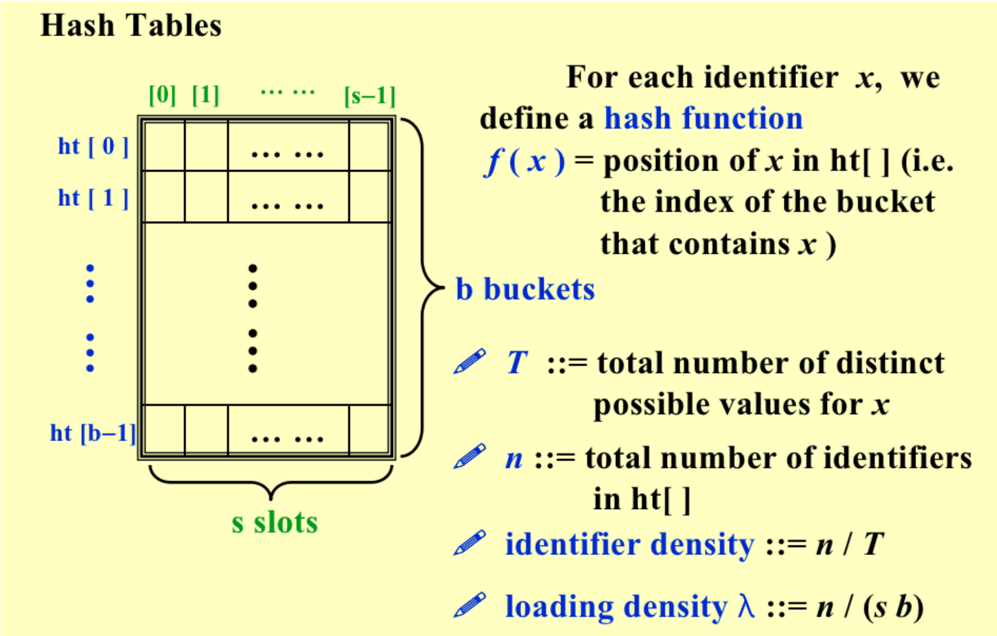
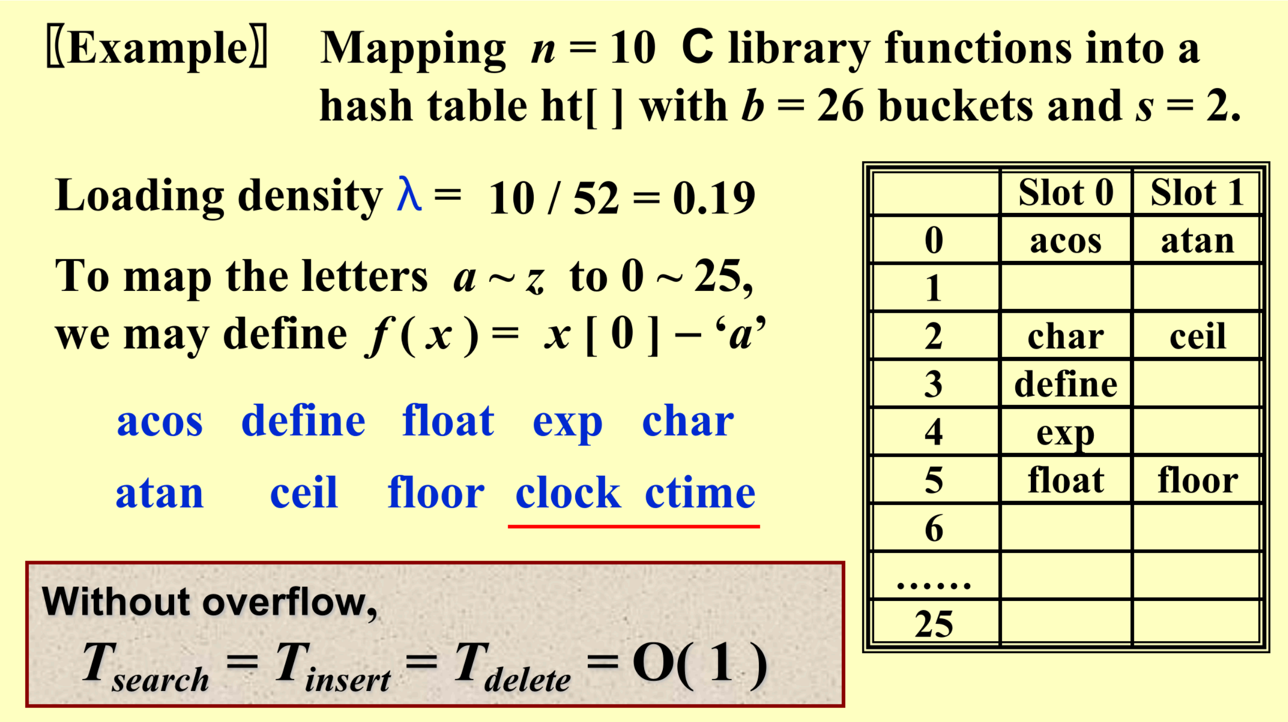
1 General Idea



- A collision occurs when we hash two nonidentical identifiers into the same bucket, i.e. f (i1) = f (i2) when $i_1 \neq i_2$.
- An overflow occurs when we hash a new identifier into a full bucket.



