演算法 搜尋(雜湊)

- 雜湊函數
- 雜湊函數的碰撞問題
- 雜湊搜尋

雜湊法 Hashing

- 雜湊函數 (hash function)
  將任意資料或字串轉換爲較小整數的方法

- 雜湊法: 將資料建立爲雜湊表格 (hash table)
  索引位置 (hash address) 爲輸入資料的雜湊函數

  鍵值爲 { 57, 8, 62, 26, 77, 42 }, N=6，
  除法雜湊函數表示爲 h(x) = x mod M

0	1	2	3	4	5	6	7	8	9	10	11	12
26			42		57			8		62		77

  M=13，資料索引位置 h(x) 等於鍵值的餘數值

除法雜湊函數

- 索引位置等於鍵值的餘數值 h(x) = x mod M
  資料長度 N，表格長度 M (通常取 M>N)

- M 不一定需爲質數，某些雜湊函數
  取 M 爲質數減少碰撞 (collision)

- M 太小引發碰撞問題 (M=7)

0	1	2	3	4	5	6
77, 42	57, 7				26	62

平方取中間位數法

- 鍵值平方取中間 K 位數當作索引位置
  表格大小爲 10^K，如取任意3位數
  x=5762　x^2=33200644　　k=064
  x=2642　x^2= 6980164　　k=016

- 爲避免碰撞，取哪些位數需要先分析

摺疊法

- 位移摺疊法(shift folding)
  鍵值位數折成片段相加，除以 M 取餘數

x = 3812315963 9	381	381
	231	132
h(x) = x' mod M	596	596
	+ 39	+ 93
	x' = 1247	x' = 1202
	mod 101	mod 101
	h(x) = 35	h(x) = 91

- 邊界摺疊法
  鍵值折成片段，間隔將鍵值數字翻轉
  相加除以 M 取餘數

位數分析法 (digital analysis)

- 分析鍵值的每一個位數分佈狀況，挑選分佈均勻的位數
  適用於大量的靜態資料，所有鍵值已知

x

D6	D5	D4	D3	D2	D1	D0
5	8	1	1	2	1	1
5	8	0	1	1	5	3
5	7	9	3	2	3	7
2	8	3	2	2	3	9
5	8	1	3	3	1	8
5	8	0	4	1	3	2
5	7	9	5	2	5	4
5	7	9	5	3	2	5

⇨

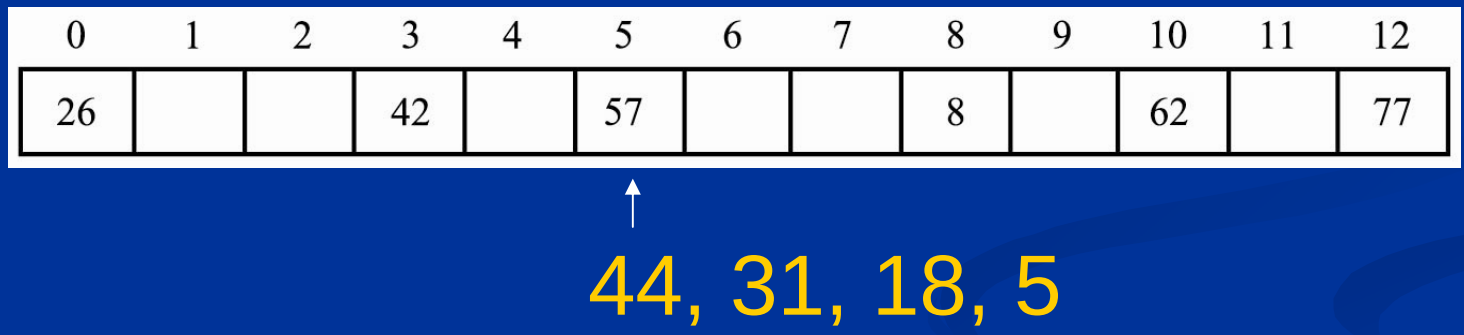
D4	D3	D1	D0
1	1	1	1
0	1	5	3
9	3	3	7
3	2	3	9
1	3	1	8
0	4	3	2
9	5	5	4
9	5	2	5

x'

x=58**11**2**11**　　x'=1111

碰撞問題(Collision)

- 當兩個不同的鍵值經過雜湊函數計算後，落在同一個位置，則稱之爲「碰撞」



- 碰撞解決方案(Collision resolution)
  1) 分離串聯法 (Separate chaining)
  2) 開放式定址法(Open addressing)

