【第十講】

搜尋與雜湊

講師: 李根逸 (Ken-Yi Lee), E-mail: feis.tw@gmail.com

課程大綱

- 各種迭代器提供的操作
- 線性搜尋法
  - 對容器做線性搜尋
  - 對範圍做線性搜尋
- 二元搜尋法
- 使用 **STL** 的搜尋
  - 線性搜尋 (std::find)
  - 二元搜尋 (std::lower_bound, std::binary_search)
- 雜湊 **(hash)**
  - 使用陣列實作雜湊
  - 串列雜湊表格
- **STL** 雜湊的集合與映射

是否支援的性質	單向迭代器 (forward_iterator)	雙向迭代器 (bidirectional _iterator)	隨機存取迭代器 (random_access_iterator)
預設建構式、複製建構式與指定運算子	支援	支援	支援
operator!=, operator== (相等性比較)	支援	支援	支援
operator*, oeprator->(存取元素內容)	支援	支援	支援
operator++(進一)	支援	支援	支援
operator-- (退一)	不支援	支援	支援
operator- (兩迭代器相差的距離)	不支援	不支援	支援
operator+ (進 N) **operator-** (退 N)	不支援	不支援	支援
operator>, operator<, operator>=, operator<= (比較大小)	不支援	不支援	支援
operator[] (存取相對元素)	不支援	不支援	支援
提供的容器	**std::forward_list**	**std::list**	**std::vector** **std::deque** **std::array**

<iterator> 標頭檔

- 迭代器相關的函式：
  - advance: 前進或後退 N 步
    - 隨機存取迭代器：operator+ 或 operator-
    - 單向或雙向迭代器：使用迴圈做 operator++ 或 operator—
  - distance: 計算兩個迭代器的距離
    - 隨機存取迭代器：operator-
    - 單向或雙向迭代器：使用迴圈做 operator++

對容器做線性搜尋

- 最基本的搜尋演算法就是循序將容器內所有元素一一比對：
  - 輸入要搜尋的陣列 c 及要找的值 v
  - 回傳在陣列中找到的值編號：

```
int Find(const vector<int> &c, int v) {
  int i = 0;
  while (c[i] != v && i < c.size()) {
    i++;
  }
  return i;
}
```

當找不到時要回傳的編號是？

對範圍做線性搜尋 [1]

- 我們改成只對容器內某個連續範圍做線性搜尋：

```
int Find(const vector<int> &c,
         int first,
         int last,
         int v) {
  int i = first;
  while (c[i] != v && i < last) {
    i++;
  }
  return i;
}
```

當找不到時要回傳的編號是？

不需要知道容器的大小？

Ans: 有可能last, first，不在vector的size範圍內。

對範圍做線性搜尋 [2]

- 利用指標來傳遞範圍：

```
int * Find(int *first,
           int *last,
           int v) {
  int *p = first;
  while (*p != v && p < last) {
    ++p;
  }
  return p;
}
```

當找不到時要回傳的指標是？

使用指標的優缺點？

Ans: 可確定指標是否溢位，且控制size大小？缺點..

【範例】對範圍做線性搜尋 [3]

- 使用函式模版與迭代器：

```
template<class Iterator, class Type>
Iterator Find(Iterator first,
              Iterator last,
              Type v) {
  Iterator p = first;
  while (*p != v && p !=last) {
    ++p;
  }
  return p;
}
```

p < last ?

單向迭代器？

當找不到時要回傳的迭代器是？

使用迭代器的優缺點？

Ans:擴增至不同容器的搜尋，且程式碼一致。

[範例] find.cpp

有序型容器

- 如果對於容器內的元素排列沒有任何假設，則每次使用線性搜尋所需要花的時間與容器內的元素個數成正比！
- 但是我們其實可以讓容器內的元素具有某種順序：
  - 我們可以讓容器在新增或刪除元素時保持有順序的狀態 (參考前面章節的練習)
  - 我們也可以對容器作排序 (sort) 讓他們具有順序
- 使用有序型的容器可以讓我們更容易做搜尋嗎？

二元搜尋法 [1]

