# Chap. 5 Hashing

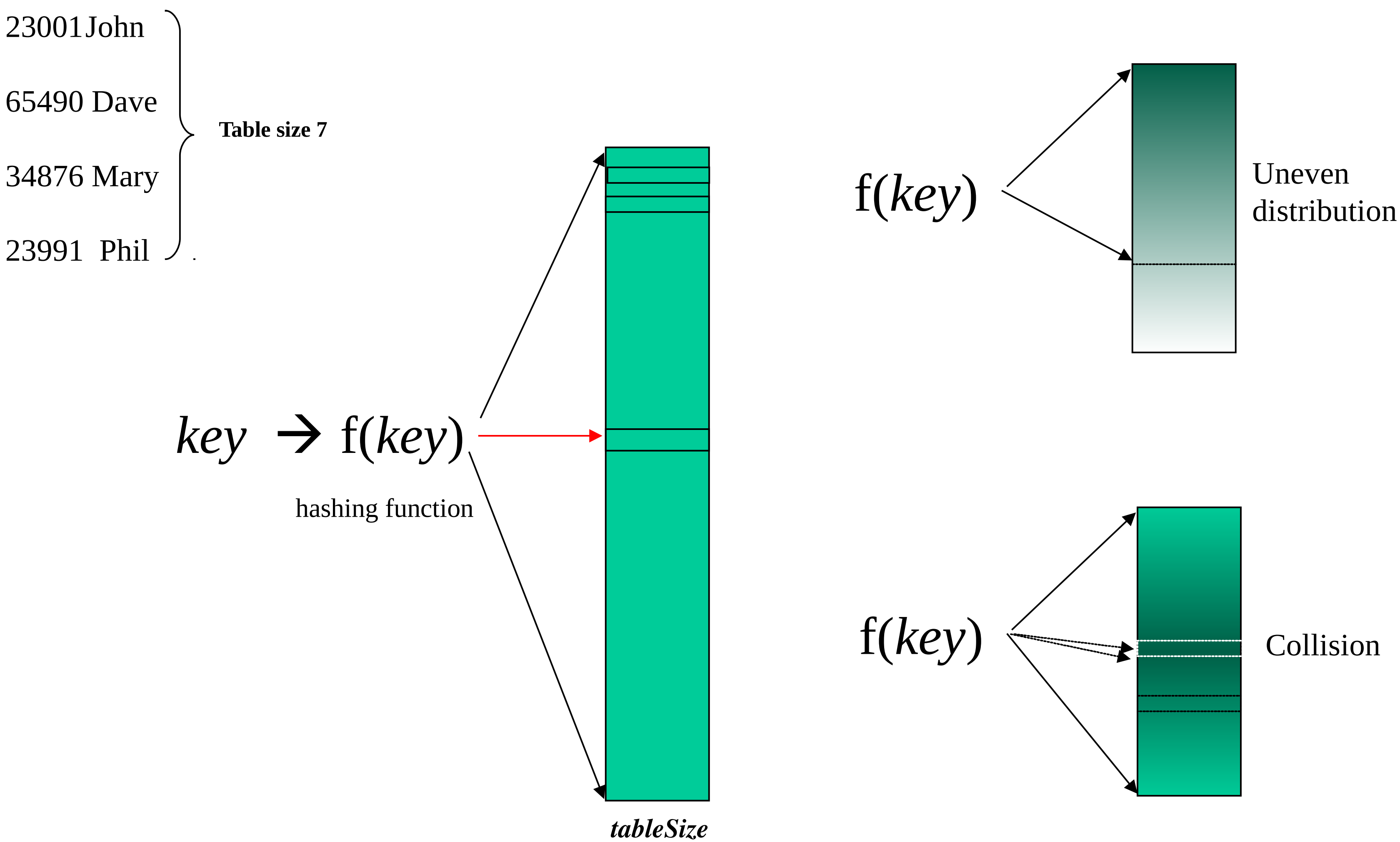
## 5.1 Basics

– Hashing is a technique mainly used to support fast (constant average time) insertion, deletion and find operations. It is not efficient, however, for order-based operations such as *findMin*, *findMax*, sorted print, etc.

– Hash table is a fixed sized array ($0$ to $tableSize - 1$). Each data item has a key that is mapped to a number between $0$ and $tableSize - 1$ by a hash function, and then placed in the corresponding cell of the hash table.

– If two data items are mapped to the same location, we say a collision occurs. Finding the appropriate hash function and collision handling are the two main issues in hashing.