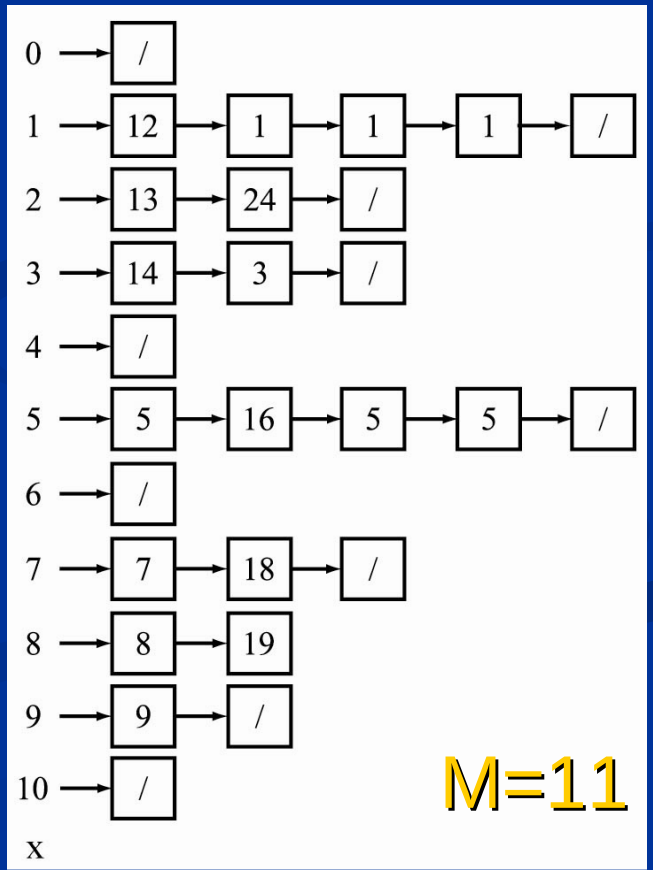
分離串聯法 (Separate chaining)

- 雜湊值碰撞的鍵值連結在一起，儲存在雜湊表格外(open hashing, dynamic hashing)

  輸入{1, 19, 5, 1, 18, 3, 8, 9, 14, 7, 5, 24, 1, 13, 16, 12, 5}

  雜湊函數 h(x) = x mod M

- 動態配置記憶體
  + 串列搜尋

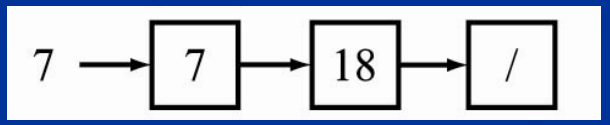


分離串聯法

範例 9-1

```
struct node { struct node *next;    int key};
int M;
int hash(int key){   return key % M;   }

void
add_key(struct node *hash_table[], int key){
    struct node *h;
    h=(struct node *)malloc(sizeof(struct node));
        h->key = key;
        h->next = hash_table[hash(key)];
        hash_table[hash(key)] = h;
}
```



分離串聯法

範例 9-1

製作雜湊表格

```
main(){
   int n=10000; int A[n]; int k;   srand(time(0));
        for(k=0; k<n; k++) A[k]=rand()%n;
   M=n/10;  struct node *h, *hash_table[M];
   for(k=0; k<M; k++) hash_table[k]=NULL;
   for(k=0; k<n; k++)
        add_key( hash_table, A[k]);
   k=rand()%n; printf( "key=%d\n", A[k] );
        h=hash_search(hash_table, A[k]);
   if(h!=NULL) printf( "found %d", h->key );
}
```

雜湊搜尋

雜湊搜尋

```
struct node*
hash_search(struct node* hash_table[], int key) {
    struct node* h=hash_table[hash(key)];
    while (h !=NULL) {
        if (h->key == key) return h;
        h=h->next;
    }
    return NULL;
}
```

