Data Structures and Algorithms

- Hashing -

School of Electrical Engineering
Korea University

Hashing

- O(1) search by hash-table lookup
- A hash function $\boldsymbol{h}(\boldsymbol{x})$ 정의
  - Input key $\boldsymbol{x}$에 대해 cell 주소 값을 계산
  - 충돌(collision)을 최소화하도록 선택

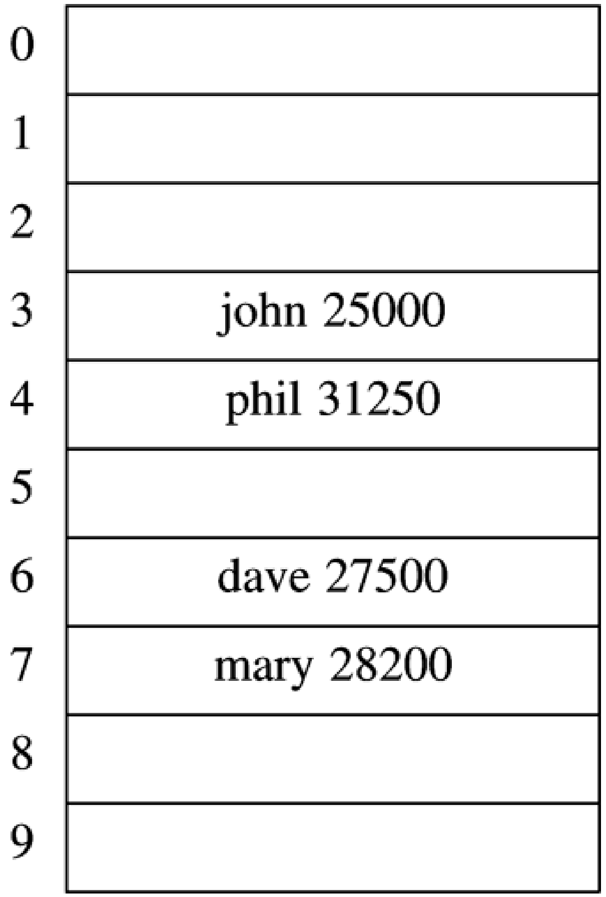
$$X = F(x)$$

$$Y = F(y)$$

$$\text{as long as } x \neq y \text{ then X} \neq \text{Y in most of the times/cases}$$

  - Hash function requirements: uniform distribution, same computing times, generate few collisions

Ideal hash table



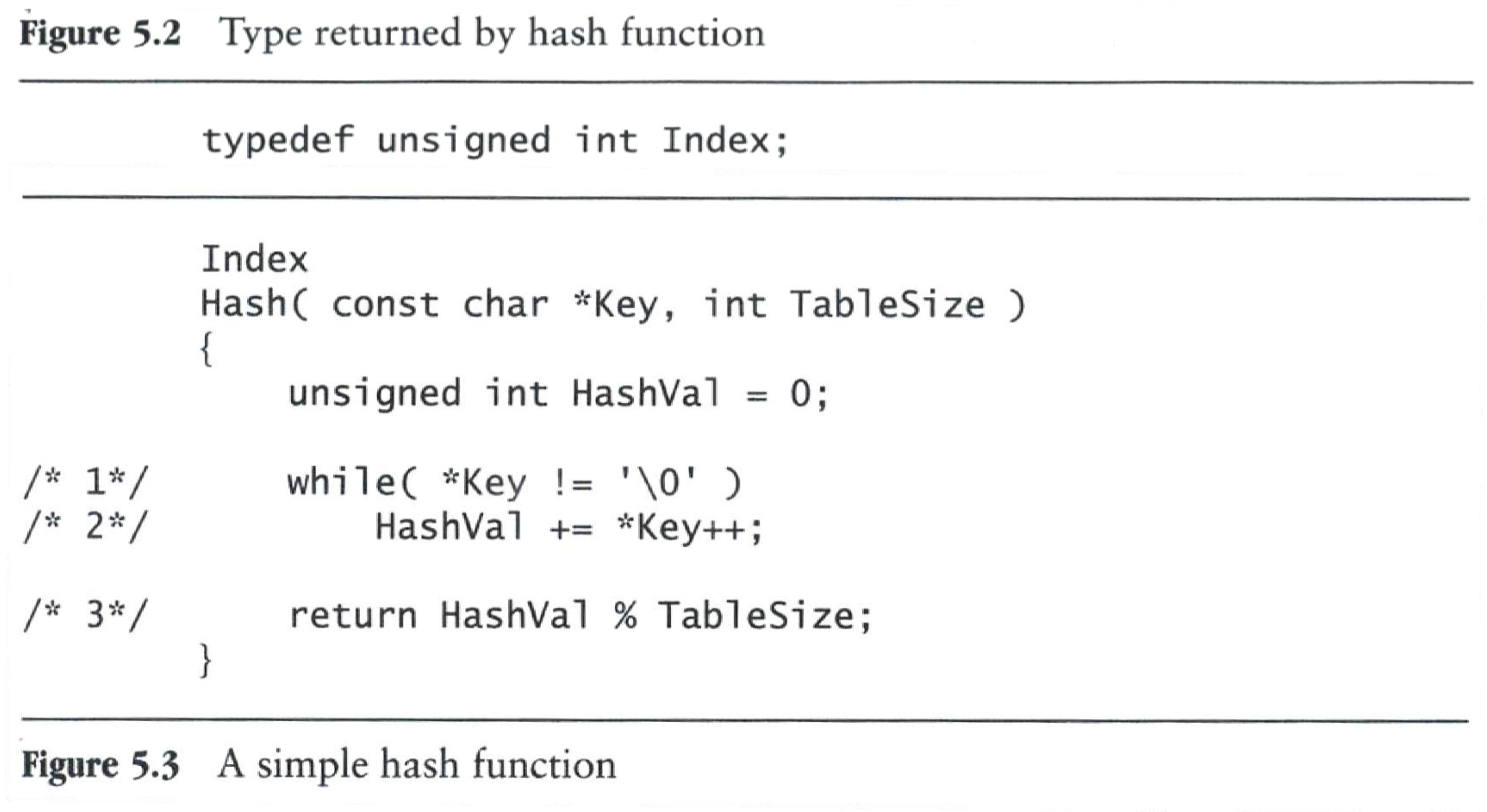


Figure 5.2 Type returned by hash function

```
        typedef unsigned int Index;
```

```
        Index
        Hash( const char *Key, int TableSize )
        {
            unsigned int HashVal = 0;

/* 1*/      while( *Key != '\0' )
/* 2*/          HashVal += *Key++;

/* 3*/      return HashVal % TableSize;
        }
```