# Find, Insert, Remove Elements



## 5.2 Hash function

- Hash function needs to have certain properties: easy implementation, simple (fast) computation, even distribution (over the hash table domain)
- Integer keys:

  $f(key) = (key \bmod tableSize)$

  *tableSize* is normally chosen to be a prime number to avoid uneven distribution
- String keys:
  - Add up the ASCII values of the characters in the string

```
for (i = 0; i< key length ( ); i++)
    hashVal += key[i];
```

  This function is simple to compute and implement, but does not distribute well.

- $f(key) = (key[0] + 27 * key[1] + 27^2 * key[2])$

  $f(key) = f(key) \bmod tableSize$

  It distribute well only when the first 3 characters of all keys are random (English is not random).

- A polynomial function of 37:

$$f(key) = \sum_{i=0}^{keySize-1} key[keySize - i - 1] \bullet 37^i$$

Horner's rule can be used to compute it.

e.g. $k_0 + 37k_1 + 37^2k_2 = ((k_2) \cdot 37 + k_1) \cdot 37 + k_0$

It has good distribution, but the computation can be expensive when keys are too long. Partial keys may be used to simplify the computation.

# 5.3 Separate Chaining

– Keep a list of all elements that hash to the same value. List ADT and linked list implementation can be used for each list.

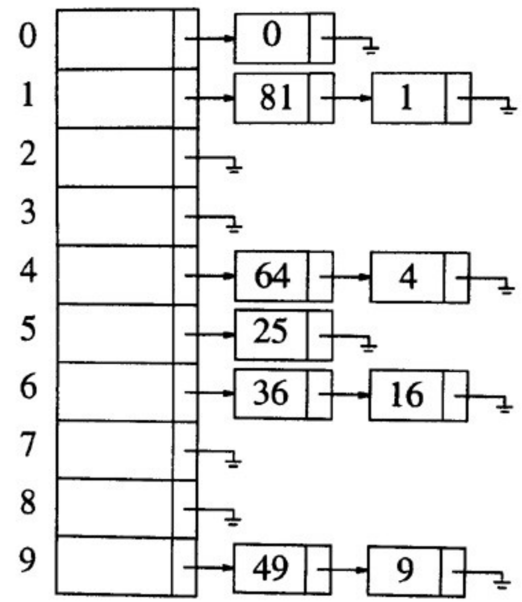



**Figure 5.5** A separate chaining hash table

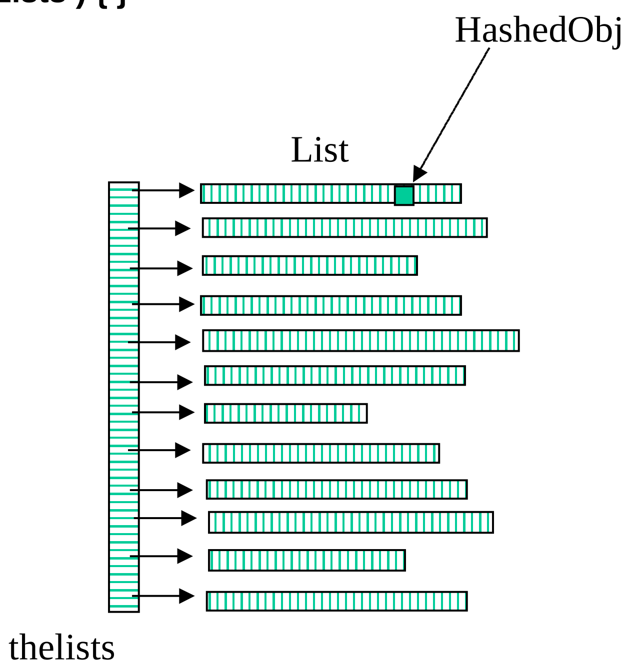
– *find( ):* identify the list (by hashing), and then perform a find operation in list ADT.

– *Insert( ):* perform find first. If found, increment the number of instances in the data item; otherwise, link it to the front end of the list.

