資料結構與演算法

演算法基本知識

複雜度計算

C++ & STL

Array 陣列

Linked List 鏈串

- Structure
- ADT
- Implement
- Operation
- Application

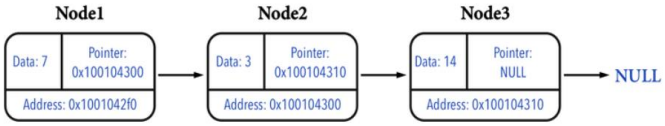
- Structure

  需要熟悉 指標、物件指標

  - Single Linked List:

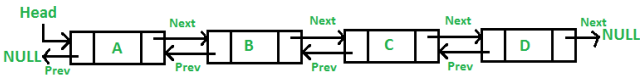
```
struct listnode{
    int data;
    listnode *next;

    listnode(int val){
        next = NULL;
        data = val;
    }
};
```



  - Double Linked List

```
struct listnode{
    int data;
    listnode *next;
    listnode *prev;
};
```



Listnode 是什麼?...結構(物件 class)

Listnode 下面有 int data, 另外有

Stack 堆疊

Queue 佇列

Tree 樹

Binary Tree 二元樹

Binary Search Tree 二元搜尋樹

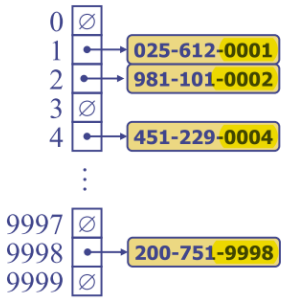
Hash Table 雜湊表

- Intro & Application
- Structure &
- ADT
- Implement
- Practice

- Intro & Application

  Binary Search Tree 提供 Search:O(log(n))
  設計可以O(1) 查詢的 Hash Table

  例如 social security number:
  小資料庫
  Browser caches



- Structure
  Hash Table 由兩個部分組成:
  Hash function $H(x)$ 和 array[N]
  (1). $H(x)$ 由 Hash code 和 Compression function 組成
  (2). Array[N] 可存放 key 和 Data
  Key 用來搜尋， Data 是儲存資料 → key-value pair

```
struct dict{
    int key;
    string value;
    dict(){
        key = 0;
        value = "";
    }
    dict(int Key, string Value){
        key = Key;
        value = Value;
    }
};
```

所謂 key-value pair，
可以用 struct 包成 dictionary

Hash Table 真正難的是在建立 Hash Fuction，和避免 collision。

- 建立 Hash Fuction $H(x)$:
  $H(x)$ 由 Hash code × Compression function 組成
  - Hash Code:
    把 key(any type)轉成 Hash code(int)
    例如: stop
    (1). 用 ASCII $115 + 116 + 111 + 112 = 454$
    但 stop,tops,post 一樣…

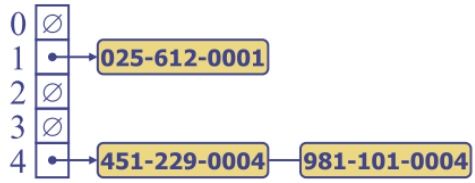
    (2). Polynomial Hash Code:
    Horner's rule:
    例如: csie
    $= 99 + (115 + (105 + 101z)z)z$ 可以自己決定 $(測試)$

  - Compression Function $C(y)$:
    通常有 Division method 和 Multiply, Add & Divide，
    以下提到 Division Method:

    $C(y) = |y|\%N$
    N 代表 Hash Table 的大小$(size)$，請**一定要使用質數**。

- 避免 collision
  當兩組不同的 data 因為各種原因， $H(x_1) = H(x_2) =$ $index$
  兩組數據將存在 array 中的同一個位置，查找刪除時可能會出錯

  兩種解決方法:
  Separate Chaining:
  利用連在 array[index]上的 linked list，將數據串下去



  當資料很大時，我們通常用 separate chaining

Open Addressing:

一個 array[index]，只放一個 data，重複的，找空的塞入。

有幾種子方法

(1).Linear Probing:

$NewH(x) = (H(x) + i)\%N$

i 是什麼 = probing 次數，從 0 開始，失敗就++。

⇒ i越大 效能越差

pro: 這個很簡單，依序找下去

con: 一個塞，接下來幾個都遭殃，其他地方空，這附近狂塞。

狂塞的壞處是什麼: probing 次數爆增，從 o(1) 上升至 O(n)

(像 linear search = =)

依序找，太有規律了，資料必須散的更開一點，效能才會好。

(2).Quadratic Probing:

使用二次方程次擾亂 index

$NewH(x) = (H(x) + c_1(i) + c_2(i)^2)\%N$ $(ex: c_1 = c_2 = 0.5)$

(3).Double Probing:

直接導入第二個 Hash Function 擾亂 index

$\text{NewH}(x) = (x\%N + i * (q - key\%q))\%N$

$x = \text{key}$, $\text{N} = \text{arraySize}(prime)$,

$\text{i} = \text{probing times(from } 0, ++)$,

$q = (0 < q < N)(prime)$

$* \text{GCD}(N, q) = 1$

■ 補充 load factor $\alpha = \frac{n}{N}$ $(n = 填入數, N = arraySize)$

減少 collision

浪費空間

超過 n, hash table 要 resize

- ADT

  尋找:

  string Find(c.key);

  一直 prob 直到

  1.有 key-value 相同的（成功）

  2.有 empty cell 3.