- 搜尋直接經由運算得到資料所在的位置
  平均時間複雜度爲 $\Theta(1)$, 最糟時間複雜度 $\Theta(n)$

分離串聯法

- 將 N 個資料均勻分配到 M 個雜湊表格
  平均表格長度 α =N/M, 又稱爲 load factor

  平均失敗搜尋次數 α
  平均成功搜尋次數 1+ α /2

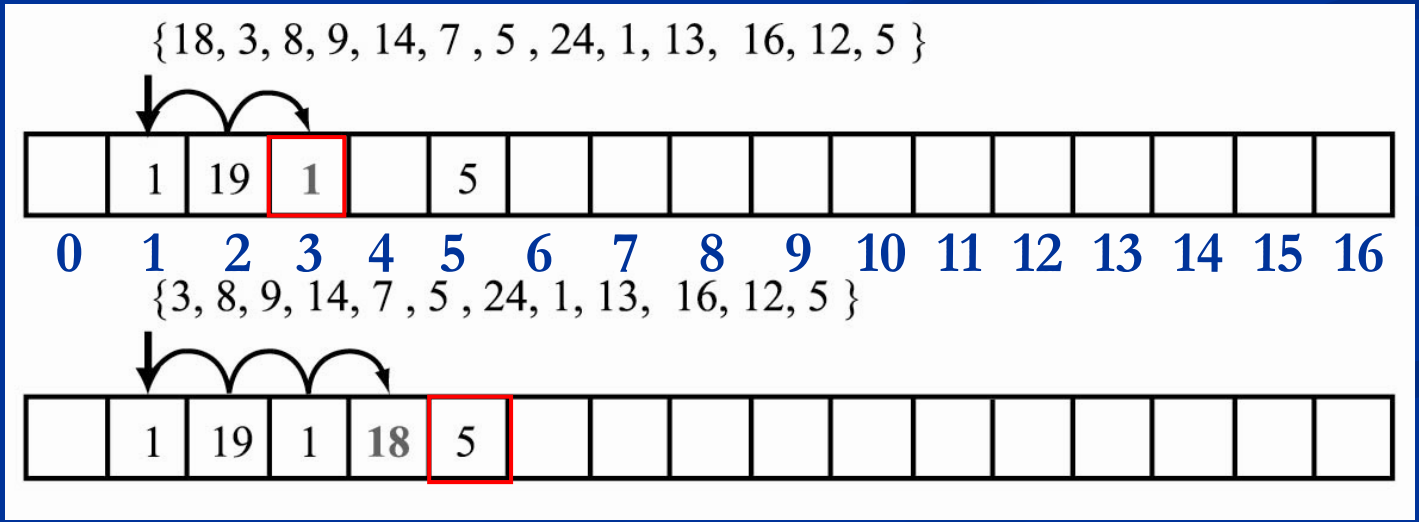
開放式定址法(Open addressing)

- 適用固定陣列

  只使用陣列內的位置(closed hashing, static hashing)

- 線性探測法 (Linear probing)

  當碰撞發生時，就循序在表格中選擇下一個空間

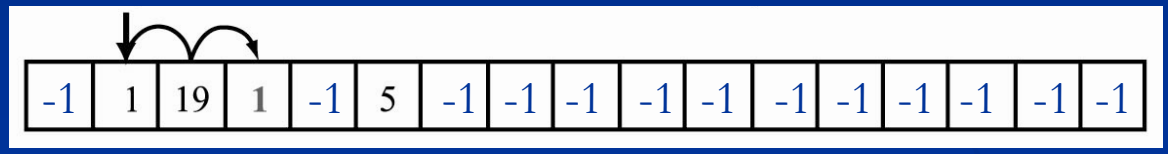


線性探測法

範例 9-2

```
int M;
int hash(int key){   return key % M;   }

void
add_key(int hash_table[], int key){
    int h=hash(key);
    while (hash_table[h] != -1){
        h=hash(h+1);
    }
    hash_table[h] = key;
}
```



線性探測法

範例 9-2

製作雜湊表格

```
main(){
   int n=10000; int A[n]; int k;   srand(time(0));
        for(k=0; k<n; k++) A[k]=rand()%n;
   M=2*n;
        int *hash_table=(int*)malloc(M*sizeof(int));
   for(k=0; k<M; k++) hash_table[k]= -1;
   for(k=0; k<n; k++)
        add_key( hash_table, A[k]);
   k=rand()%n; printf( "key=%d\n", A[k] );
        k=hash_search(hash_table, A[k]);
   if( k != -1) printf( "found %d", A[k]);
}
```

雜湊搜尋

雜湊搜尋

```
int
hash_search(int hash_table[], int key) {
    int h=hash(key), h0=h;
    while ( hash_table[h] != -1) {
        if ( hash_table[h] == key) return h;
        h=hash(h+1);
        if (h==h0) return -1;
    }
    return -1;
}
```

- 搜尋直接經由運算得到資料所在的位置
  平均時間複雜度爲 $\Theta(1)$, 最糟時間複雜度 $\Theta(n)$

線性探測法

- 將 N 個資料均勻分配到 M 個雜湊表格
  平均表格長度 α =N/M (load factor)
  限制 $\alpha \leq 1$

  平均失敗搜尋次數 $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

  平均成功搜尋次數 $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

平方探測法 (Quadratic Probing)

- 線性探測的缺點 → 群聚現象(clustering)

  $h_{next} = (h + 1) \bmod M$

  可寫成

  $h_{next} = (key + c) \bmod M$ c=1, 2 ,3, ...

