
      cp->next = table->buckets[i];
      table->buckets[i] = cp;
   }
   cp->value = value;                                     3
}
```

The Enter Operation



The Enter Operation



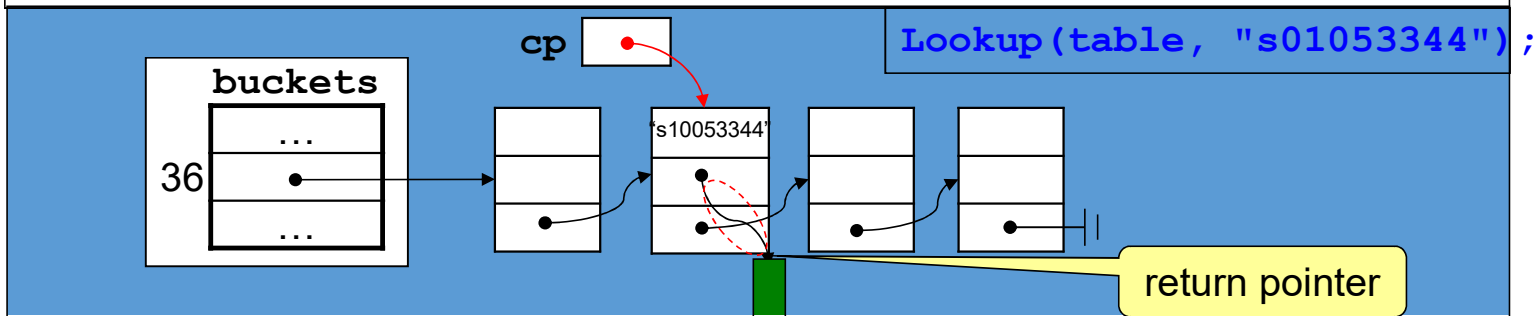
hashtable.c (continue)

```
void Enter(hashtableADT table, char *key, void *value)
{
   cellT *cp;
   int i;
   i = Hash(key, 101);                                    1
   cp = FindCell(table->buckets[i], key);                 2
   if (cp == NULL) {
      cp = (cellT *)malloc(sizeof(cellT));                3
      cp->key = (char *)malloc((strlen(key) + 1)
                               * sizeof(char));
      strcpy(cp->key, key);
      cp->next = table->buckets[i];                       4
      table->buckets[i] = cp;                             5
   }
   cp->value = value;
}
```

The Lookup Operation

hashtable.c (continue)

```
void *Lookup(hashtableADT table, char *key) {
   cellT *cp;
   int i;

   i = Hash(key, 101);
   cp = FindCell(table->buckets[i], key);
   if (cp == NULL)    // unmatched key
      return NULL;    // return NULL for unmatched key
   return cp->value;    // key matched, return the value
}
```



Choosing a Hash Function

- The final component is the implementation of the magic hash function.

hashtable.c (continue)

```
#define MULTIPLIER -16644117991L

int Hash(char *s, int nBuckets) {
   int i;
   unsigned long hashCode;

   hashCode = 0;
   for (i = 0; s[i] != '\0'; i++)
      hashCode = hashCode * MULTIPLIER + s[i];
   return (hashCode % nBuckets);
}
```

Choosing a Hash Function

- This is typical of the functions most often used in commercial practice.

i	0	1	2	3	4	5	6	7	8	9
s[i]	‘s’	‘1’	‘0’	‘0’	‘5’	‘3’	‘3’	‘4’	‘4’	‘\0’
ASCII	115	49	48	48	53	51	51	52	52	

```
hashCode = 0;
hashCode = hashCode * MULTIPLIER + 115;
hashCode = hashCode * MULTIPLIER + 49;
hashCode = hashCode * MULTIPLIER + 48;
hashCode = hashCode * MULTIPLIER + 48;
hashCode = hashCode * MULTIPLIER + 53;
hashCode = hashCode * MULTIPLIER + 51;
hashCode = hashCode * MULTIPLIER + 51;
hashCode = hashCode * MULTIPLIER + 52;
hashCode = hashCode * MULTIPLIER + 52;
return (hashCode % 101);
```

Choosing a Hash Function

A *mysterious* constant (too advanced to explain)

hashtable.c (continue)

```
#define MULTIPLIER -1644117991L

int Hash(char *s, int nBuckets) {
   int i;
   unsigned long hashCode;

   hashCode = 0;
   for (i = 0; s[i] != '\0'; i++)
      hashCode = hashCode * MULTIPLIER + s[i];
   return (hashCode % nBuckets);
}
```

Returns an integer in the range [0, nBuckets - 1].