Figure 5.3 A simple hash function

// add up the ASCII values of the characters

// Not good when table size is big and hash value range is too small

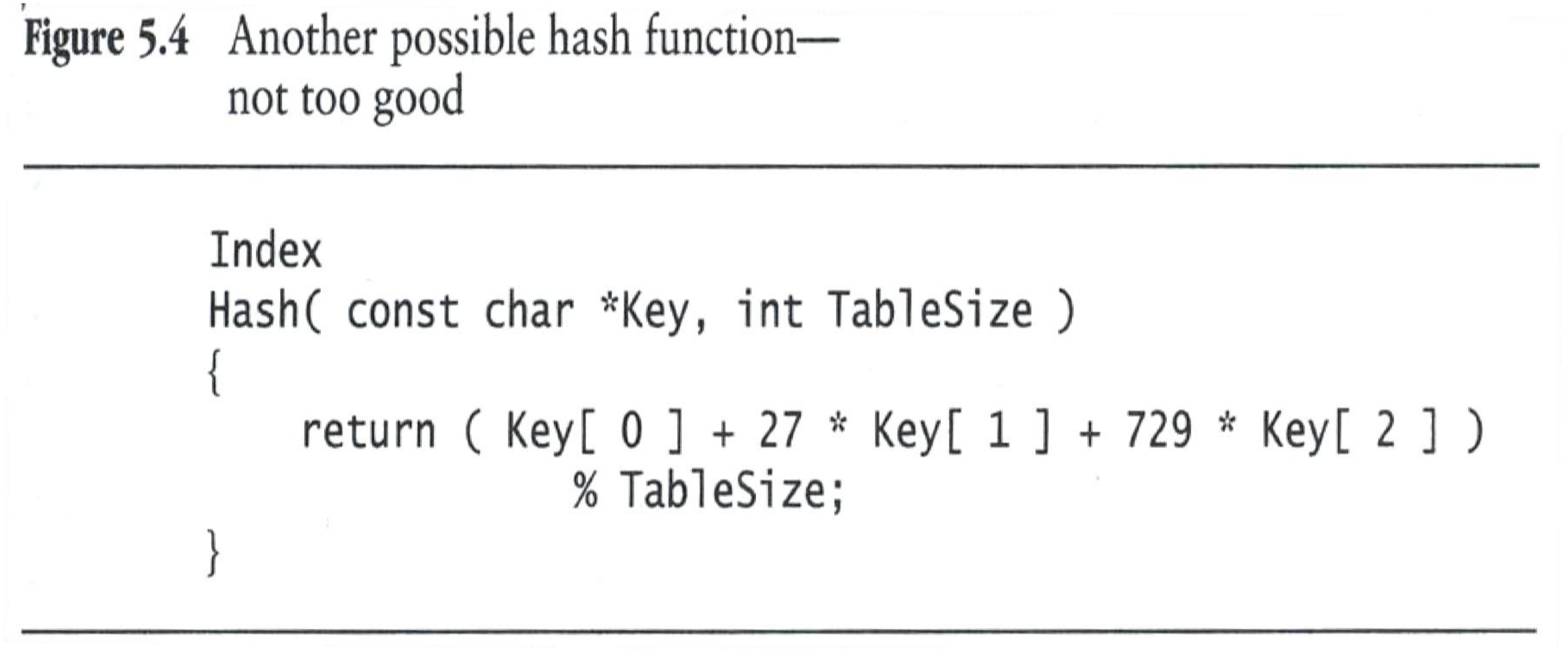
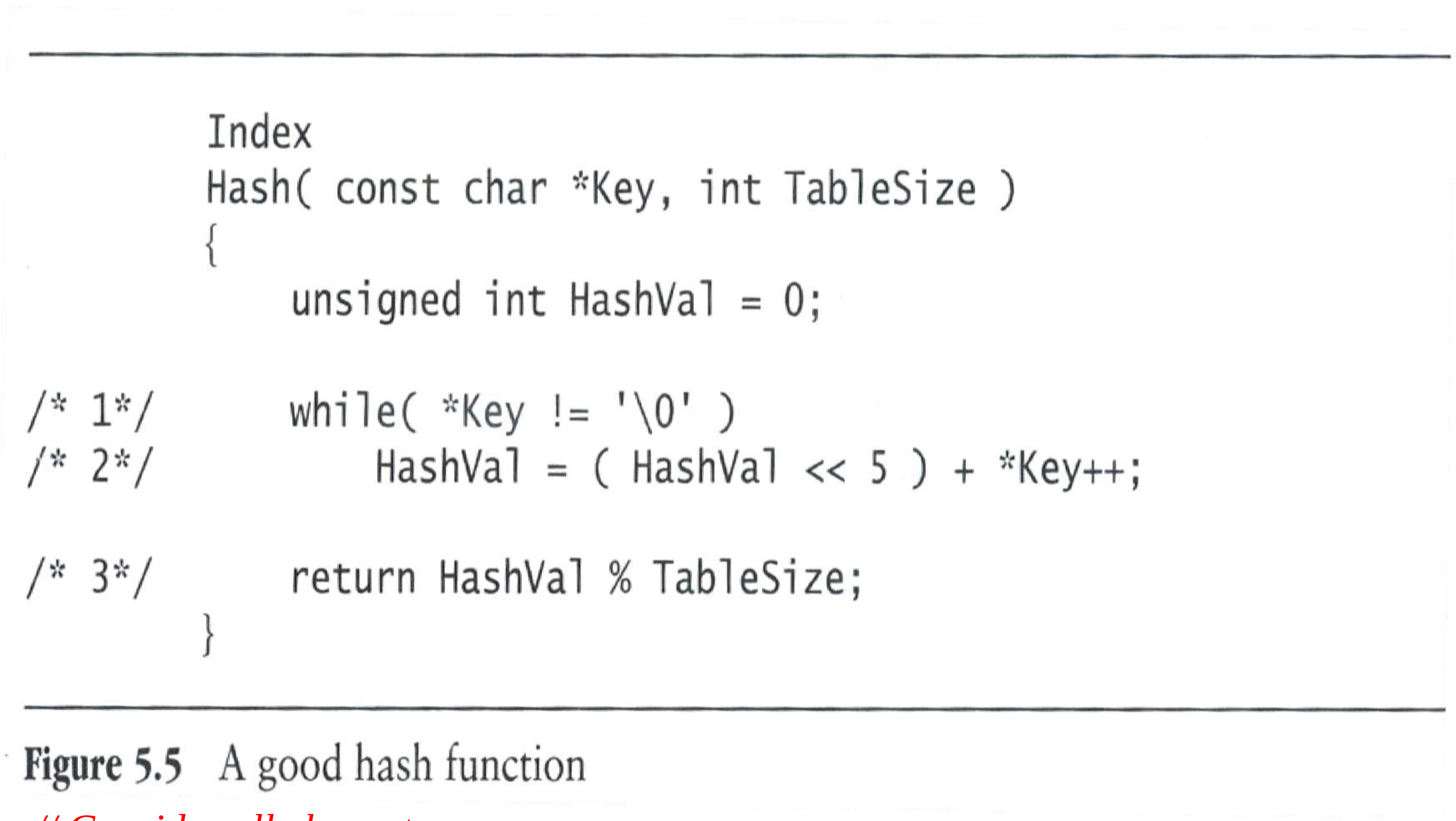


Figure 5.4 Another possible hash function—
not too good

```
Index
Hash( const char *Key, int TableSize )
{
    return ( Key[ 0 ] + 27 * Key[ 1 ] + 729 * Key[ 2 ] )
                % TableSize;
}
```

// add up the weighted ASCII values of the first three characters
*//For three characters, 26*26*26 = 17,576 possible combinations*
// , but only 2,851 combinations in reality.



```
Index
Hash( const char *Key, int TableSize )
{
    unsigned int HashVal = 0;

    while( *Key != '\0' )
        HashVal = ( HashVal << 5 ) + *Key++;

    return HashVal % TableSize;
}
```

Figure 5.5 A good hash function

// Consider all characters

// use 32 instead of 27 for bit-shifting operation

Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Ex:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys

- The integer $h(x)$ is called the hash value of key x
- The goal of a hash function is to uniformly disperse keys in the range $[0, N - 1]$

Hash Functions and Hash Tables

- A hash table for a given key type consists of
  - Hash function *h* and Array (table) size *N*
- When implementing a dictionary with a hash table, the goal is to store item
  (*k*, *o*) at index $i = h(k)$
- A collision occurs when two keys in the dictionary have the same hash value
- Collision handing schemes:
  - Chaining: colliding items in the list
  - Open addressing: colliding item in a different cell of the table

Design options of hashing

- load factor $\lambda < 1.0$
- table size S – choose a prime number
- Data store – array/ list
- hash function F
- alternative location on collision:
  - chaining
  - open addressing

Collision resolution

- Separate chaining
  - add into the linear list on collision
  - search time is not constant .
- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
- Rehashing
  - Doubling the table size to $2N$
  - cost of rehashing old data: $O(N)$

Separate chaining

- Add the key into the linear list on collision
- Search time is dependent on the length of the list.
- Find operation
- Insert opetation

Example of separate chaining

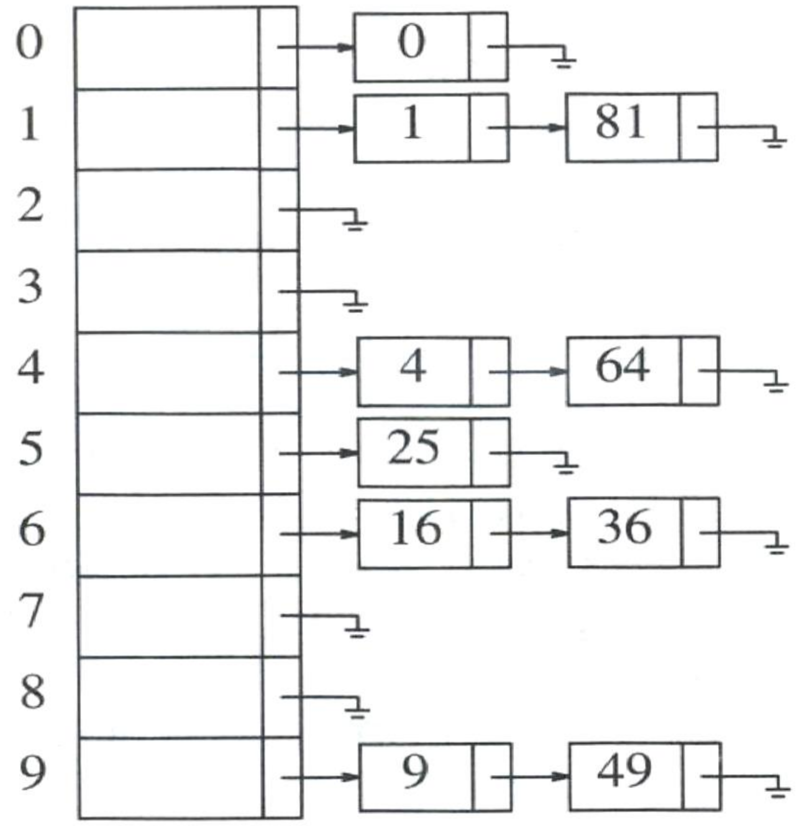
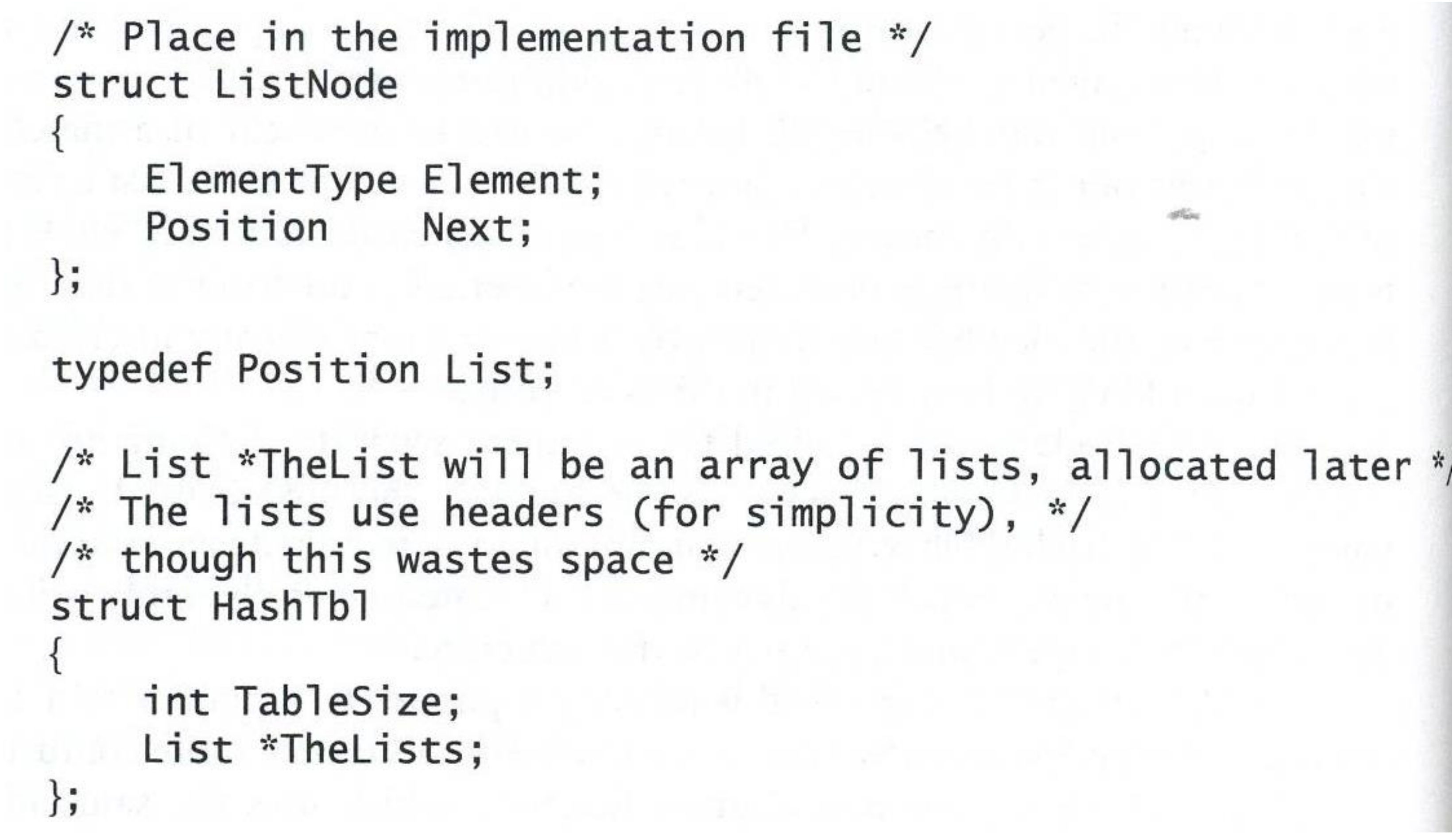