– Implementation:

```
#ifndef _SEPARATE_CHAINING_H_
#define _SEPARATE_CHAINING_H_
#include "vector.h"
#include "mystring.h"
#include "LinkedList.h"
// SeparateChaining Hash table class
// CONSTRUCTION: an initialization for ITEM_NOT_FOUND
//   and an approximate initial size or default of 101
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Hashable find( x )     --> Return item that matches x
// void makeEmpty( )      --> Remove all items
// int hash( string str, int tableSize )--> Global method to hash strings
```

```
template <class HashedObj>
class HashTable  {
        public:
          explicit HashTable( const HashedObj & notFound, int size = 101 );
          HashTable( const HashTable & rhs )
            : ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ), theLists( rhs.theLists ) { }
          const HashedObj & find( const HashedObj & x ) const;
          void makeEmpty( );
          void insert( const HashedObj & x );
          void remove( const HashedObj & x );
          const HashTable & operator=( const HashTable & rhs );
        private:
          vector<List<HashedObj> > theLists;   // The array of Lists
          const HashedObj ITEM_NOT_FOUND;
      };
      int hash( const string & key, int tableSize );
      int hash( int key, int tableSize );
      #include "SeparateChaining.cpp"
      #endif
```

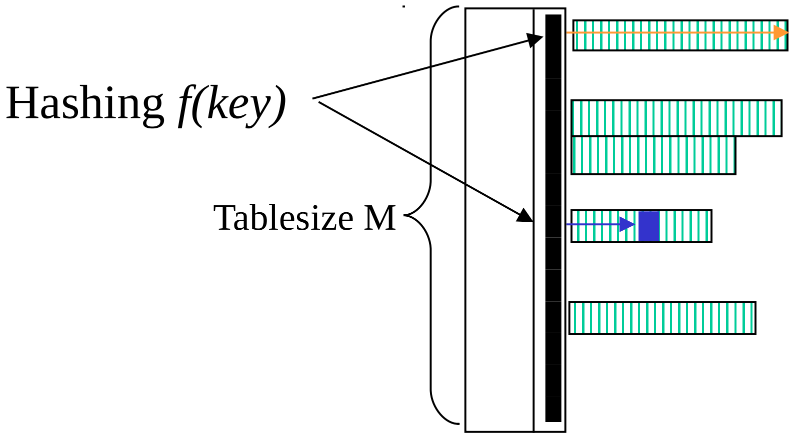


```
/*** Construct the hash table.*/
template <class HashedObj>
HashTable<HashedObj>::HashTable( const HashedObj &
  notFound, int size ) : ITEM_NOT_FOUND( notFound ),
  theLists( nextPrime( size ) )
{       }
/*** Insert item x into the hash table. If the item is
 * already present, then do nothing.        */
template <class HashedObj>
void HashTable<HashedObj>::insert( const HashedObj & x )
{
   List<HashedObj> & whichList = theLists[ hash( x,
  theLists.size( ) ) ];
   ListItr<HashedObj> itr = whichList.find( x );
   if( itr.isPastEnd( ) )
      whichList.insert( x, whichList.zeroth( ) );
}
/**Remove item x from the hash table.        */
template <class HashedObj>
void HashTable<HashedObj>::remove( const HashedObj &
  x )
{
  theLists[ hash( x, theLists.size( ) ) ].remove( x );
}
/*** Find item x in the hash table. Return the matching item or
   ITEM_NOT_FOUND if not found       */
template <class HashedObj>
const HashedObj & HashTable<HashedObj>::find( const
  HashedObj & x ) const
{
   ListItr<HashedObj> itr;
   itr = theLists[ hash( x, theLists.size( ) ) ].find( x );
   return itr.isPastEnd( ) ? ITEM_NOT_FOUND : itr.retrieve( );
}
```

```
/* * Make the hash table logically empty. */
template <class HashedObj>
void HashTable<HashedObj>::makeEmpty( )
{
   for( int i = 0; i < theLists.size( ); i++ )
      theLists[ i ].makeEmpty( );
}
/** * Deep copy. */
template <class HashedObj>
const HashTable<HashedObj> &
HashTable<HashedObj>::operator=( const HashTable<HashedObj> & rhs )
{       if( this != &rhs )
      theLists = rhs.theLists;
   return *this;
}
/* * A hash routine for string objects.*/
int hash( const string & key, int tableSize )
{       int hashVal = 0;
   for( int i = 0; i < key.length( ); i++ )
      hashVal = 37 * hashVal + key[ i ];
   hashVal %= tableSize;
   if( hashVal < 0 )
      hashVal += tableSize;
   return hashVal;
}
/** A hash routine for ints.        */
int hash( int key, int tableSize )
{       if( key < 0 ) key = -key;
   return key % tableSize;
}
```

