演算法與程式解題實務

Mong-Jen Kao (高孟駿)

Monday 18:30 – 21:20

使用 C++ 的 vector 資料容器

C++ 的 vector 容器

- 在概念上, vector 容器為加強版的 array,
  「能依實際使用狀況, 自動增加陣列大小」
  - 宣告時, 不須同時決定其大小
  - 任何時間皆可以**動態**在陣列尾巴新增/移除元素, 複雜度 $O(1)$.
  - 裡面的資料, 存放在連續的記憶體空間中

引入 vector 需要的標頭檔

- 跟 C 一樣, 使用 C++ 的 STL 前, 需要先 include 相對應的 header file

```
#include <vector>
```

- 另一個更便利的做法是, 把所有標準函式庫裡的東西全部引入進來

```
#include <bits/stdc++.h>
```

  - 這樣做, 可以把 C 以及 C++ 提供的所有函式庫都一併引入進來

宣告 vector

- vector 是實作了 list / array 的概念的資料容器

  宣告時需要用 < > 加註儲存的資料型態

```
vector<int>  my_int_list;
```

宣告了一個用來儲存 int 的 vector 物件

使用 vector

- vector 是實作了 list / array 的概念的資料容器

```
vector<int>  my_int_list;
```

- 把一筆資料放入 vector 裡 (放到陣列的尾端)

```
my_int_list.push_back( 5 );
```

- 存取 vector 裡的資料 (把它當成陣列來使用)

```
my_int_list[ k ]
```

要注意不能超過當前的陣列索引值範圍

- 取得目前 vector 裡的資料筆數 (list 的長度)

```
my_int_list.size()
```

Pros & Cons

使用 vector 時需要知道的限制

- vector 內部用動態配置的陣列空間來儲存資料
  - 因此, 在 push_back 或是 resize 之後,
    陣列空間的實際位置可能會不一樣！
  - **使用 vector 的 iterator 或指標時, 需特別小心**

- 用 [ ] 來索引 vector 裡的資料時, 要注意不可超過合法的範圍

使用 C++ 的 set 資料容器

前言

- 考慮以下的情境 (需求)
  - 假設我們需要一個能夠達到以下功能的資料容器
    1. 能夠隨時新增/刪除資料
       (capable of storing dynamic data)
    2. 能夠快速判斷一個元素 k 是否在容器裡
       (quick membership query)
    3. 能夠以排序好的順序, 處理容器裡的元素
       (iterate the elements in sorted order)
  - 以我們目前所學過的東西, 要怎麼達到以上需求？
    各個 operation 的效能又分別為何？

方法一 – 使用(未排序的)陣列

- 最原始的方法是使用(未排序的)陣列來儲存資料。

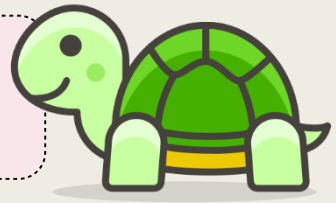
1. 新增/刪除資料

新增資料：O(1)

刪除資料：O(n)

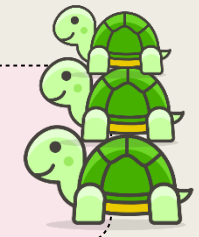
2. 判斷一個元素 k 是否在容器裡

搜尋元素：O(n)



3. 以排序好的順序, 處理容器裡的元素

Iterate in sorted order：$O(n^2)$

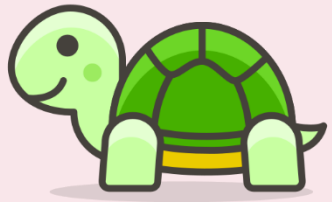


方法二 – 改良版

- 同學會說, 那我們**改成以排序好的順序來儲存資料**。

1. 新增/刪除資料

新增元素： O(n)
刪除元素： O(n)



2. 判斷一個元素 k 是否在容器裡

搜尋元素： O(log n)

3. 以排序好的順序, 處理容器裡的元素

Iterate in sorted order： O(n)

是否有其它更巧妙的做法？

- 事實上, 以我們目前所學過的東西,
  不可能快速又全面地達到原情境的所有功能要求。

- 原因在於：
  - **要能達到功能 3, 那麼資料需要以排序好的順序儲存**
  - **新增/刪除資料, 會改變資料間的排序順序**

是否能同時保有排序順序、並提供動態增刪的功能？

魚與熊掌不可兼得. 但,
若願意在效能上稍做妥協,
則可取得良好平衡！

Store Dynamic Data in Sorted Order

- C++ 的 set 容器, **能以排序好的順序儲存動態資料, 並提供增/刪功能**。在原情境下, **可達到一個理想的效能平衡**。

1. 新增/刪除資料

新增元素： O(log n)

刪除元素： O(log n)

