CHAPTER 7

HASHING

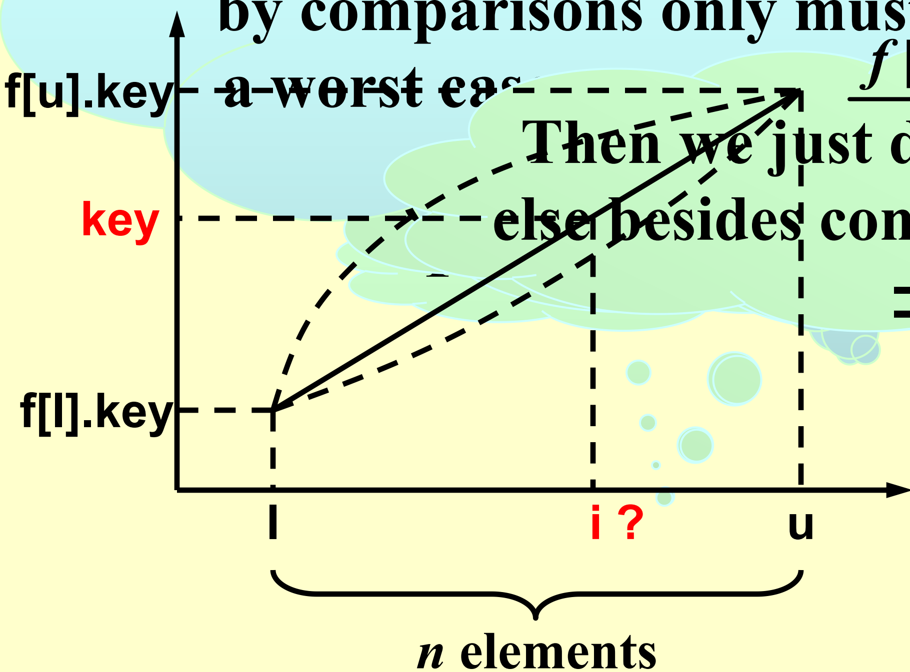
Search by Formula

Interpolation Search :

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

There's a theorem saying that "Any algorithm that sorts by comparisons only must have a worst case ...

Then we just do something else besides comparison.



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

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

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

Else u = i ;

§ 1 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)
- ……

This is the worst disaster in California since I was elected.

California Governor Pat Brown, discussing a local flood

〖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)

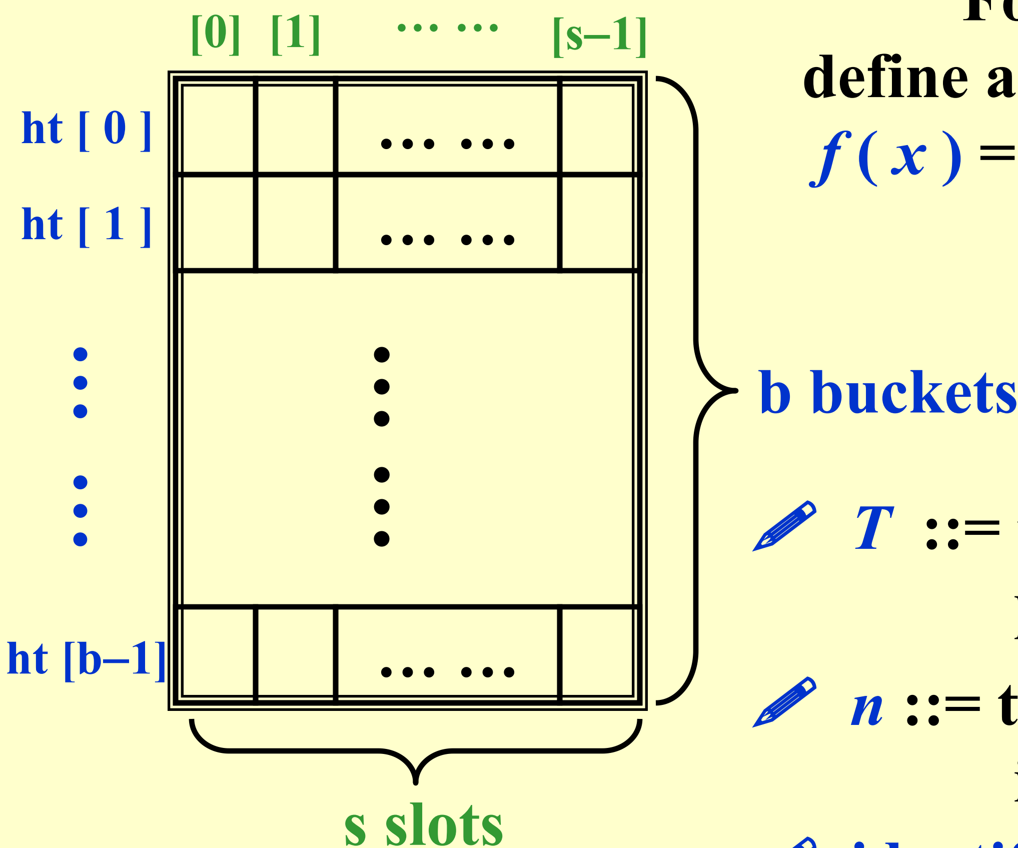
☞ Boolean IsIn(symtab, name)

☞ 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)

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

- 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 and overflow happen simultaneously if s = 1.

