國立清華大學 電機工程學系
九十八學年度 第二學期
EE2410 Data Structure

Chapter 8 Hashing

清華大學電機系 黃錫瑜

Outline

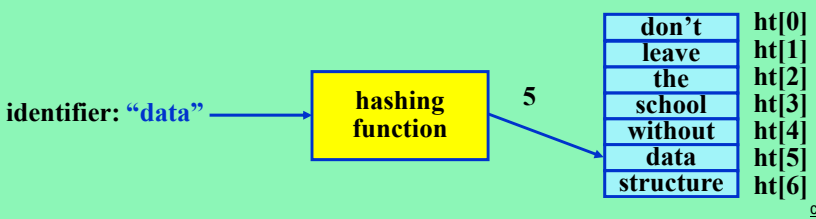
- The Symbol Table ADT
- Static Hashing
  - Hash Table
  - Hashing Function
  - Overflow Handling

Symbol Table

- **Symbol Table**
  - Can be viewed as a a set of name-attribute pairs
  - A form of dictionary
  - Applications include spelling checker, thesaurus, loaders, compilers
- **Common operations on a symbol table**
  - search a particular name in the table
  - retrieve the attributes of that name
  - modify the attributes of that name
  - insert a new name and its attributes
  - delete a name and its attributes

How To Implement Symbol Table?

- **Binary Search Tree**
  - allows efficient search, insert, and delete operation in O(h), where h is the height of the tree
  - Worst case O(n), where n is the total number of identifiers
  - Can be improved to O(log n) → Chapter 10
- **Hash Table**
  - A fixed-size linear array, ht
  - For an identifier, x,
  - The address of x is determined by a hashing function, h(x)



Terminology

- **Bucket and Slot**
  - There are 8 buckets and two slots per bucket in the hash table shown below
- **Identifier density**
  - is the ratio n/T, where
  - n is the number of identifiers in the table
  - T is the total number of possible identifiers
- **Loading factor**
  - is $\alpha = n/(sb)$, where
  - b is the number of buckets, s is the number of slots per bucket
- **Synonyms**
  - Two identifiers, I_1 and I_2 are synonyms if $h(I_1) = h(I_2)$

0	A	A2
1		
2		
3	D	
4		
5		
6		
7	GA	G

A hash table

Collision and Overflow

- **Collision**
  - When two nonidentical identifiers are hashed into the same bucket
- **Overflow**
  - when a new identifier is mapped or hashed by h into a full bucket

hashing function h(x) = 1st character of identifier x

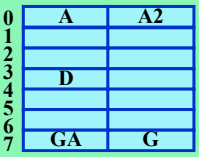
Collision exists at location 0 and 7
Overflow will occur when AA is hashed into the table !

0	A	A2
1		
2		
3	D	
4		
5		
6		
7	GA	G

A hash table

Efficiency of Hash Table

- **Search or insertion time of a hash table**
  - (1) compute the hash function
  - (2) search a bucket
  - The above times are independent of n



A hash table

- **Collision is inevitable**
  - Taking the first character is not a good hashing because of too much collision
    - many variables in a program begins with the same character
  - An overflow mechanism is necessary

Uniform Hash Function

- **Basic desired properties of hash function**
  - Easy to compute
  - The number of collisions is minimized
- **A good hash function**
  - should also depend on every character of an input identifier
- **Uniform hash function**
  - Let x be an identifier chosen at random
  - Then, the probability that h(x) = i is 1/b for every bucket i
  - That is, the hash function does not result in a biased use of the hash table for random inputs

Hash Function (I): Division

hash function $h_D(x) = x \% M$
where % is the modulo operator
That is, the remainder is used as the hash address

6 bits per character

0	0	0	0	0	0	A	1

right-justified zero-filled

6 bits per character

A	1	0	0	0	0	0	0

left-justified

- **Hash address**
  - in the range from 0 through (M-1) → implies that table size is M
- **M should not be a power of 2**
  - otherwise, $h_D(x)$ may depend only on the least significant bits of x
  - E.g., $M=2^3$, then A1→encoded to $2^6(10) + 1$ → $h_D(A1) = 1$
    XY1 → encoded to $2^{12}(33) + 2^6(34) + 1$ → $h_D(A1) = 1$
  - M is usually a prime number

Hash Function (II): Mid-Square

mid-square function $h_m(x)$ = use appropriate # bits from (x^2)

Identifier	Internal Representation	
x	八進位 x	八進位 x^2
A	1	1
A1	134	20420
A2	135	20711
A3	136	21204
A4	137	21501
A9	144	23420
B	2	4
C	3	11
G	7	61
DMAX	4150130	21526443617100
DMAX1	415013034	5264473522151420
AMAX	1150130	135423617100
AMAX1	115013034	3454246522151420

