Hashing

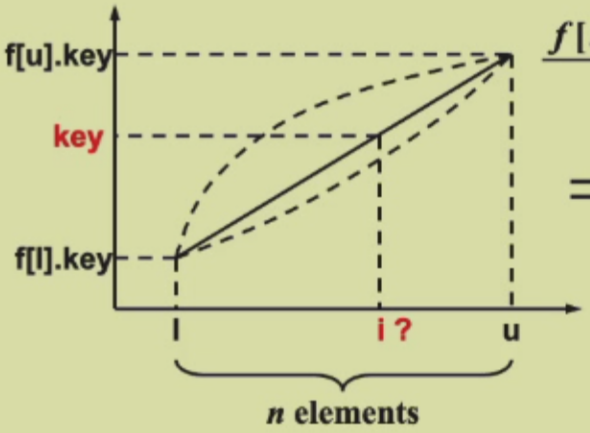
CHAPTER 7

HASHING

Search by Formula

Interpolation Search :

Find key from a sorted list f [l].key, f [l+1].key, ... , f [u].key.



$$\frac{f[u].key - f[l].key}{n} = \frac{key - f[l].key}{i - l}$$

$$\Rightarrow \quad i = l + \frac{(key - f[l].key) \cdot n}{f[u].key - f[l].key}$$

If (f [i].key < key) l = i ;

Else u = i ;

General Idea

Symbol Table (== Dictionary) ::= { < name, attribute > }

〖Example〗 In Oxford English dictionary

name = since

attribute = a list of meanings
- M[0] = after a date, event, etc.
- M[1] = seeing that (expressing reason)
- ....

〖Example〗 In a symbol table for a compiler

name = identifier (e.g. int)

attribute = a list of lines that use the identifier, and some other fields

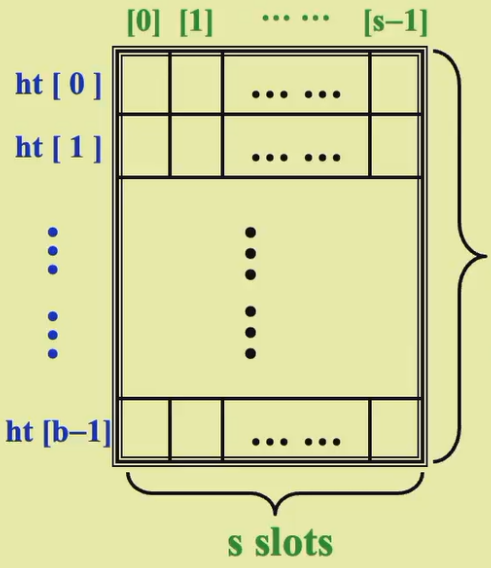
❖ Symbol Table ADT:

Objects: A set of name-attribute pairs, where the names are unique

Operations:

- SymTab Create(TableSize)
- Boolean IsIn(symtab, name)
- Attribute Find(symtab, name)
- SymTab Insert(symtab, name, attr)
- SymTab Delete(symtab, name)

Hash Tables



For each identifier x, we define a hash function $f(x)$ = position of x in ht[ ] (i.e. the index of the bucket that contains x)

b buckets

- T ::= total number of distinct possible values for x
- n ::= total number of identifiers in ht[ ]
- identifier density ::= n / T
- loading density λ ::= $n / (s\ b)$

> T：x的可能取值范围
>
> n：table中的所有x的数量
>
> loading density: n占整个hash table的比例

§ 1 General Idea

A collision occurs when we hash two nonidentical identifiers into the same bucket, i.e. $f(i_1) = f(i_2)$ when $i_1 \neq i_2$.

An overflow occurs when we hash a new identifier into a full bucket.

> collision: 不同的identifier归到了同一个bucket
>
> overflow: 将一个identifier归到了一个full bucket

〖Example〗 Mapping $n = 10$ C library functions into a hash table ht[ ] with $b = 26$ buckets and $s = 2$.

Loading density $\lambda = 10 / 52 = 0.19$

To map the letters $a \sim z$ to $0 \sim 25$, we may define $f(x) = x[0] - 'a'$

acos define float exp char
atan ceil floor clock

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
.....		
25		

Without overflow,

$$T_{search} = T_{insert} = T_{delete} = O(1)$$

Hash function

Properties of f:

① $f(x)$ must be easy to compute and minimizes the number of collisions.