Figure 5.6 A separate chaining hash table

Type declaration

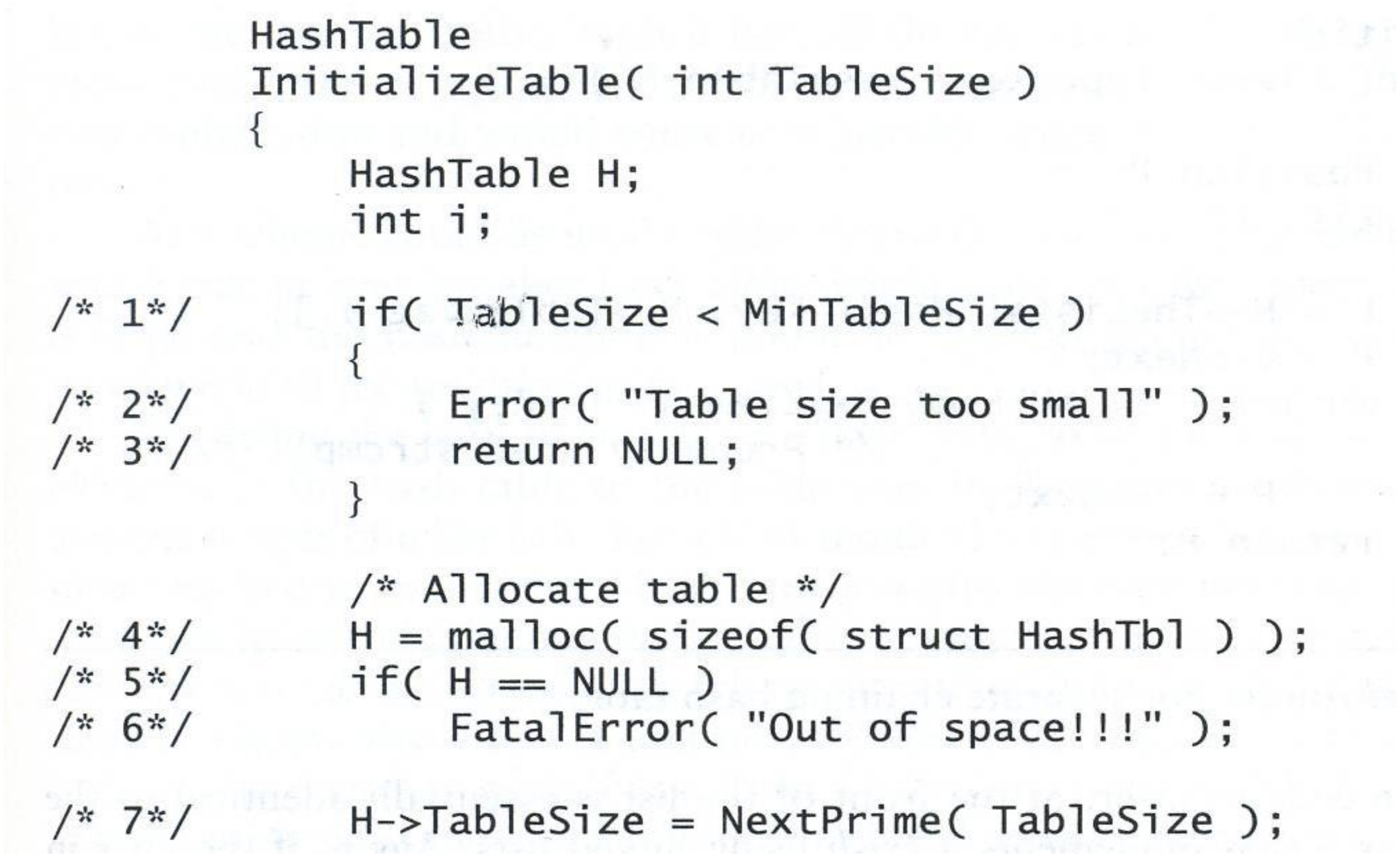
```
#ifndef _HashSep_H

struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable( int TableSize );
void DestroyTable( HashTable H );
Position Find( ElementType Key, HashTable H );
void Insert( ElementType Key, HashTable H );
ElementType Retrieve( Position P );
/* Routines such as Delete and MakeEmpty are omitted */

#endif  /* _HashSep_H */
```

Type declaration



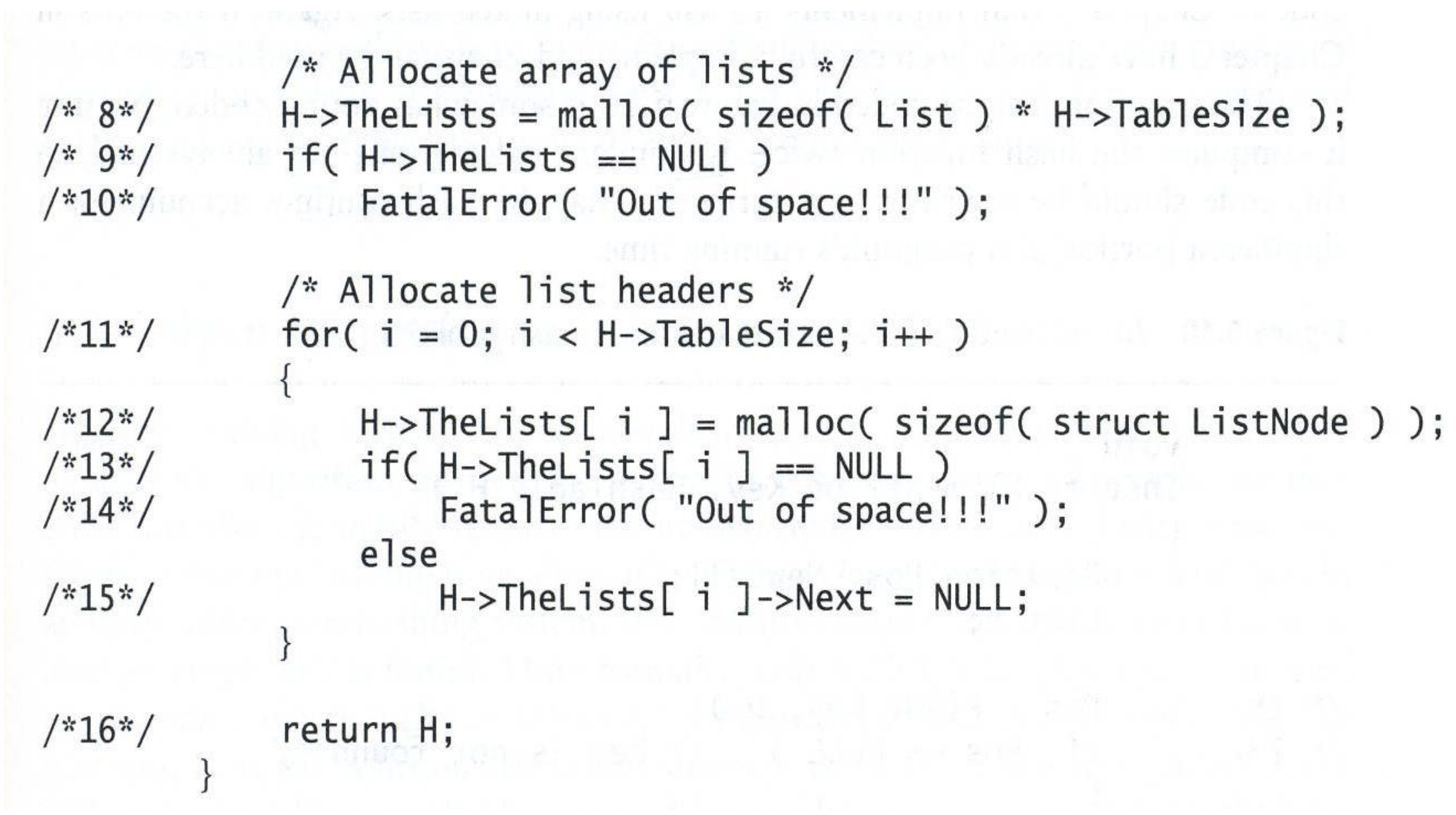
```
/* Place in the implementation file */
struct ListNode
{
    ElementType Element;
    Position    Next;
};

typedef Position List;

/* List *TheList will be an array of lists, allocated later */
/* The lists use headers (for simplicity), */
/* though this wastes space */
struct HashTbl
{
    int TableSize;
    List *TheLists;
};
```