2. 判斷一個元素 k 是否在容器裡

搜尋元素： O(log n)

3. 以排序好的順序, 處理容器裡的元素

Iterate in sorted order： O(n log n)

C++ 的 set 容器

- 以下是使用 set 容器時, 你需要知道的事情：

1. set 容器基於給定的 compare function 運作, **若沒有指定, 則預設使用 <**

2. set 容器, 概念上是模擬數學上的「集合」的概念
   因此, **容器裡同樣的元素只會被儲存一次**。

3. set 的底層, 是一棵平衡的二元搜尋樹 (Balanced Binary Search Tree)
   - **能保証樹的高度為 O(log n)**
   - 預設的實作, 是使用紅黑樹 (Red-Black Tree)
   - **每一筆資料, 是樹裡的一個節點**

C++ 的 set 容器

- 以下是使用 set 容器時, 你需要知道的事情 (續) :

4. set 上絕大多數的 operation, 例如, 新增/刪除/搜尋 等,
   所需的時間全部都是 O(log n)

5. 在概念上, set 是以排序好的順序儲存資料,
   因此, 可用 iterator 的 ++ (或 --) 依序走過所有的資料節點.

   每個 ++ 或 -- 移動所需的時間也是 O(log n)

宣告與使用 set

- 與 vector 一樣, 宣告 set 時,

  需要在後面用 < > 加註儲存的資料型態

```
set<int>  s;
```

宣告了一個可儲存 int 的 set

常用的成員函式

- insert(data), erase(data)
  - 將 data 存入容器; 從容器裡刪除 data
- begin(), end()
  - 傳回 begin / end 的 iterator
- find(data)
  - 搜尋並傳回 data 的節點 iterator; 若找不到, 則傳回 end() 對應的 iterator
- lower_bound(data), upper_bound(data)
  - 在容器裡做 binary search:
    lower_bound 傳回容器裡第一個 $\not<$ data 的節點 iterator;
    upper_bound 傳回容器裡第一個 > data 的節點 iterator

set 的 iterator 操作

- 由於 set 底層實際上是一棵平衡搜尋樹, 因此
  - iterator ++ 或 -- 所需的時間為 O(log n)

範例:

```
auto  it = s.begin();
while(  it != s.end() )
{
  printf("%d ", *it );
  it++;
}
```

此行所需的時間為 O(log n),
故, 印出所有元素整體所需的時間為 O(n log n)

使用 set 搭配自定的 comparator

(自訂 compare function)

C++ 的 set 容器

- C++ 的 set 是強大的工具, 能以「排序好的順序」來儲存資料, 並提供動態新增/刪除的功能。
  - 底層**實際上是一棵平衡的二元搜尋樹 (Balanced Binary Search Tree), 每筆資料都是搜尋樹裡的一個節點** (node)
  - 提供數學上「集合」的概念, **同樣的資料在 set 裡只會被儲存一次**
  - 運作基於給定的 **compare function, 若沒指定, 則預設使用 <**

接下來的目標是學習在 set 裡使用自訂的 comparator

函式物件 (Function object)

- 在 C++ 的 STL 裡,

  指定自訂 comparator 的方式, 是透過傳遞 **函式物件** (Function objects)。

  - 顧名思義, 函式物件是能作為函式來使用的物件。

- sort , lower_bound , upper_bound , set , map ,

  還有 priority_queue 等與排序、搜尋相關的 STL 工具,

  都可以透過指定 custom comparator, 來改變預設的比較基準。

能活用這些工具的話, 在很多時候可以化繁為簡、事半功倍！

(特別是在方法很複雜的場合)

宣告函式物件

- 宣告函式物件的方式很簡單,

  只需要在 struct (或 class) 的宣告裡, 宣告 operator() 函式即可。

```
struct my_cmp
{
      bool operator() ( const int a, const int b ) const
      {
            return a < b;
      }
};
```

宣告了一個函式物件 my_cmp, 可以比較兩個 int 的大小

在 set 裡使用自定的 comparator

- 使用自訂的 comparator 的方式很簡單,

  在宣告 set 時, 一併傳入 function object 即可。

```
struct my_cmp
{
      bool operator() ( const int a, const int b ) const
      {
            return a < b;
      }
};

set< int, my_cmp > s;
```

需注意的是,
函式參數的型態, 必須與 set 儲存的型態相同

在特殊的場合,
compare function 的傳回值
有時會令人覺得混淆

一個容易記的方式是:

「 a 的順序是否必定在 b 前面 」

使用 C++ 的 multiset 資料容器

C++ 的 multiset 容器

- 以下是使用 set 容器時, 你需要知道的事情：

  1. set 容器, 概念上是模擬數學上的「集合」的概念
     因此, **容器裡同樣的元素只會被儲存一次**。

- 若需要存放同樣的元素, 可改用 multiset 容器.
  - 其它的部份與 set 相同

使用 C++ 的 Binary Search 演算法

C++ STL 的 Binary Search 演算法

- C++ 提供了以下兩種型式的 binary search 演算法.