② $f(x)$ 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.

```
f(x) = x % TableSize ;  /* if x is an integer */
```

☹ What if *TableSize* = 10 and x's are all end in zero?

☺ *TableSize* = prime number ---- good for random integer keys

$f(x) = (\Sigma\, x[i])\ \%\ TableSize$; /* if x is a string */

〖Example〗 *TableSize* = 10,007 and string length of $x \le 8$.

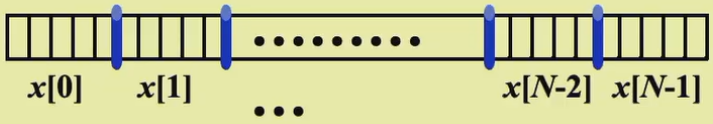
If $x[i] \in [0, 127]$, then $f(x) \in [0,$ 1016 $]$ ☹

$f(x) = (x[0] + x[1]*27 + x[2]*27^2)\ \%\ TableSize$;

Total number of combinations = $26^3 = 17{,}576$

☹ Actual number of combinations < 3000

$f(x) = (\Sigma\, x[N-i-1] * 32^i)\ \%\ TableSize$;




```
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

太长了就截取字符串

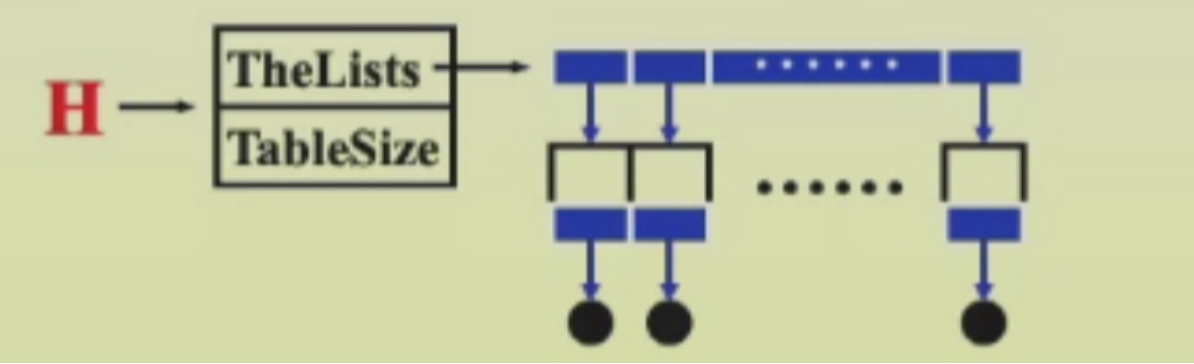
Separate Chaining

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

```
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;
};
```

Create an empty table