〖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 ctime

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

Without overflow,
$$T_{search} = T_{insert} = T_{delete} = O(1)$$

§ 2 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

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

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

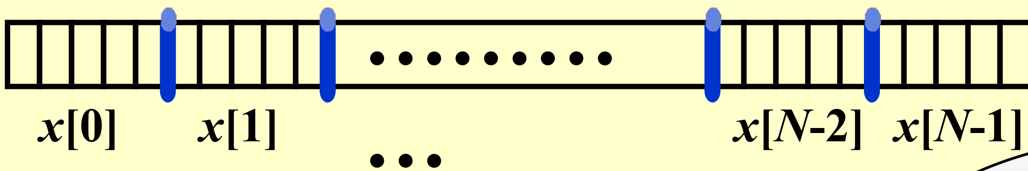
If $x[i] \in [0, 127]$, then $f(x) \in [0,$ **1016** $]$ ☹

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

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

☹ Actual number of combinations < 3000

$$f(x) = \left(\sum x[N-i-1] * 32^i\right) \% \ TableSize \ ;$$




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

Can you make it faster?

Carefully select some characters from x.

☹ 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

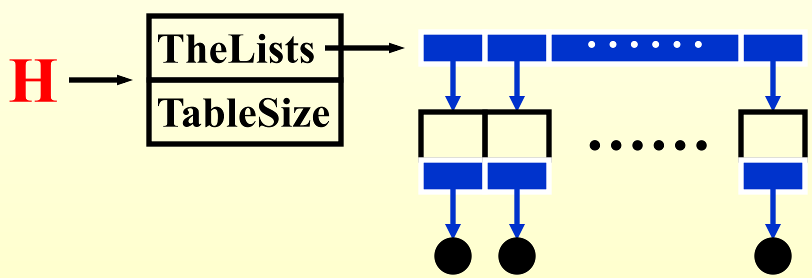
```
struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

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;
}
```

Your hash function

Identical to the code to perform a *Find* for general lists -- List ADT

☞ 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;
            }
    }
}
```

☹ Again?!

Tip: Make the TableSize about as large as the number of keys expected (i.e. to make the loading density factor $\lambda \approx 1$).

§ 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;
          if ( table is full )    break;
          else    i ++;
   }
   if ( table is full )
          ERROR ("No space left");
   else
          insert key at index;
}
```

Collision resolving function. $f(0) = 0$.

Tip: Generally $\lambda < 0.5$.

1. Linear Probing

$f(i) = i$; /* a linear function */

〖Ex…le〗 Mapp… C library functions into a … 6 buckets and s = 1.

…h n is small,

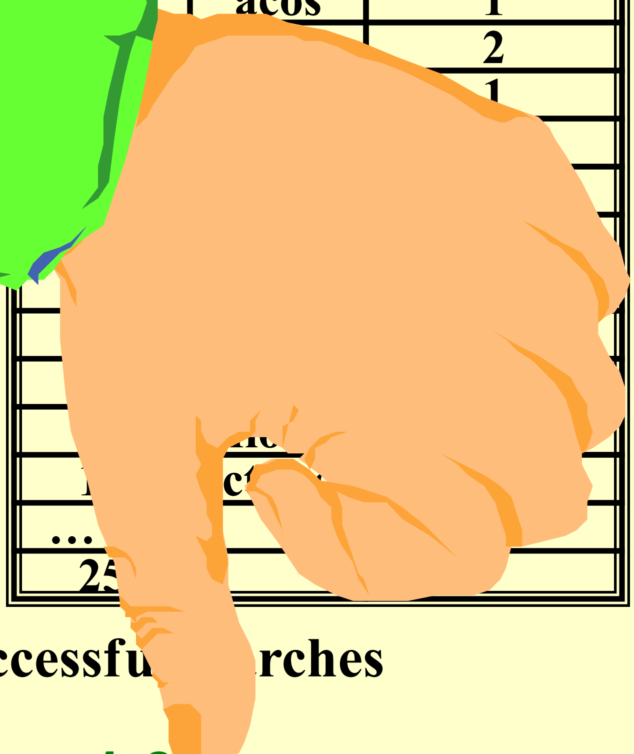
Cause *primary clustering*: any key that hashes into the cluster will add to the cluster after several attempts to resolve the collision.

acos…

ceil…

Loa…

…ket	x	search time
	acos	1
		2
		1
…		
25…		



Average search time = 41 / 11 = 3.73

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

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

= 1.36