Initialization routine



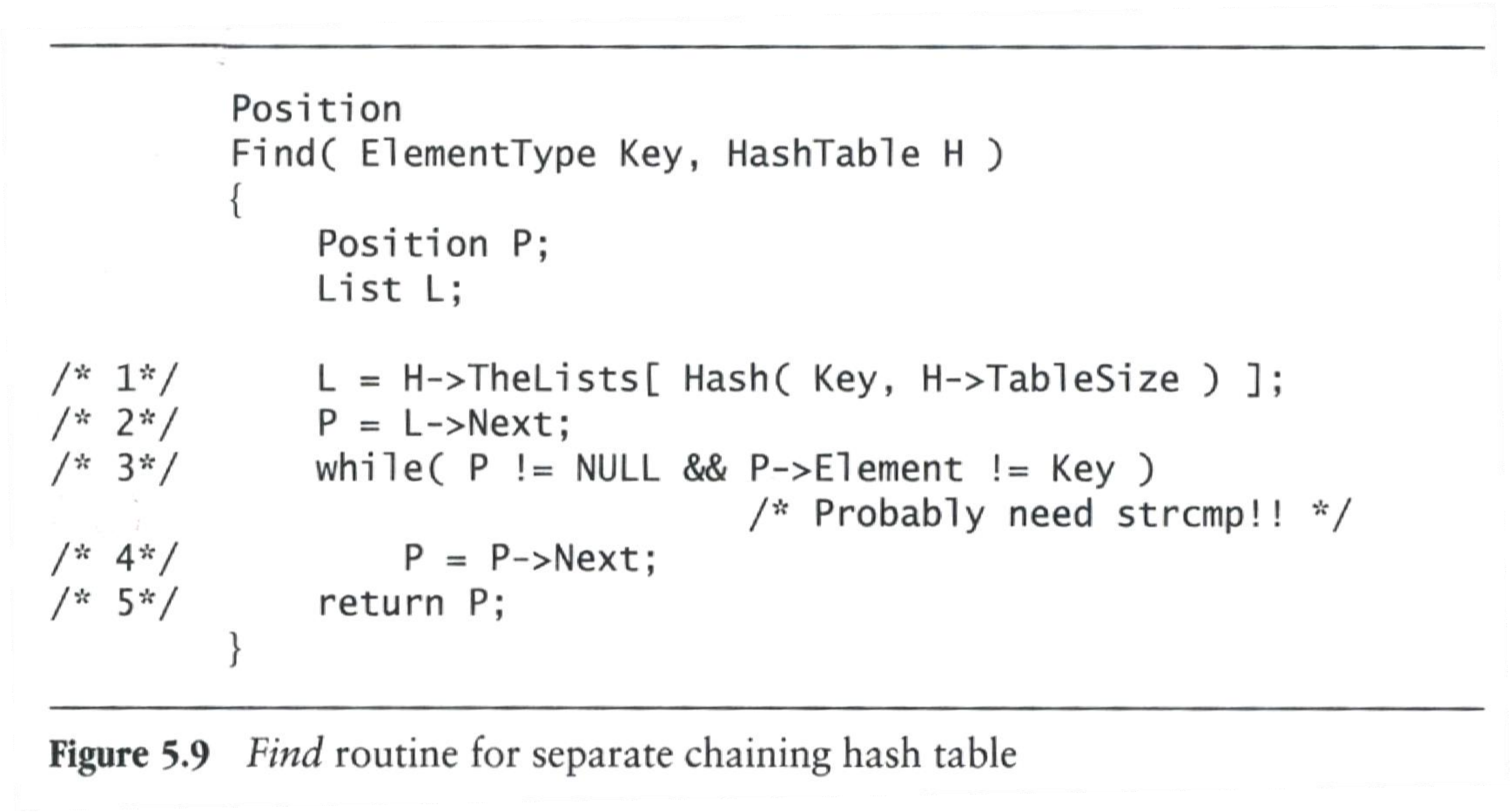
```
HashTable
InitializeTable( int TableSize )
{
    HashTable H;
    int i;

    if( TableSize < MinTableSize )
    {
        Error( "Table size too small" );
        return NULL;
    }

    /* Allocate table */
    H = malloc( sizeof( struct HashTbl ) );
    if( H == NULL )
        FatalError( "Out of space!!!" );

    H->TableSize = NextPrime( TableSize );
```

Initialization routine



```
        /* Allocate array of lists */
        H->TheLists = malloc( sizeof( List ) * H->TableSize );
        if( H->TheLists == NULL )
            FatalError( "Out of space!!!" );

        /* Allocate list headers */
        for( i = 0; i < H->TableSize; i++ )
        {
            H->TheLists[ i ] = malloc( sizeof( struct ListNode ) );
            if( H->TheLists[ i ] == NULL )
                FatalError( "Out of space!!!" );
            else
                H->TheLists[ i ]->Next = NULL;
        }

        return H;
    }
```

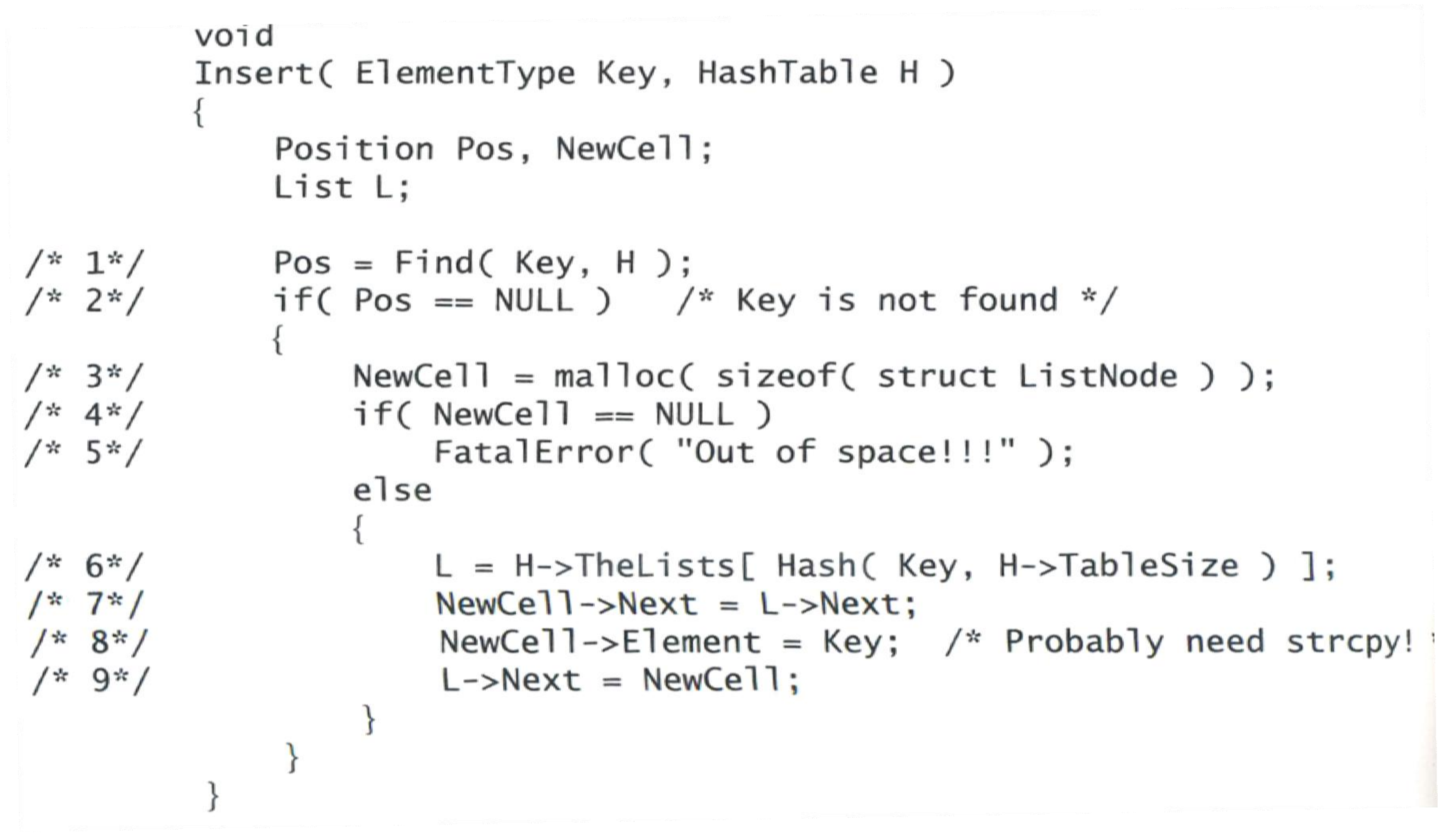
Find routine



```
Position
Find( ElementType Key, HashTable H )
{
    Position P;
    List L;

    L = H->TheLists[ Hash( Key, H->TableSize ) ];
    P = L->Next;
    while( P != NULL && P->Element != Key )
                              /* Probably need strcmp!! */
        P = P->Next;
    return P;
}
```

Figure 5.9 *Find* routine for separate chaining hash table

Insert routine



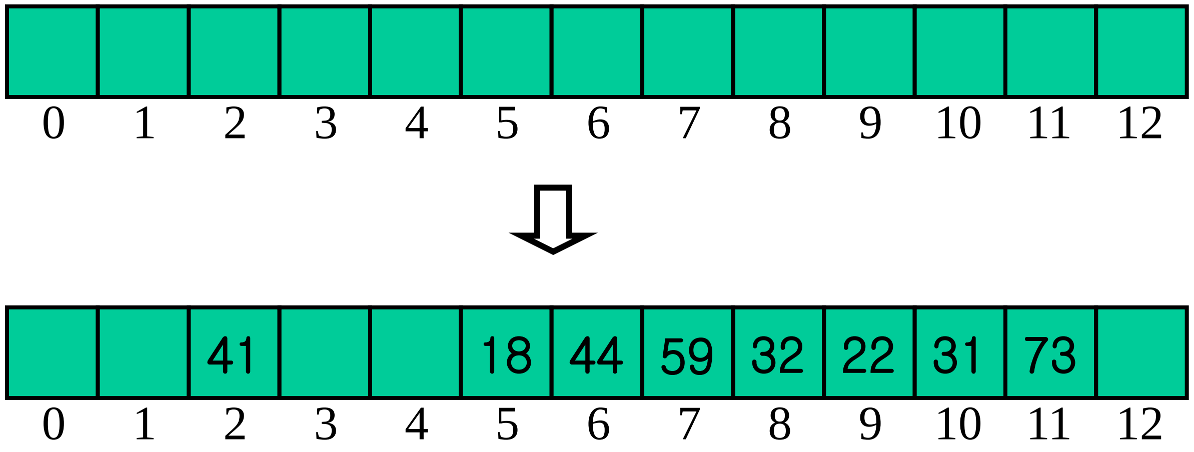
```
void
Insert( ElementType Key, HashTable H )
{
    Position Pos, NewCell;
    List L;

    Pos = Find( Key, H );
    if( Pos == NULL )   /* Key is not found */
    {
        NewCell = malloc( sizeof( struct ListNode ) );
        if( NewCell == NULL )
            FatalError( "Out of space!!!" );
        else
        {
            L = H->TheLists[ Hash( Key, H->TableSize ) ];
            NewCell->Next = L->Next;
            NewCell->Element = Key;  /* Probably need strcpy!
            L->Next = NewCell;
        }
    }
}
```