2 Hash Function

- f (x) must be easy to compute and minimizes the number of collisions.
- f (x) should be unbiased. That is, for any x and any i, we have that Probability(f (x) = i) = 1 / b. Such kind of a hash function is called a uniform hash function.
- PPT 上有很多hash函数

$$f(x) = (\Sigma x[N-i-1] * 32^i)\ \%\ TableSize \tag{5}$$

```c
Index Hash3( const char *x, int TableSize )
{
        unsigned  int  HashVal = 0;
/* 1*/  while( *x != '\0' )
/* 2*/        HashVal = ( HashVal << 5 ) + *x++;
/* 3*/  return HashVal % TableSize;
}
```

- If x is too long (e.g. street address), the early characters will be left-shifted out of place.

3 Separate Chaining

---- keep a list of all keys that hash to the same value

```c
struct  ListNode;
typedef  struct  ListNode  *Position;
struct  HashTbl;
typedef  struct  HashTbl  *HashTable;
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;
};
```

```c
HashTable  InitializeTable( int TableSize )
{   HashTable  H;
    int  i;
    if ( TableSize < MinTableSize ) {
      Error( "Table size too small" );  return NULL;
    }

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

    H->TableSize = NextPrime( TableSize );  /* Better be prime */
    H->TheLists = malloc( sizeof( List ) * H->TableSize );  /*Array of lists*/
    if ( H->TheLists == NULL )   FatalError( "Out of space!!!" );

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

```
        H->TheLists[ i ]->Next = NULL;
    }
    return  H;
}
```

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

    L = H->TheLists[ Hash( Key, H->TableSize ) ]; // Hash是找key位置的链表头

    P = L->Next;
    while( P != NULL && P->Element != Key )  /* Probably need strcmp */
      P = P->Next;
    return P;
}
```

```
void  Insert ( ElementType Key, HashTable H )
{
    Position   Pos, NewCell;
    List  L;
    Pos = Find( Key, H );
    if ( Pos == NULL ) {   /* Key is not found, then insert */
      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;
      }
    }
}
```

4 Open Addressing

---- find another empty cell to solve collision (avoiding pointers)

```
Algorithm: insert key into an array of hash table
{
    index = hash(key);
    initialize i = 0 ------- the counter of probing;
    while ( collision at index ) {
      index = ( hash(key) + f(i) ) % TableSize;
      // Collision resolving function. f(0) = 0.
      if ( table is full )    break;
      else  i ++;
    }
    if ( table is full )
      ERROR ("No space left");
    else
      insert key at index;
}
```

4.1 Linear Probing

$$hash(i) = index + f(i) \tag{6}$$

产生冲突就下移一个位置

〖Example〗 Mapping $n = 11$ C library functions into a hash table ht[ ] with $b = 26$ buckets and $s = 1$.

acos atoi char define exp
ceil cos float atol floor ctime

bucket	x	search time
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
… …		
25		

Loading density $\lambda = 11 / 26 = 0.42$

Average search time = 41 / 11 = 3.73

Analysis of the linear probing show that the expected number of probes

$$p = \begin{cases} \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right) & \text{for insertions and unsuccessful searches} \\ \frac{1}{2}\left(1+\frac{1}{1-\lambda}\right) & \text{for successful searches} \end{cases} = 1.36$$

- Cause primary clustering: any key that hashes into the cluster will add to the cluster after several attempts to resolve the collision.
- search time最少也是1

4.2 Quadratic Probing

$$f(i) = i^2 \tag{7}$$

> *Theorem : If quadratic probing is used, and the table size is prime, then **a new element can always be inserted if the table is at least half empty.*** (考试可能考)

> *Note: If the table size is a prime of the form 4k + 3, then **the quadratic probing $f(i) = \pm i^2$ can probe the entire table.***

```
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 ) {
      CurrentPos += 2 * ++CollisionNum - 1;
      if ( CurrentPos >= H->TableSize )  CurrentPos -= H->TableSize;
    }
    return CurrentPos;
}
```

```
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 ].Element = Key; /* Probably need strcpy */
    }
}
```

4.3 Double Hashing

$$f(i) = i * hash_2(x) \tag{8}$$

ⓘ **$hash_2(x) \neq 0$;** ⓘ **make sure that all cells can be probed.**

Tip: $hash_2(x) = R - (x \% R)$ with R a prime smaller than TableSize, will work well.

Note: ① **If double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy.**
② **Quadratic probing does not require the use of a second hash function and is thus likely to be simpler and faster in practice.**

- Double Hashing是理论上最优的，但是实际上操作很慢

5 Rehashing

- Build another table that is about twice as big;
- Scan down the entire original hash table for non-deleted elements;
- Use a new function to hash those elements into the new table.

    花费 $T(N) = O(N)$ 的时间

When to rehash：As soon as the table is half full; When an insertion fails; When the table reaches a certain load factor