  Return c.value;

  更新:

  void Erase(key);

  Prob 到 1.找到→key,value 歸零 2.沒找到 return NULL

  void Put(key,value);

  throw exception → table is Full

  Prob 到 1.找到空位→key,value 賦值 2.沒成功→exception

- Implement

  - Separate Chain(使用 STL)

    基本屬性:

    Table:

    - 由 vector 組成，接 linked list
    - vector 要規定大小(*size*)

    constructor

    設計 Hash Function

    Put(),Erase(),Find

```
class HashChain{
int size;
vector<list<dict>> table;
HashChain()
HashChain(int m)
int HashFunction(string key_str);
void Put(dict data);
void Erase(string key_str);
string Find(string key_str);
};
```

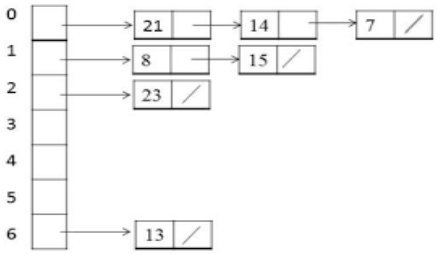
設計 Hash Function

```
int Hashcode(string key_str){ //Horner; rule
      int exp = 9; // z, 自由選擇 或是經過測試選 collision 最少的
      int code = 0, p = 1;
      for(int i = int(key_str.size()-1); i>=0;i--){
         code += key_str[i]*p;
         p *= exp;
      }return code;
   }
   int HashFunction(string key_str){
      return (Hashcode(key_str)% this->size);// Hashcode*Compression function
   }
```

Put(),Erase(),Find() 使用 stl vector, list, iterator

```
void Put (dict data){
   int index = HashFunction(data.key);      // vector<list<dict>> table;
   table[index].push_front(data); }         // list: begin(),front(),push_front/back()...

void Erase (string key_str){
   int index = HashFunction(key_str);
   for(list<dict>::iterator itr = table[index].begin();itr != table[index].end(); itr++){
   // 熟悉迭代器嗎? 看 stl
      if(itr->key == key_str){
         table[index].erase(itr);     // stl::list::erase(const_iterator position)
         break;}
}}
string Find (string key_str){
   int index = HashFunction(key_str);
   for(list<dict>::iterator itr = table[index].begin();itr != table[index].end(); itr++){
      if(itr->key == key_str){
         return itr->value;  // key-value pair, 用 key 去找，回傳 value
         break;
  }}return "...no element.";}
```



- Open Address

  基本屬性:

  Table:

  -由 struct(key,value)組成

  constructor

  設計 Hash Function:

  -linear,quadratic,double

  Put(),Erase(),Find()

  Linear probing 太姆咪

  以下介紹 Quadratic probing 和 Double Hashing

```
class HashOpenAddress{
int size;
dict *table;
HashOpenAddress()
HashOpenAddress(int size){
   table = new dict[size];}
int HashFunction(int key, int i);
void Put(int key, string value);
void Erase (int key);
string Find(int key);
};
```

■ Quadratic Probing

```
int QuadraticProbing(int key, int i)
    return(int(((key%size)+0.5*i+0.5*i*i))%size); //c1 = c2 = 0.5


void Insert(int key, string value){
    bool flag = false; int i = 0;
    while(flag != true && i!=size )
        int index = QuadraticProbing(key,i);
        if(table[index].value==""){ table[index]賦值; flag = true;    break;}
        else    i++;
    if(i==size)    cout << "Hash Table is full"<<endl;}


void Delete(int key){
    int i =0; bool flag = false;
    while(flag != true && i!=size )
        int index = QuadraticProbing(key,i);
        if(table[index].key==key){ table[index]歸零; break;}
        else    i++;}}


string Search(int key){
    int i = 0; bool flag = false;
    while(flag != true && i!=size ){
        int index = QuadraticProbing(key,i);
        if(table[index].key==key)    return table[index].value;
        else    i++;
    return "...not found\n";}
```

■ Double Hashing

```
int DoubleHashing(int key, int i){
    return ((key%size+i*DoubleFunction(key))%size); // size = 17…prime
}
int DoubleFunction(int key){
    int q = 7;            // q is prime && gcd(q,size)=1
    return(q-key%q);
}
```

- Review

  效能:假設資料均勻散開
  (chaining,probing 次數少)⇒ $O(1)$
  效能:假設資料塞在一起
  (chaining,probing 次數多)⇒ $O(n)$

  我們可以提供一個小的 load factor α
  我們可以預計 probing 次數為 1/(1-α) (有數學證明)
  http://cseweb.ucsd.edu/classes/fa14/cse100/lectures/Lec27_before_pdf.pdf

  "In practice, hashing is very fast provided the load factor is not close to 100%"

- Practice

Heap & Priority Queue

搜尋演算法

排序演算法