Linear Probing

- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a “probe”
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Linear Probing

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Open addressing hash table with linear probing, after each insertion

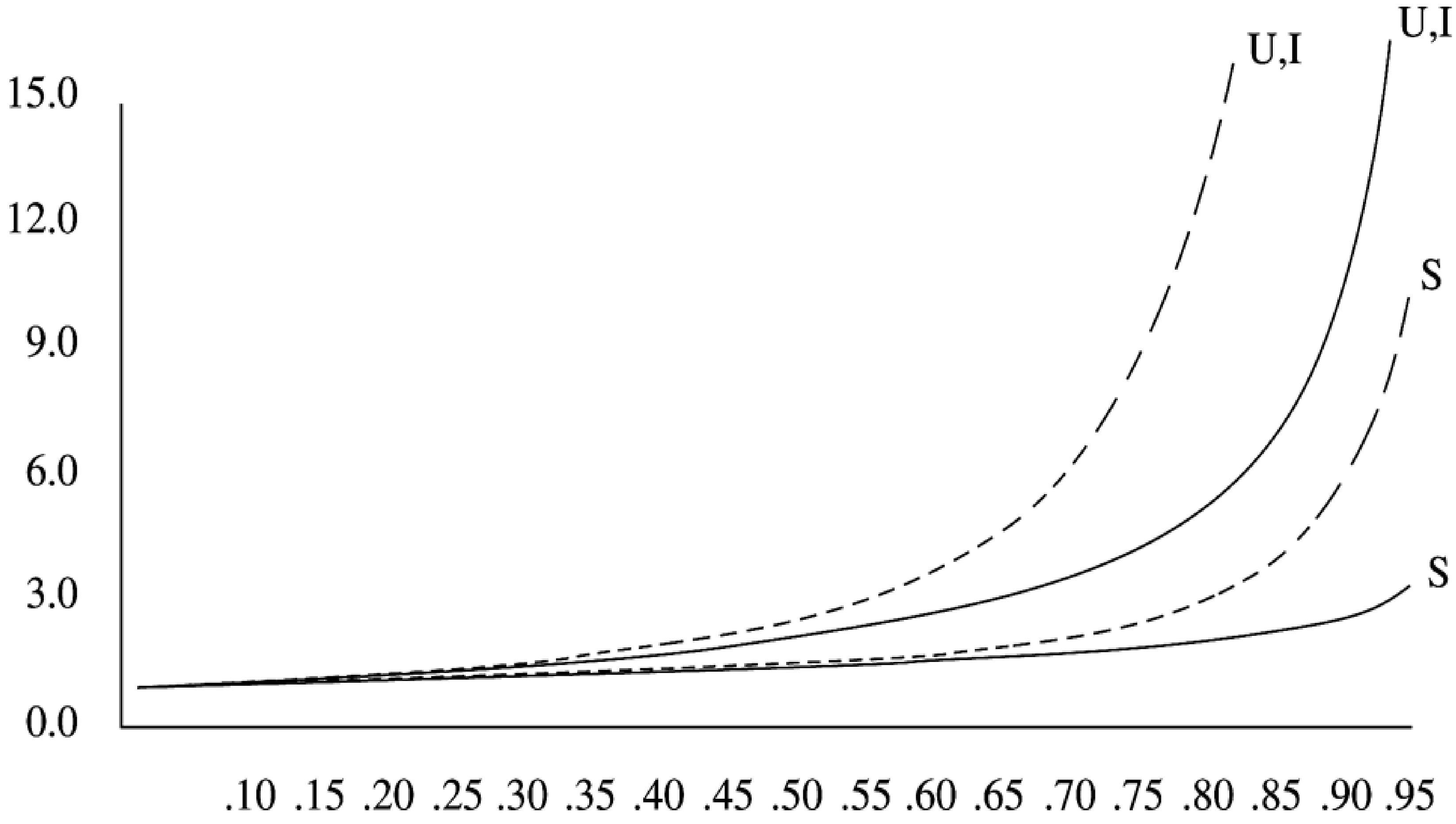


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (S is successful search, U is unsuccessful search, and I is insertion)

89, 18, 49, 58, 69

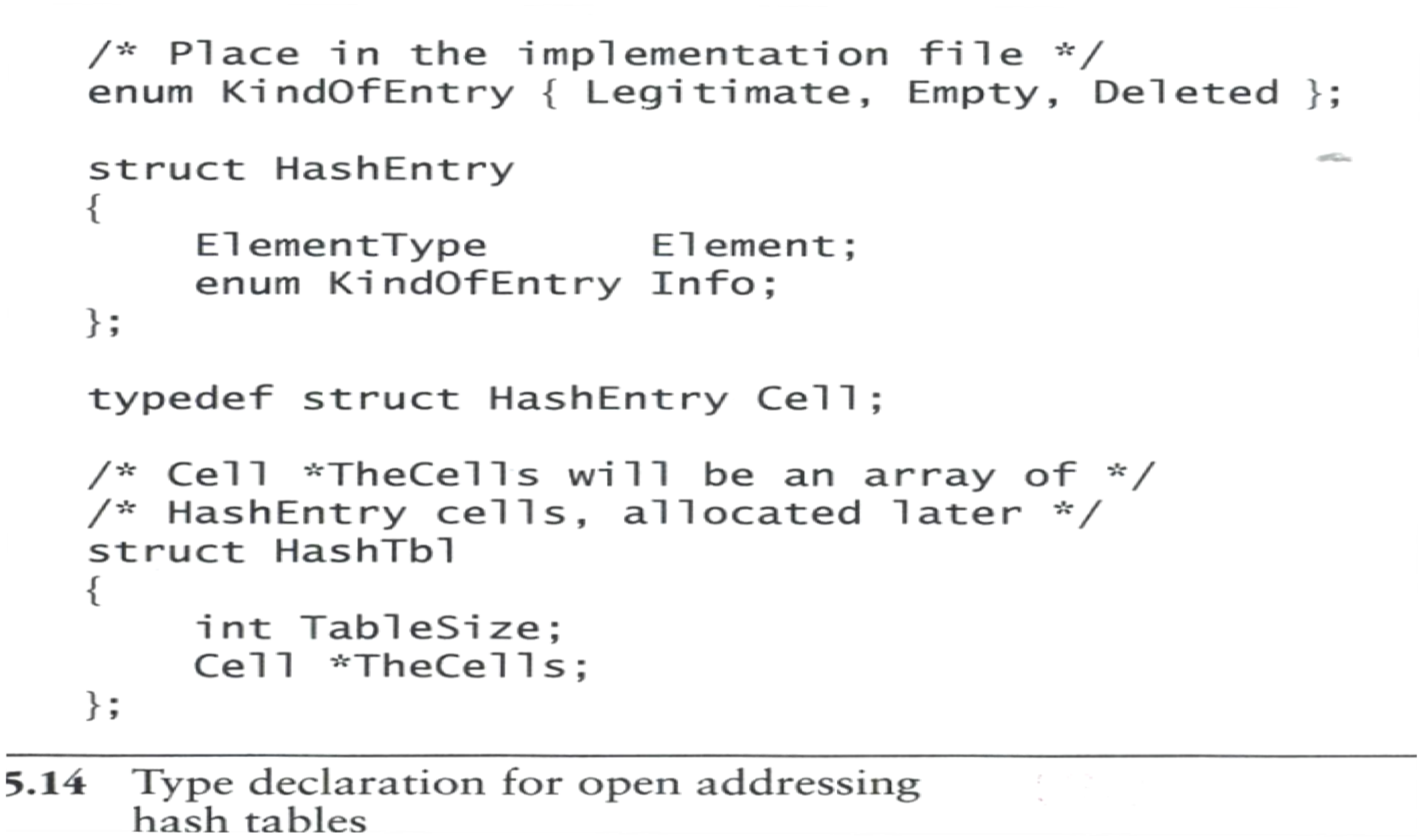
	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.13 Open addressing hash table with quadric probing, after each insertion

Type declaration for open addressing

```
#ifndef _HashQuad_H

typedef unsigned int Index;
typedef Index Position;

struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable( int TableSize );
void DestroyTable( HashTable H );
Position Find( ElementType Key, HashTable H );
void Insert( ElementType Key, HashTable H );
ElementType Retrieve( Position P, HashTable H );
HashTable Rehash( HashTable H );
/* Routines such as Delete and MakeEmpty are omitted */

#endif  /* _HashQuad_H */
```