```
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;
}
```



Find a key from a hash table

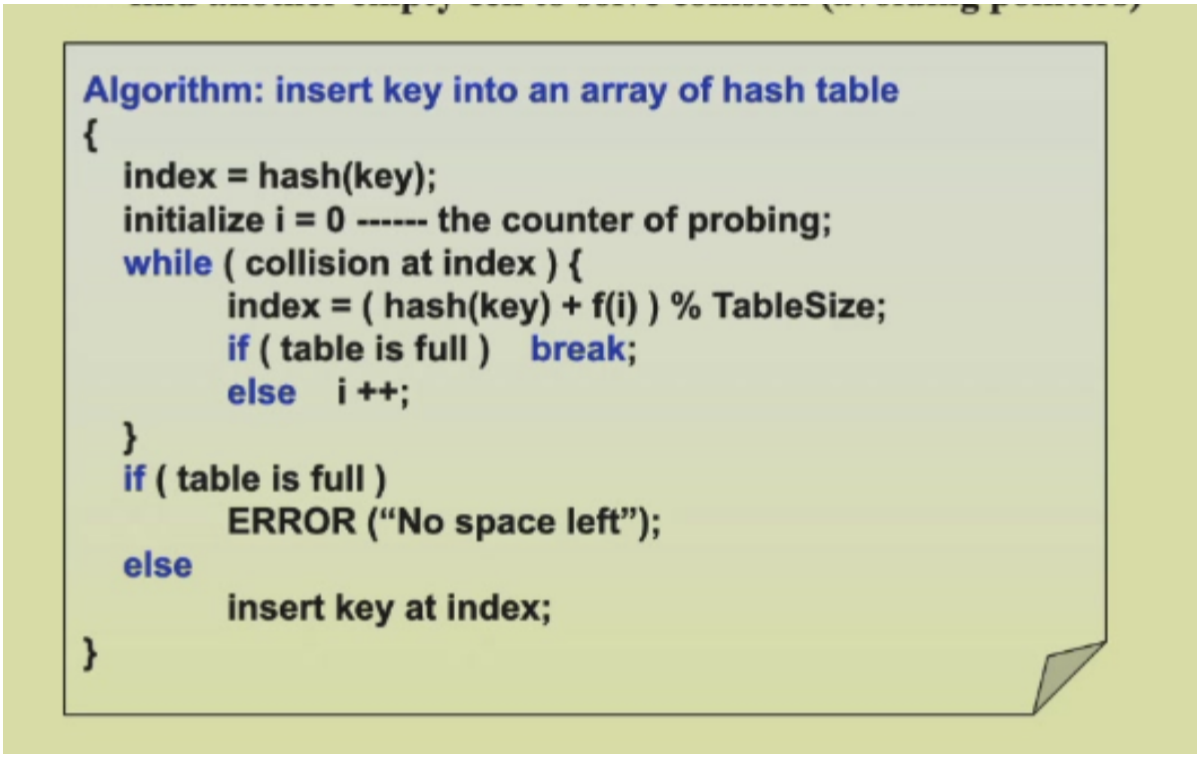
```
Position  Find ( ElementType Key, HashTable H )
{
   Position P;
   List L;

   L = H->TheLists[ Hash( Key, H->TableSize ) ];

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

Insert a key into a hash table

```
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;
         }
   }
}
```