- 平方探測法

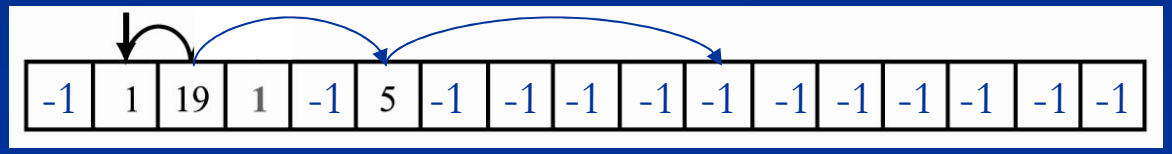
  $h_{next} = (key + c^2) \bmod M$ c=1, 2 ,3, ...

平方探測法

範例 9-3

```
int M;
int hash(int key){   return key % M;   }

void
add_key(int hash_table[], int key){
    int h=hash(key), c=1;
    while (hash_table[h] != -1){
          h=hash(key+c*c); c++;
    }
    hash_table[h] = key;
}
```



雜湊搜尋

```
int
hash_search(int hash_table[], int key) {
    int h=hash(key), h0=h, c=1;
    while ( hash_table[h] != -1) {
        if ( hash_table[h] == key) return h;
        h=hash(key+c*c); c++;
        if (h==h0) return -1;
    }
    return -1;
}
```

- 搜尋直接經由運算得到資料所在的位置
  平均時間複雜度爲 $\Theta(1)$, 最糟時間複雜度 $\Theta(n)$

雙重雜湊法(Double hashing)

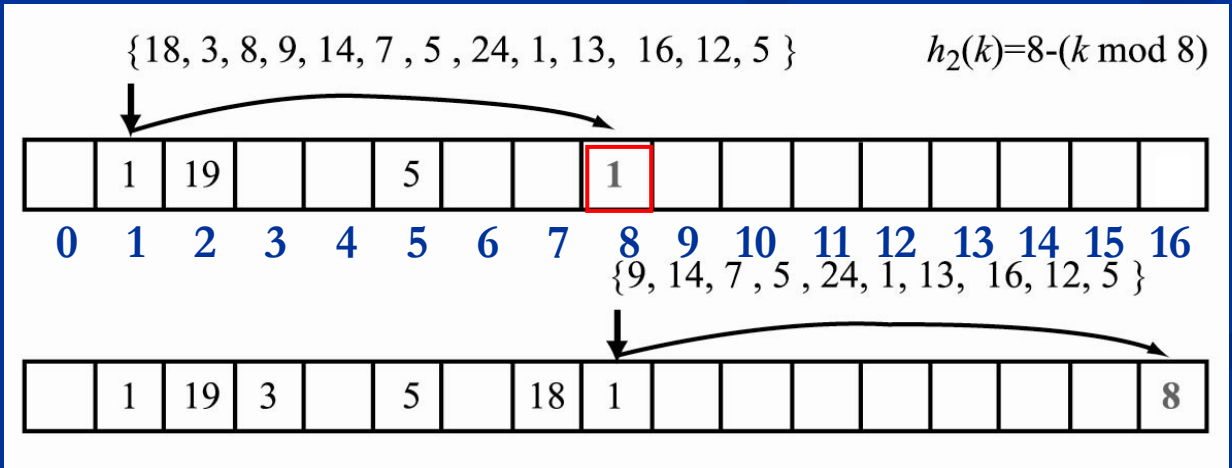
- 雙重雜湊法

  使用另一個雜湊函數選擇下一個空間

$$h_{next} = (key + c\ h_2(key)) \bmod M$$

$$c=1,2,3, \ldots$$

$$h_2(k) = 8 - (k \bmod 8)$$



雙重雜湊法

```
int M;
int hash(int key){   return key % M;   }

void
add_key(int hash_table[], int key){
    int h=hash(key), c=1;
    while (hash_table[h] != -1){
        h=hash(key+c*(8 - key%8)); c++;
    }
    hash_table[h] = key;
}
```

雜湊搜尋

```
int
hash_search(int hash_table[], int key) {
    int h=hash(key), h0=h, c=1;
    while ( hash_table[h] != -1) {
        if ( hash_table[h] == key) return h;
        h=hash(key+c*(8 - key%8)); c++;
        if (h==h0) return -1;
    }
    return -1;
}
```