Type declaration for open addressing



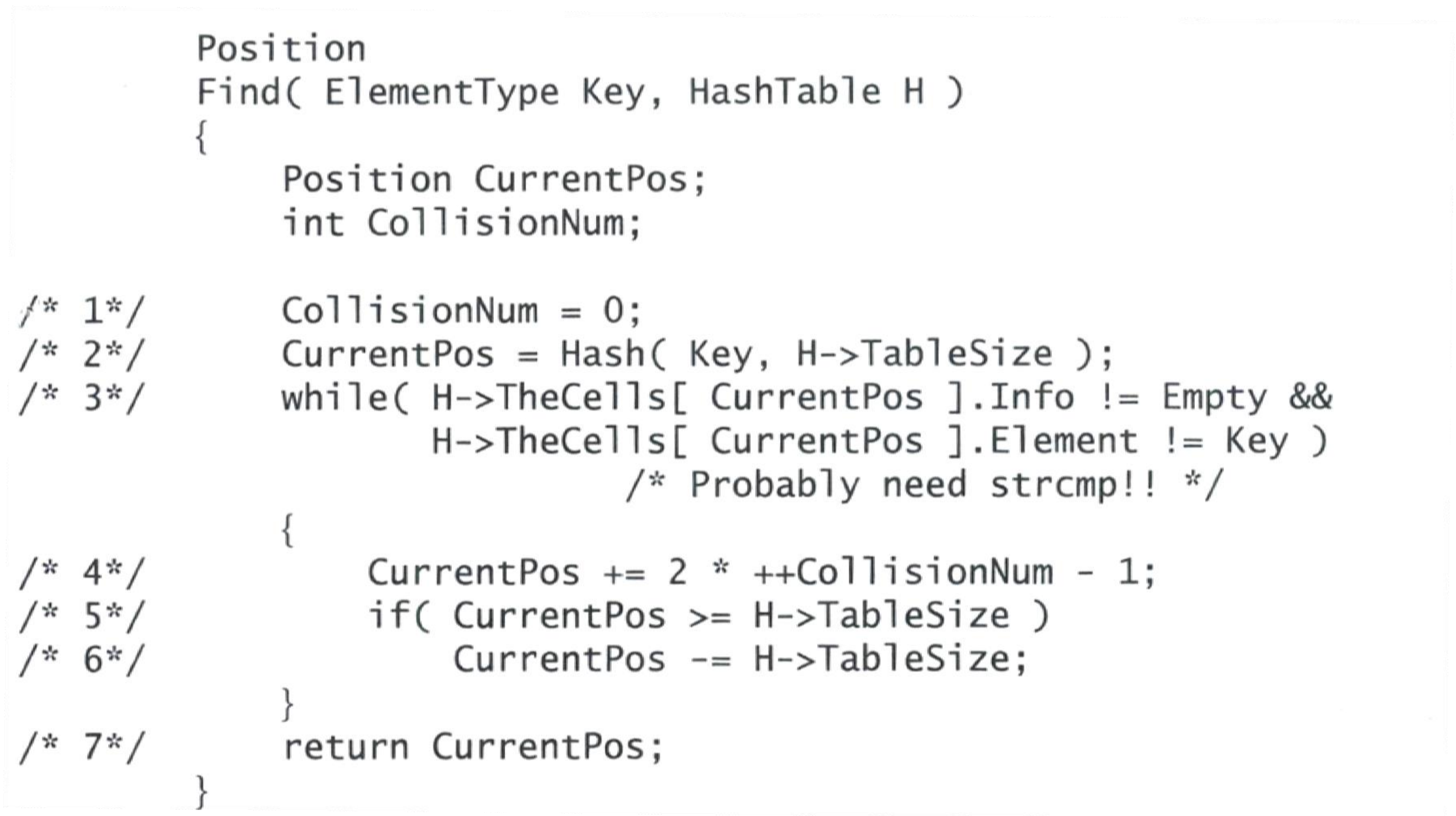
```
/* Place in the implementation file */
enum KindOfEntry { Legitimate, Empty, Deleted };

struct HashEntry
{
    ElementType       Element;
    enum KindOfEntry Info;
};

typedef struct HashEntry Cell;

/* Cell *TheCells will be an array of */
/* HashEntry cells, allocated later */
struct HashTbl
{
    int TableSize;
    Cell *TheCells;
};
```

5.14 Type declaration for open addressing hash tables

Initialization for open addressing

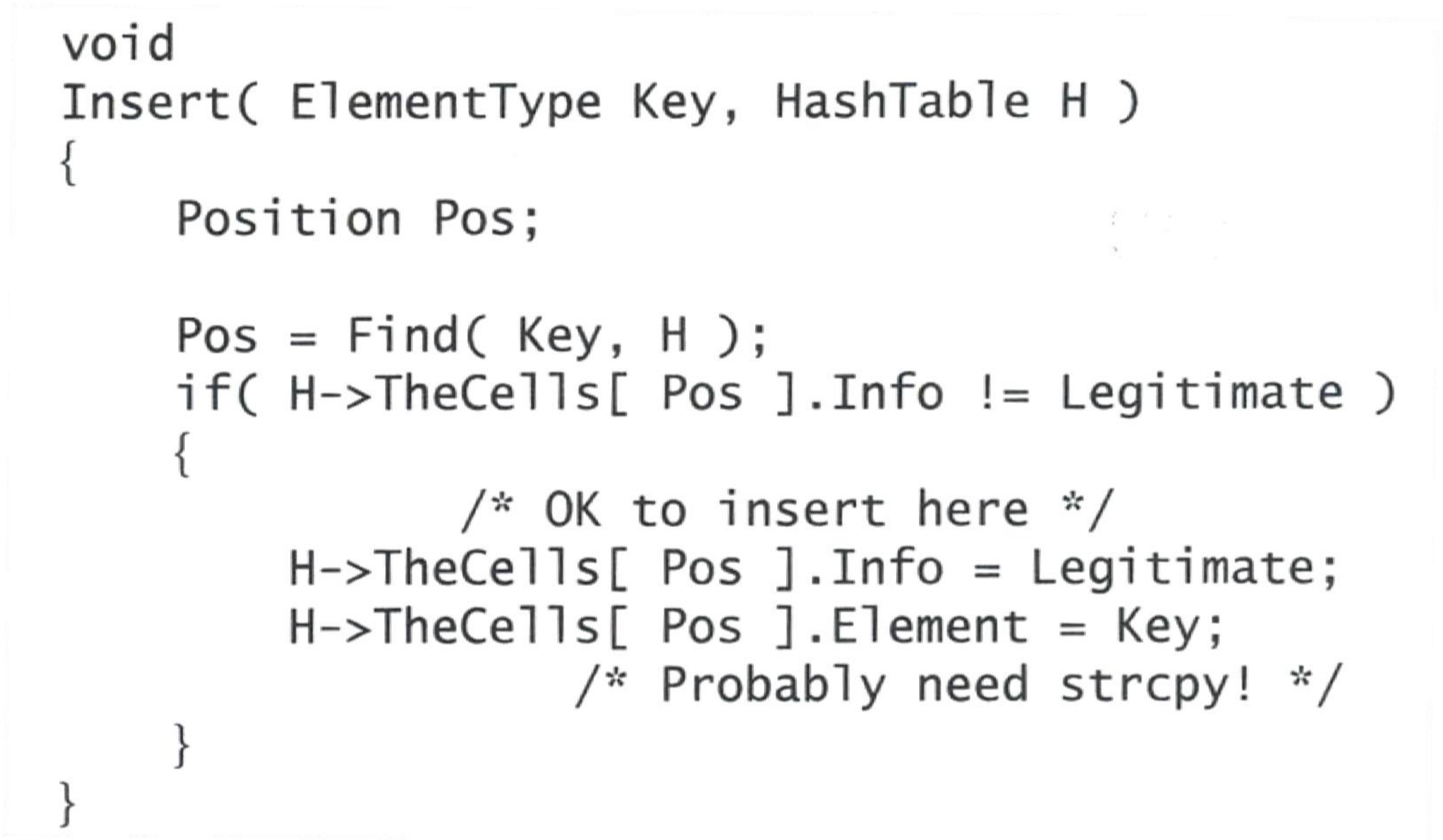
```
               HashTable
               InitializeTable( int TableSize )
               {
                   HashTable H;
                   int i;

/* 1*/             if( TableSize < MinTableSize )
                   {
/* 2*/                 Error( "Table size too small" );
/* 3*/                 return NULL;
                   }

                   /* Allocate table */
/* 4*/             H = malloc( sizeof( struct HashTbl ) );
/* 5*/             if( H == NULL )
/* 6*/                 FatalError( "Out of space!!!" );

/* 7*/             H->TableSize = NextPrime( TableSize );

                   /* Allocate array of Cells */
/* 8*/             H->TheCells = malloc( sizeof( Cell ) * H->TableSize );
/* 9*/             if( H->TheCells == NULL )
/*10*/                 FatalError( "Out of space!!!" );

/*11*/             for( i = 0; i < H->TableSize; i++ )
/*12*/                 H->TheCells[ i ].Info = Empty;

/*13*/             return H;
               }
```

Find for hashing with QP



```
Position
Find( ElementType Key, HashTable H )
{
    Position CurrentPos;
    int CollisionNum;

    CollisionNum = 0;
    CurrentPos = Hash( Key, H->TableSize );
    while( H->TheCells[ CurrentPos ].Info != Empty &&
           H->TheCells[ CurrentPos ].Element != Key )
                    /* Probably need strcmp!! */
    {
        CurrentPos += 2 * ++CollisionNum - 1;
        if( CurrentPos >= H->TableSize )
            CurrentPos -= H->TableSize;
    }
    return CurrentPos;
}
```

Insert for hashing with QP



```
void
Insert( ElementType Key, HashTable H )
{
    Position Pos;

    Pos = Find( Key, H );
    if( H->TheCells[ Pos ].Info != Legitimate )
    {
                /* OK to insert here */
        H->TheCells[ Pos ].Info = Legitimate;
        H->TheCells[ Pos ].Element = Key;
                    /* Probably need strcpy! */
    }
}
```

Double Hashing

- Double hashing uses a secondary hash function $\boldsymbol{h_2(k)}$ and handles collisions by placing an item in the first available cell of the series

$$\boldsymbol{(i + j^* h_2(k)) \bmod N} \text{ for } \boldsymbol{j = 0,\ 1, \ldots, N-1}$$

- The secondary hash function $\boldsymbol{h_2(k)}$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