The coding of identifier x is right-justified, zero-filled, and has six bits per character
Table size will be a power of 2

Hash Function(III): Folding

- **Example: x = 12320324111220**
- **Step1: Partition the identifier into several parts**
  - P_1=123, P_2=203, P_3=241, P_4=112, P_5=20
- **Step 2: Add up each part as a hash address**
  - (1) **Shift folding**

$$h(x) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

  - (2) **Folding at the boundaries**

$$h(x) = 123 + 302 + 241 + 211 + 20 = 897$$

reversed

Hash Function (IV): Digit Analysis

- **Application**
  - when all the identifiers are known in advance
- **Procedure**
  - Step 1: Each identifier is interpreted as a number using radix r
  - Step 2: Analyze the distribution of each digit
  - Step 3: Drop biased digits
    - The digits with the most skewed distributions are deleted one by one until the remaining digits is small enough to give an address
- **Example**
  - Given three identifiers in radix-9 form: 891, 792, 793
  - Digit distribution: 1st {8, 7, 7}, 2nd {9, 9, 9}, 3rd {1, 2, 3}
  - The most skewed digits: the 2nd

Outline

- The Symbol Table ADT
- Static Hashing
  - Hash Table
  - Hashing Function
  - Overflow Handling

Overflow handling

- Problem
  - When a new identifier is hashed into a full bucket, then we need to find another open bucket
- Methods
  - linear probing (or linear open addressing)
    - find the closest bucket that is not full
  - chaining
    - implement each bucket as a linked list

Symbol Table Class Definition

```
struct  identifier {
    char  *id;
    int     n;
};

// Assume that operators == and != are defined for identifier
int  operator==(identifier&, identifier&);
int  operator!=(identifier&, identifier&);

class  SymbolTable {
public:
      SymbolTable( int  size = defaultsize ) {
            buckets = size;
            ht = new identifier[buckets];  // linear array as the table
      }
private:
      int  buckets;
      identifier  *ht;
};
```

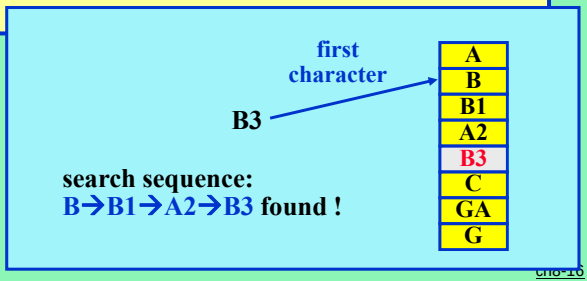

Linear Open Addressing

- **Procedure of searching an identifier x**
  - **Step 1: compute h(x)**
  - **Step 2: examine identifiers at positions ht[h(x)], ht[h(x)+1], ..., ht[(h(x)+j] in this order until one of the following happens:**

    (a) ht[h(x)+j] = x; in this case x is found

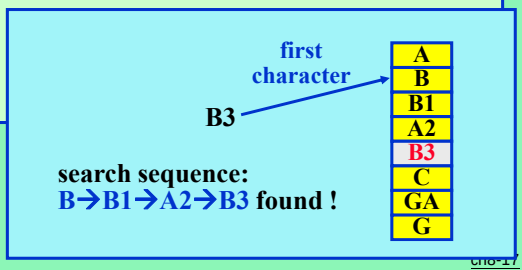
    (b) ht[h(x)+j] is null; x is not in the table

    (c) We return to the starting position h(x); the table is full and x is not in the table



Linear Search

```
int SymbolTable::LinearSearch(
          const identifier& x, int (*hashfunc) (identifier))
// Search the hash table ht ( each bucket has exactly one slot) for x using
// linear probing.
// Return j such that if x is already in the table, then ht[j] = x
// If x is not in the table, return –1
// The hash function "hashfunc" is passed as an argument to LinearSearch
{
  int i = hashfunc(x);
  for ( int j=i; ht[j].id && ht[j] !=x; ) {
       j = (j+1) % buckets;  // treat the table as circular
       if ( j==i ) return –1;  // back to start point
   }
   if ( ht[j] == x)  return j;
   else  return –1;
}
```