– *Load factor:* $\lambda$ = *number of data elements / tableSize*

- The average length of the list is $\lambda$
- Unsuccessful search costs, on average, $\lambda$ plus hash function computation
- Successful search: $1_{\text{hash function}} + (\lambda / 2)$
- A general rule: $\lambda \approx 1.0$




Expected other nodes searched

$(N-1)/M = \lambda - 1/M \rightarrow \lambda$

Where M is the number of list and N is the num of element

## 5.4 Open Addressing

- Basic
  - Disadvantages of separate chaining:

    dynamic memory allocation; the implementation of a separate data structure.
  - Open addressing:

    if a collision occurs, alternative cells are tried until an empty cell is found, *i.e.* $h_0(x), h_1(x), h_2(x), \ldots$ are tried successively, where

$$h_i(x) = (hash(x) + f(i)) \ mod \ tableSize \qquad (f(0) = 0)$$

  - function $f$ is called the <u>*collision resolution strategy*</u> function
  - Load factor in open addressing is normally $\lambda \leq 0.5$
- Linear Probing
  - $f$ is a linear function of $i$.

    example: $f(i) = i$
  - Example: `[ Fig. 5.11]`

$h_0(x) = hash(x)$

$h_1(x) = (hash(x) + 1) \bmod tablesize$

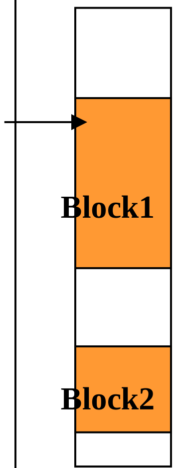
$h_2(x) = (hash(x) + 2) \bmod tablesize$

.. ..

$h_i(x) = (hash(x) + i) \bmod tablesize$

{ 89, 18, 49, 58, 69 }

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | 0 | | 0 | | 0 | 49 | 0 | 49 | 0 | 49 |
| 1 | | 1 | | 1 | | 1 | | 1 | 58 | 1 | 58 |
| 2 | | 2 | | 2 | | 2 | | 2 | | 2 | 69 |
| 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 8 | | 8 | | 8 | 18 | 8 | 18 | 8 | 18 | 8 | 18 |
| 9 | | 9 | 89 | 9 | 89 | 9 | 89 | 9 | 89 | 9 | 89 |




- Average number of probes:

  insertion & unsuccessful searching: $\frac{1}{2}(1+\frac{1}{(1-\lambda)^2})$

  successful searching: $\frac{1}{2}(1+\frac{1}{1-\lambda})$

- Primary clustering: blocks of the hash table are formed so that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.
- Random collision resolution:

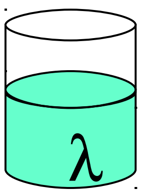
  If the clustering is not considered, i.e. each probe is independent of the previous probes:

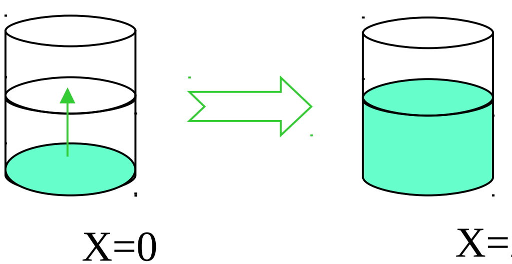
  Unsuccessful Search: $\frac{1}{1-\lambda}$ cells to probe

  Successful Search: number of probes required when the particular element was inserted.

  Average successful search:

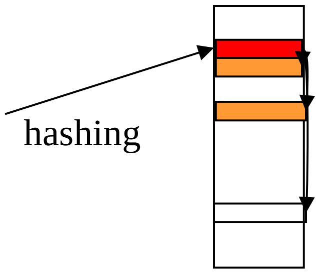
$$I(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-X}\,dx = \frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$




- Comparisons: `[Fig. 5.12]`

## – Quadratic Probing




- *f(i)* is a quadratic function

  e.g. $f(i) = i^2$

- Example: [Fig. 5.13]
- Finding an empty cell is not always guaranteed if the table is more than half full or if the *tableSize* is not prime.
- Theorem:

  If quadratic probing is used, and the *tableSize* is prime, then a new element can always be inserted if the table is at least half empty.

- Deletion cannot be performed in a standard way,

  *i.e.* lazy deletion required.

- Secondary clustering: elements that hash to the same position will probe the same alternative cells.
- Implementation: [Fig. 5.14, 5.15, 5.16]