Double Hashing Example

- Bad one

  (Ex) $\boldsymbol{hash_2(X) = X \bmod 9}$ and we want to insert 99

  - The function must never evaluate to zero
  - Make sure all cells can e pro

- Good one

  (Ex) $\boldsymbol{hash_2(X) = R - (X \bmod R)}$, where R is a prime smaller than TableSize

  - Figure 5.18 for R = 7

  (Ex) $hash_2(49) = 7 - 0 = 7$

89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.18 Open addressing hash table with double hashing, after each insertion

Rehashing

- If the table gets too full, the running time for the operations will start taking too long.
- To build another table that is about twice as big with a new hash function, computing new hash values for each element.
- Expensive operation requiring O(N) running time

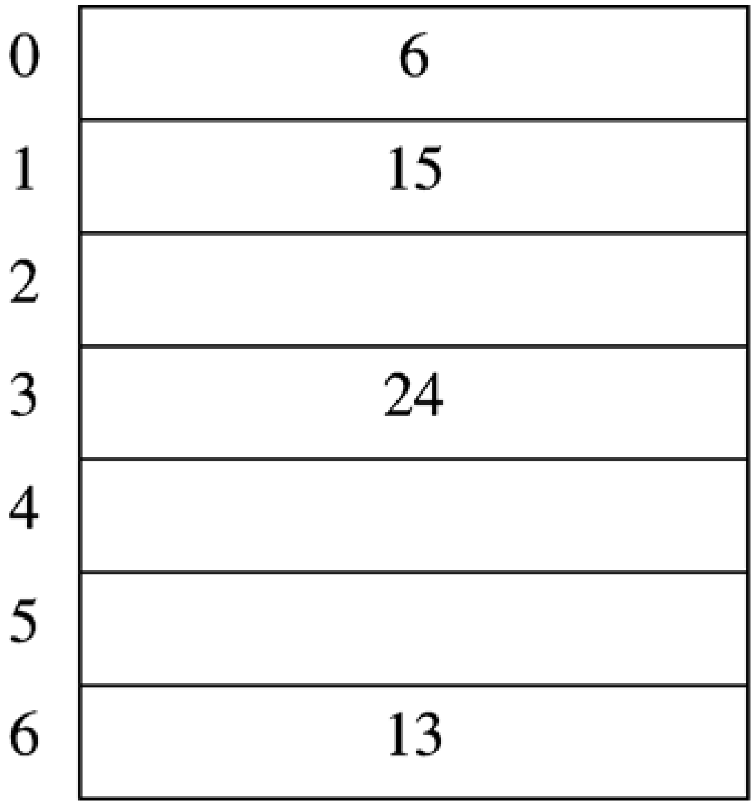


Figure 5.19 Open addressing hash table with linear probing with input 13,15,6,24

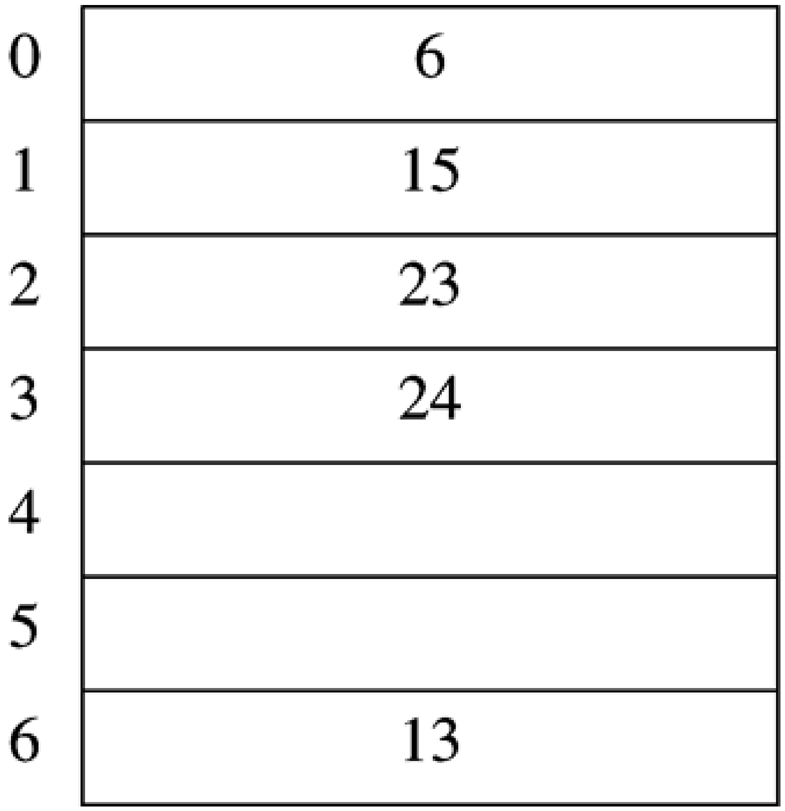


Figure 5.20 Open addressing hash table with linear probing after 23 is inserted

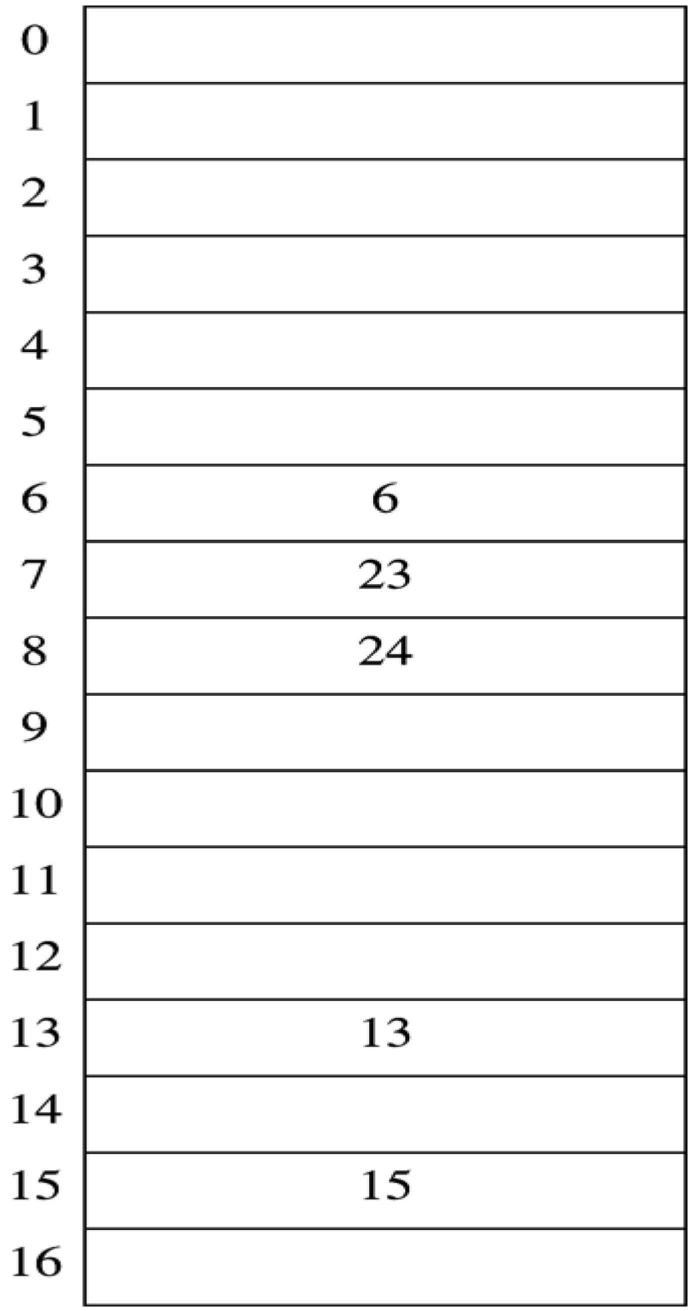


Figure 5.21 Open addressing hash table after rehashing

```
        HashTable
        Rehash( HashTable H )
        {
            int i, OldSize;
            Cell *OldCells;

/* 1*/      OldCells = H->TheCells;
/* 2*/      OldSize  = H->TableSize;

            /* Get a new, empty table */
/* 3*/      H = InitializeTable( 2 * OldSize );

            /* Scan through old table, reinserting into new */
/* 4*/      for( i = 0; i < OldSize; i++ )
/* 5*/          if( OldCells[ i ].Info == Legitimate )
/* 6*/              Insert( OldCells[ i ].Element, H );

/* 7*/      free( OldCells );

/* 8*/      return H;
        }
```

Figure 5.22 Rehashing for open addressing hash tables

Extendible Hashing

- When the amount of data is too large to fit in main memory
- Main concern is the number of disk accesses for retrieving data