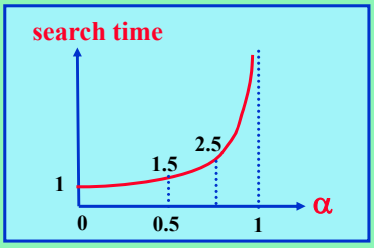


Problem of Linear Open Addressing

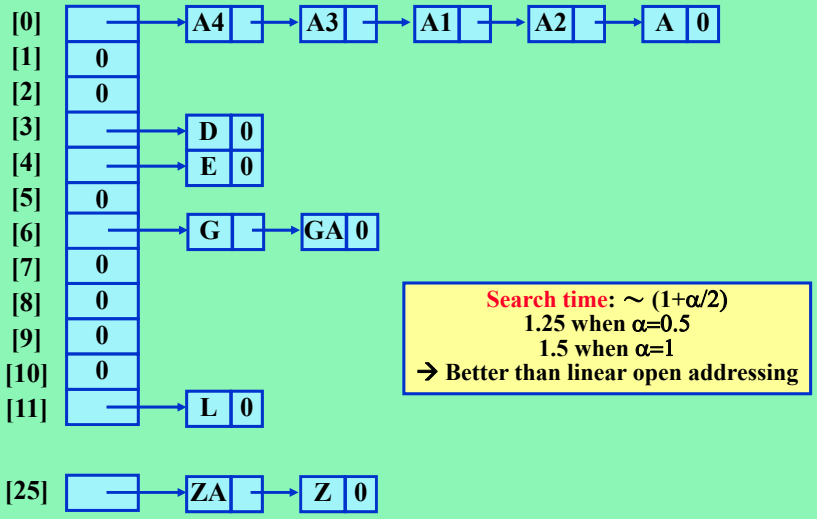
- **Identifiers tend to cluster together**
  - Increase the search time
  - Could be worse than the search tree structure
- **An analysis shows that**
  - It takes $(2-\alpha)/(2-2\alpha)$ to look up an identifier
  - Where α is the loading density
- **Quadratic probing**
  - improve the clustering problem
  - check sequence:

    $(h(x)+i^2)\%b$ and $(h(x)-i^2)\%b$

    i=1, 2, ...

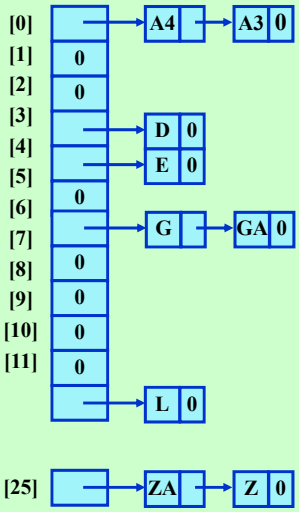


Chaining



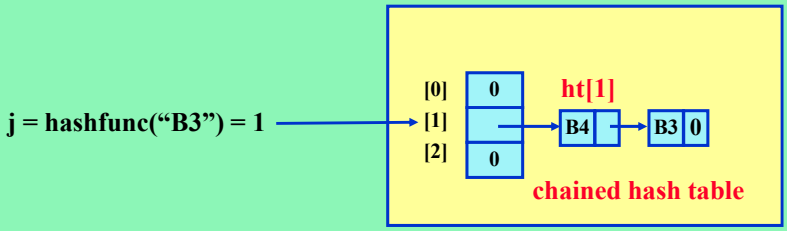
Class Definitions For Chain Search

```
class  ListNode {
friend  SymbolTable;
private:
    identifier  ident;
    ListNode  *link;
};

typedef  ListNode*  ListPtr;

class  SymbolTable {
public:
   SymbolTable (  int  size = defaultsize ) {
        buckets = size;
        ht = new  ListPtr[buckets];
   };
private:
   int  buckets;   // table size
   ListPtr *ht;    // hash table
};
```



Chain Search

```
identifier*  SymbolTable::ChainSearch(const identifier& x, int (*hashfunc)
                                      (identifier) )
// Search the chained hash table ht for x. On termination, return a pointer
// to the identifier in the hashtable. If the identifier does not exist, return 0
{
   int  j = hashfunc(x); // compute headnode address
   // search the chain starting at ht[j]
   for ( ListPtr  node = ht[j]; node; node = node→link )
         if ( node→ident == x)  return &(node→ident);
   return 0;
}
```



A Comparison

- **Hash Function**
  - division is generally superior to the other types
- **Collision handling**
  - Chaining outperforms linear opening addressing

$\alpha = \frac{n}{b}$	0.50		0.75		0.90		0.95	
Hash Function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

(Adapted from V. Lum, P. Yuen, and M. Dodd, *CACM*, 14:4, 1971)

What We Have Learned?

- **Array, Stack, Queue, Linked List**
- **Tree, Graph, Sorting, Hash Table**
- 老鼠走迷宮…
- 尤拉的散步…
- 撲克牌的排序…

Programming is not just coding.
It is all about problem solving !
Good luck on your Journey
As a problem solver !