  ForwardIterator
  lower_bound (ForwardIterator first, ForwardIterator last, value)

  - 在 first 到 last -1 的範圍間搜尋第一個 >= value 的元素,
    並傳回該元素的 iterator.
  - 此函式的行為,
    跟前半部投影片裡的 lower_bound 函式的虛擬碼相同.

C++ STL 的 Binary Search 演算法

- C++ 提供了以下兩種型式的 binary search 演算法.

  ForwardIterator
  upper_bound (ForwardIterator first, ForwardIterator last, value)

  - 在 first 到 last -1 的範圍間搜尋第一個 > value 的元素,
    並傳回該元素的 iterator.
  - 此函式的行為,
    跟前半部投影片裡的 upper_bound 函式的虛擬碼相同.

使用 C++ 的 pair

前言

- C++ 提供了一個與 STL 工具相容, 方便使用的複合資料型態 **pair**
  - 顧名思義, 在概念上它提供了 pair 的功能
    - 可以將它理解為下列的 template 宣告

```
template< class T1, class T2 >
struct pair
{
        T1 first;
        T2 second;
}
```

可以 google 參考 cpp reference !

  - 除此之外, pair 裡實作了指定運算子與所有的關係運算子,
    能與所有 STL 工具完美結合

使用範例

- 用 pair 及 vector 來儲存平面上的 n 個點, 再把它們 print 出來

```
vector< pair<int,int> > points;

for( int i = 0; i < n; i++ )
{
    scanf("%d%d", &j, &k);
    points.push_back( { j, k } );
}

for( int i = 0; i < n; i++ )
    printf("%d %d\n", points[i].first, points[i].second);
```

使用 C++ 的 map 關聯容器

(Associative container)

前言

- 在程式中, 我們時常會需要做對應轉換, 例如:
  - (實習課 W1) 姓名 => 編號
  - (數字推盤遊戲) 盤面 => 圖上的節點編號

  這一類的對應所需的時間, 往往直接影響到程式整體的效能。

- **C++ 的 map 是一個關聯容器 (Associative container),**
  功能是提供高效率的 key => mapped value 關聯性對應
  - 容器內儲存的, 是一對對的 { key, value } 的 pair
  - 可以把它看成是函數對應, 餵給它 key, 它會回傳其所對應的 value

前言

- C++ 的 map 是一個關聯容器 (Associative container),
  功能是提供高效率的 key => mapped value 關聯性對應
  - 與 set 相同,
    map 容器的底層, 實際上是一棵平衡的二元搜尋樹,
    其中**每個節點內分別儲存一對 { key, value } pair**
    並且**搜尋樹以 key 為搜尋的基準**。

搜尋關聯的時間為 **O(log n)**

新增/刪除關聯的時間也是 **O(log n)**

宣告與使用 map

- 宣告 map 時, 在後面用 < > 加註要做對應的資料型態

```
map< string, int >  m;
```

宣告了一個可以把 字串 對應到 int 的 map

使用 map

要注意的是,
這裡所有的索引動作, 所花的時間都是 **O(log n)**

- map 的使用方式很單純, 用 [ ] 運算子來做關聯性的索引即可。

```
map< string, int >  m;
```

宣告了一個可以把 字串 對應到 int 的 map

```
m["Amy"] = 1;
m["John"] = 2;
```

將 “Amy” => 1 以及
“John” => 2 兩組對應資料放入容器 m 裡

```
m["Amy"] = 3;
```

將 “Amy” 所對應的值更新為 3

```
printf("%d\n", m["Amy"]);
```

使用 map

- 除了用 [ ] 運算子做關聯性的索引外,
  也可以用 find(key) 函式來取得搜尋樹裡的節點的 iterator 。

```
map< string, int >  m;

m[“Amy”] = 1;
m[“John”] = 2;

auto it = m.find(“Amy”);
it->second = 3;

printf(“%s %d\n”, it->first.c_str(), it->second);
```

取得 “Amy” 所對應的節點 iterator 後,
將它所對應的值更新為 3

常用的成員函式

- 使用 [] 運算子做關聯性索引
- erase(key)
  - 從容器裡刪除 key 所對應的關聯
- begin(), end()
  - 傳回 begin / end 的 iterator
- find(key)
  - 搜尋並傳回 key 對應的節點 iterator; 若找不到, 則傳回 end() 對應的 iterator
- lower_bound(key), upper_bound(key)
  - 在容器裡做 binary search:
    lower_bound 傳回容器裡第一個 $\not<$ key 的節點 iterator;
    upper_bound 傳回容器裡第一個 > key 的節點 iterator