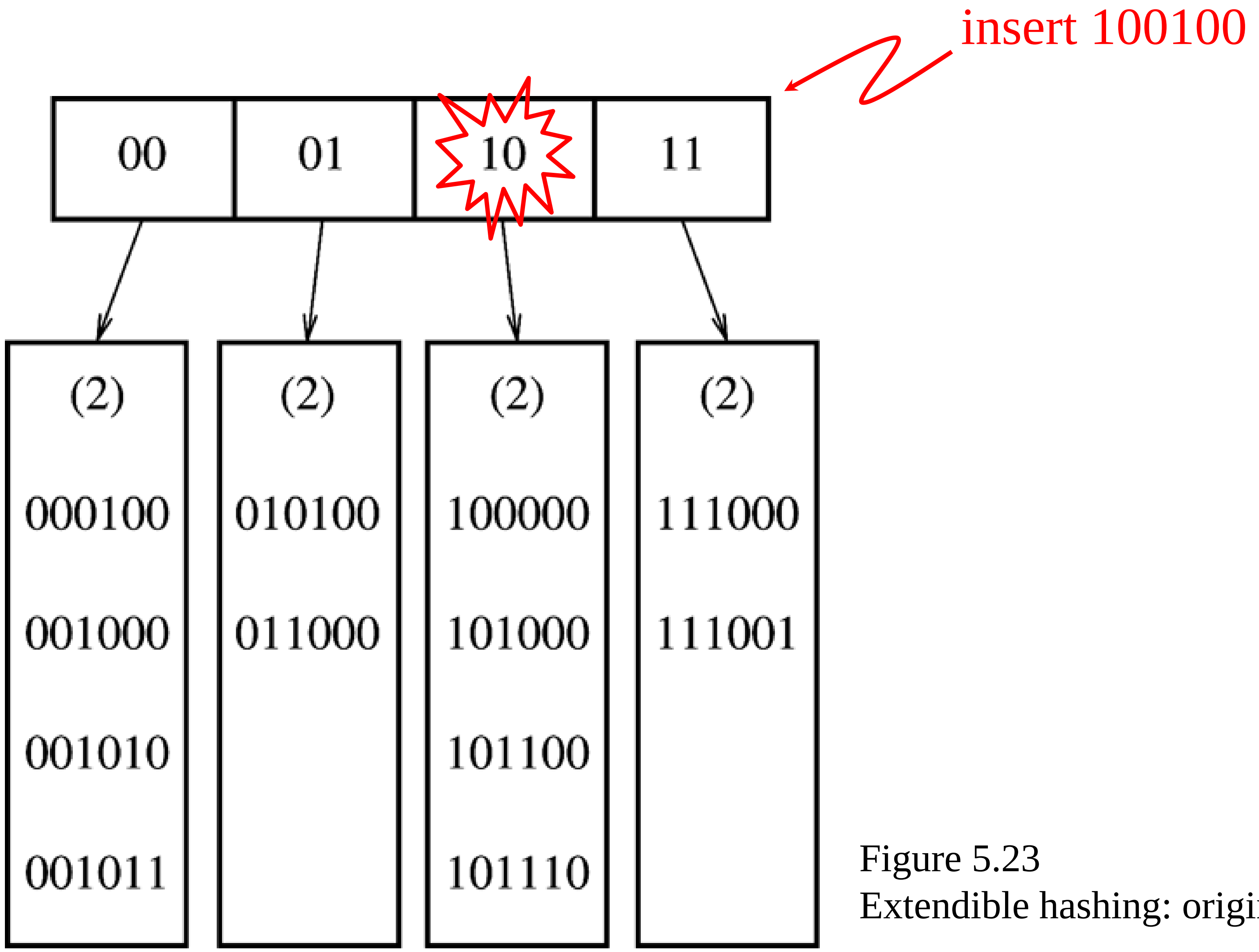


Figure 5.23
Extendible hashing: original data

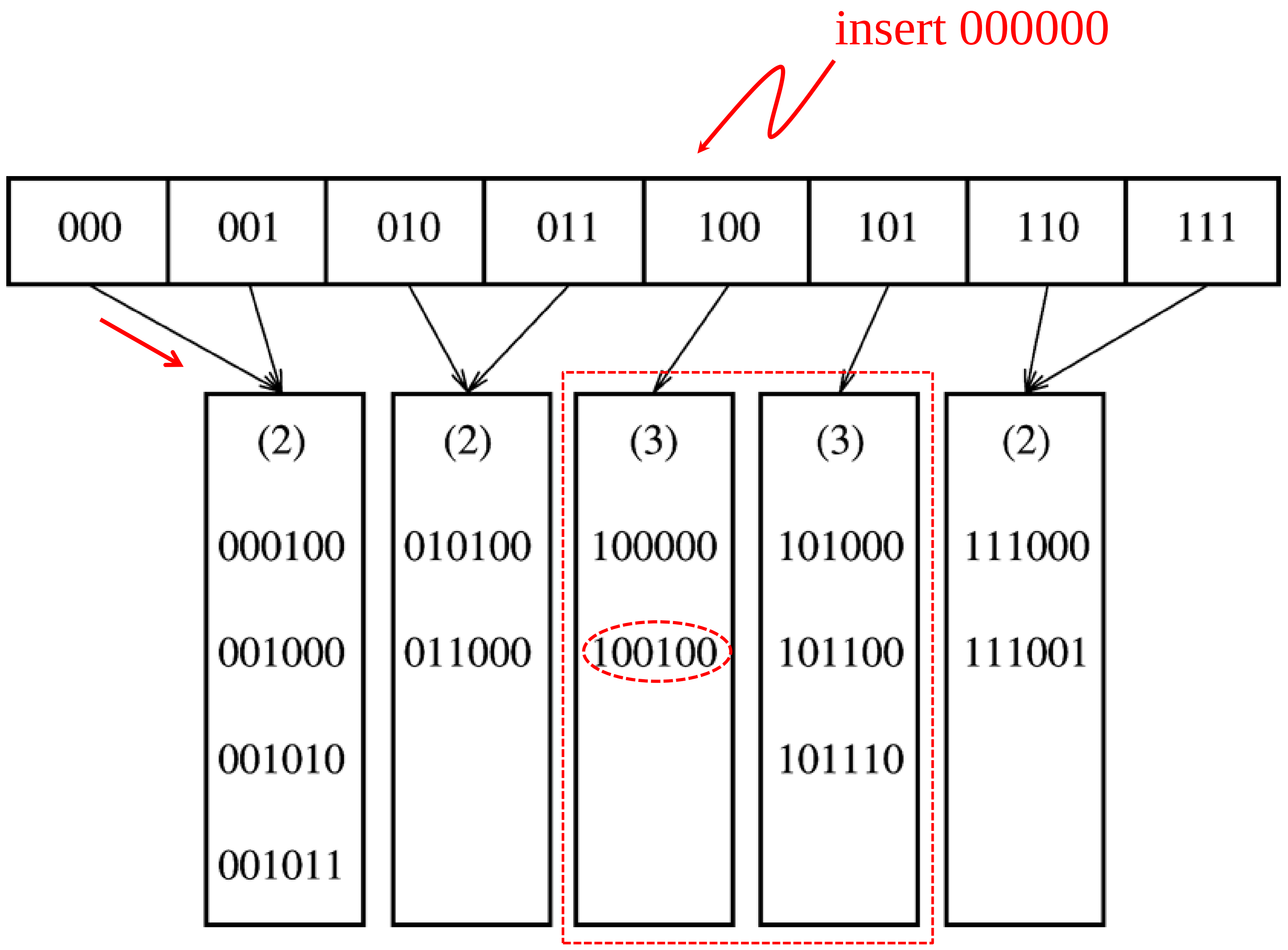


Figure 5.24 Extendible hashing : after insertion of 100100 and directory split

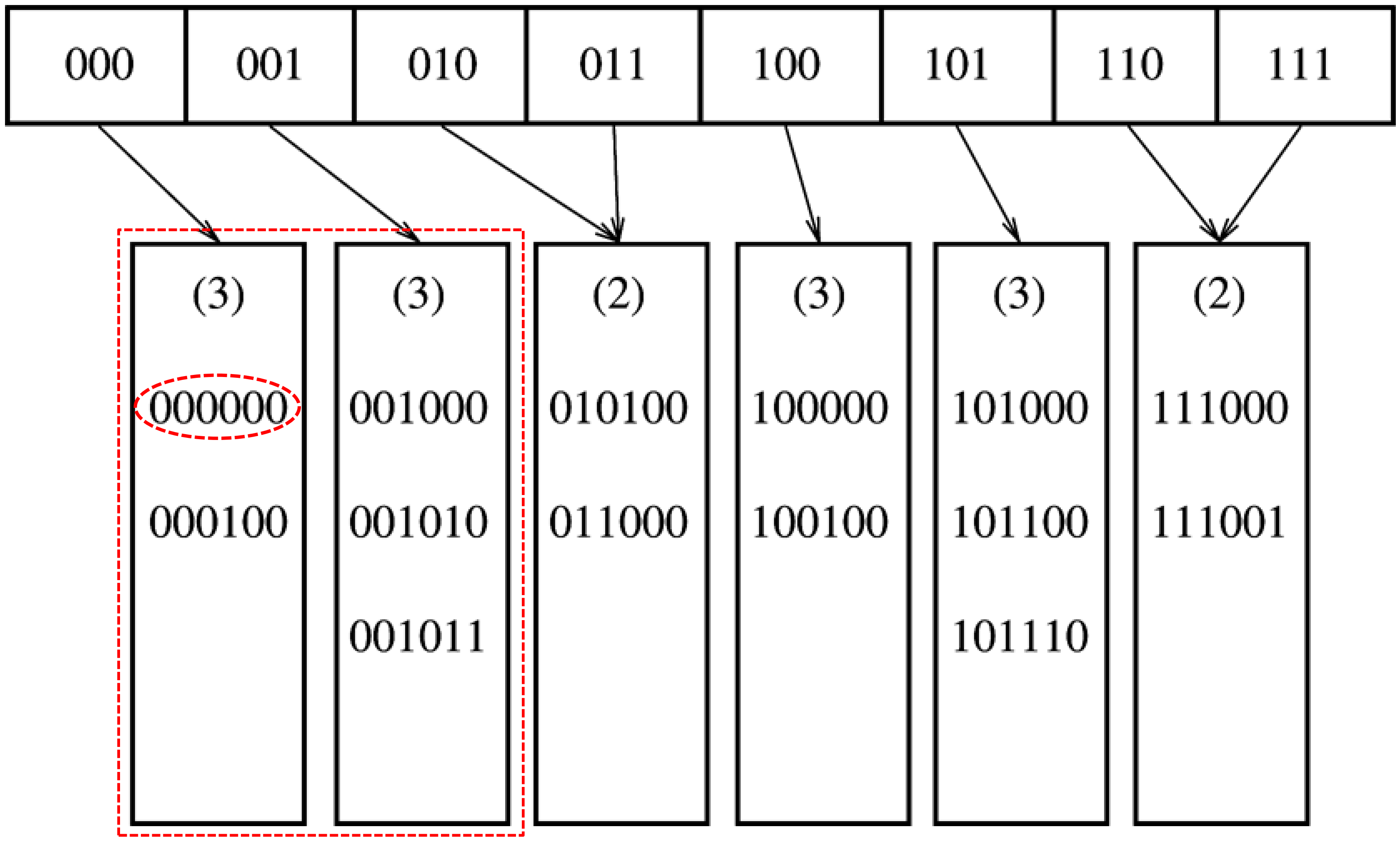


Figure 5.25 Extendible hashing : after insertion of 000000 and leaf split