> 变快：趋近——映射，减小链表长度

Open Addressing

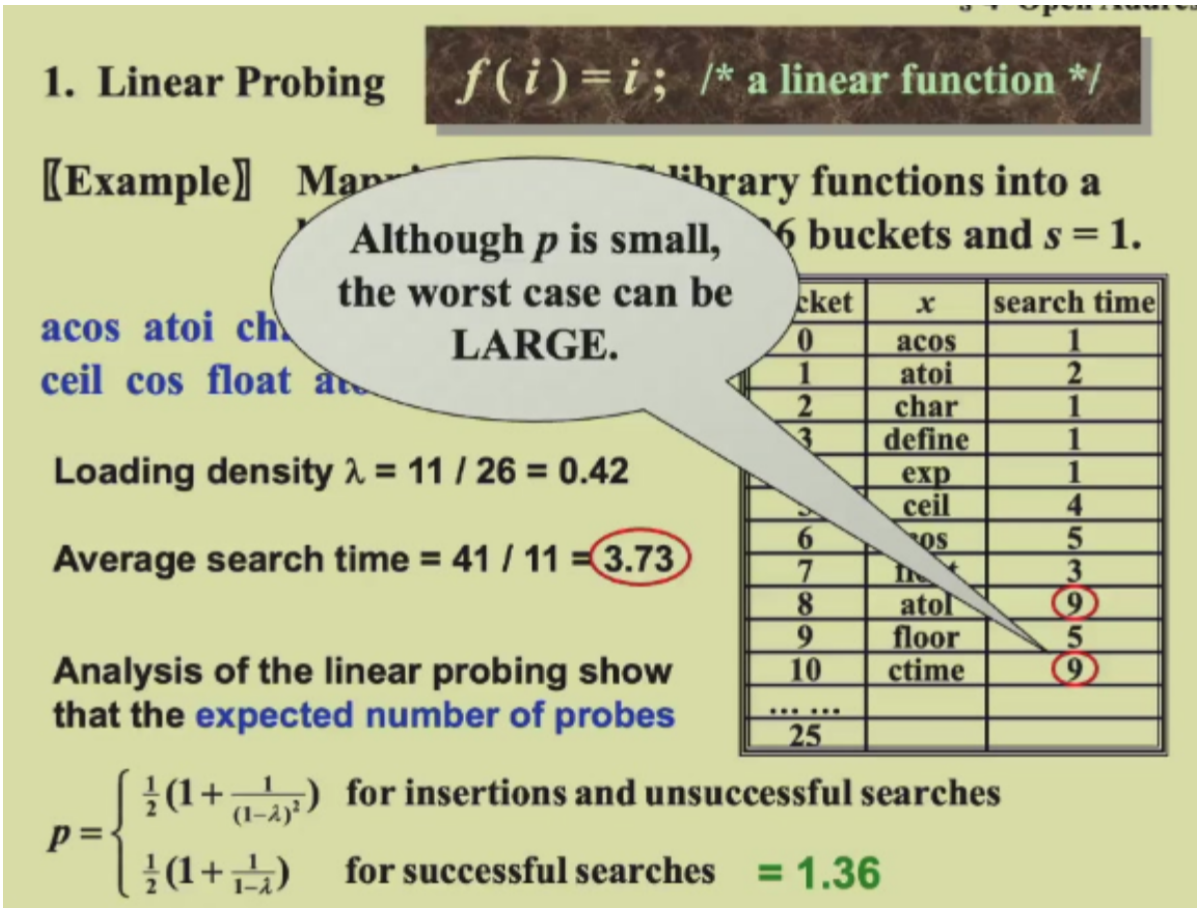


```
Algorithm: insert key into an array of hash table
{
   index = hash(key);
   initialize i = 0 ------ the counter of probing;
   while ( collision at index ) {
          index = ( hash(key) + f(i) ) % TableSize;
          if ( table is full )    break;
          else    i ++;
   }
   if ( table is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

> 使用f(i)进行一个步进，找到下一个对应的bucket
>
> f: collision resolving function
>
> Generally, $\lambda < 0.5$

Linear Probing

$$f(i) = i$$



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

> Primary clustering: 主聚集

Quadratic Probing

$$f(i) = i^2$$

【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.

Proof: Just prove that the first ⌊TableSize/2⌋ alternative locations are all distinct. That is, for any $0 < i \neq j \leq \lfloor TableSize/2 \rfloor$, we have

$$(h(x) + i^2) \% TableSize \neq (h(x) + j^2) \% TableSize$$

Suppose: $h(x) + i^2 = h(x) + j^2 \pmod{TableSize}$

then: $i^2 = j^2 \pmod{TableSize}$

$(i+j)(i-j) = 0 \pmod{TableSize}$

TableSize is prime ➡ either $(i+j)$ or $(i-j)$ is divisible by TableSize

Contradiction !

For any x, it has ⌈ TableSize/2 ⌉ distinct locations into which it can go. If at most ⌊TableSize/2⌋ positions are taken, then an empty spot can always be found.

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.

Read Figures 7.15 - 7.16 for detailed representations and implementations of initialization.

```
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;
}
```

1. 用加法替代平方
2. 用减法替代mod
3. 用移位替代*2

```
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 */
   }
}
```

Question: How to delete a key?

Note: ① Insertion will be seriously slowed down if there are too many deletions intermixed with insertions.
② Although primary clustering is solved, *secondary clustering* occurs – that is, keys that hash to the same position will probe the same alternative cells.

> Deletion直接mark，不真正删除

Double Hashing

$f(i) = i * \text{hash}_2(x)$; /* $\text{hash}_2(x)$ is the 2nd hash function */

ⓘ $\text{hash}_2(x) \neq 0$; ⓘ make sure that all cells can be probed.

☝ Tip: $\text{hash}_2(x) = R - (x \% R)$ 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.

Rehashing

§ 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.

If there are N keys in the table, then $T(N) = O(N)$

Question: When to rehash?

Answer:

① As soon as the table is half full
② When an insertion fails
③ When the table reaches a certain load factor

Note: Usually there should have been $N/2$ insertions before rehash, so $O(N)$ rehash only adds a constant cost to each insertion.

However, in an interactive system, the unfortunate user whose insertion caused a rehash could see a slowdown.

Read Figures 7.23
for detailed implementation of rehashing.