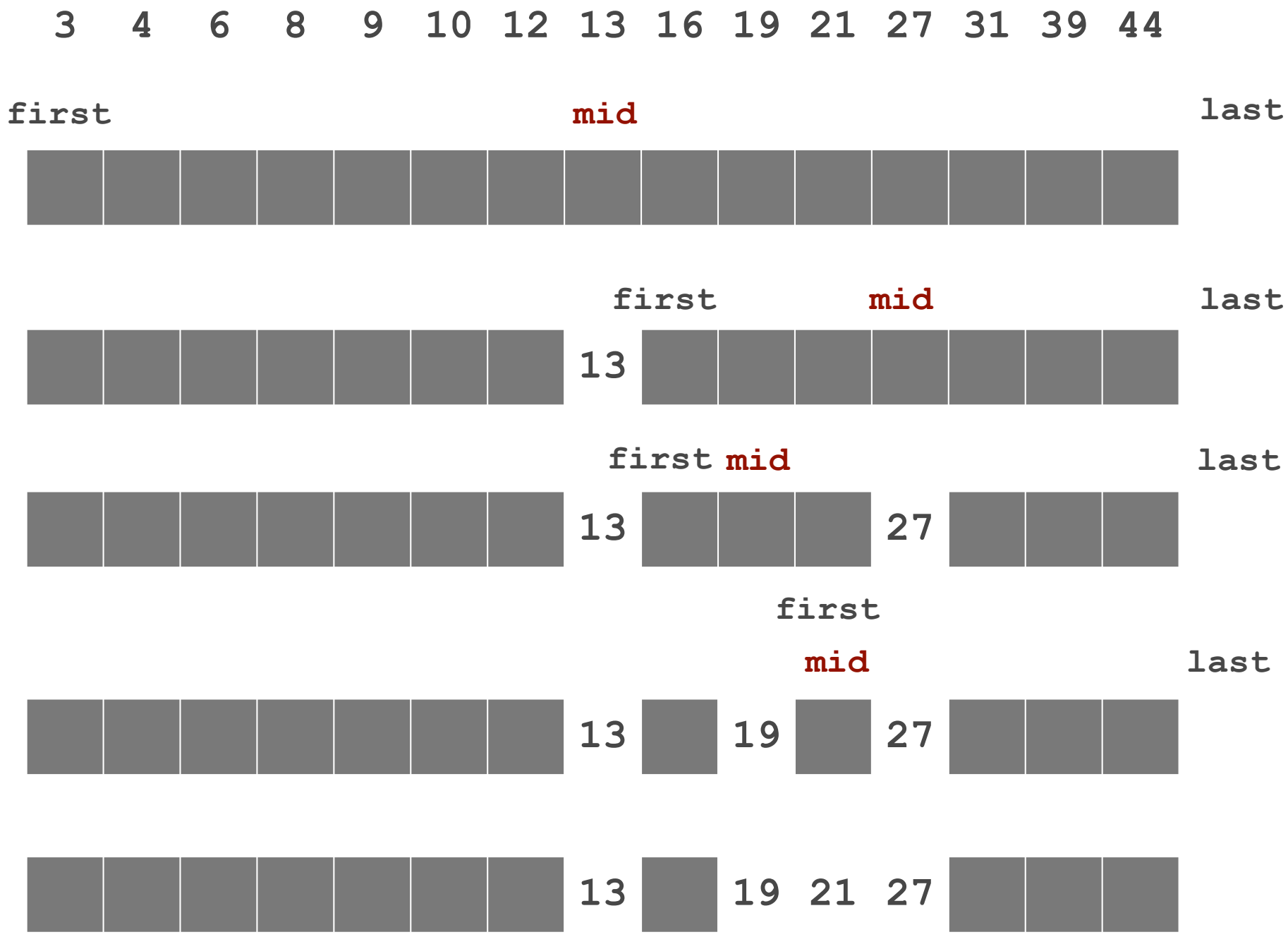
```
template <class Iterator, class Type>
Iterator LowerBound(Iterator first, Iterator last,
                    const Type &val) {
  int dist = last-first;
  Iterator mid;
  while (dist > 0) {
    mid = first;
    int step = dist / 2;
    mid += step;
    if (*mid < val ) {
      first = ++mid;
      dist -= step + 1;
    } else {
      dist = step;
    }
  }
  return first;
}
```

二元搜尋法可以對已經排序好的容器作快速搜尋

不懂

需要提供隨機存取迭代器

LowerBound(first, last, 21)



【範例】二元搜尋法 [2]

```
template <class Iterator, class Type>
Iterator LowerBound(Iterator first, Iterator last,
                    const Type &val) {
  int dist = distance(first, last);
  Iterator mid;
  while (dist > 0) {
    mid = first;
    int step = dist / 2;
    advance(mid, step);
    if (*mid < val ) {
      first = ++mid;
      dist -= step + 1;
    } else {
      dist = step;
    }
  }
  return first;
}
```

distance() 計算兩個迭代器的距離
advance() 可移動迭代器

只要提供單向迭代器

[範例] binary_search.cpp

使用 STL 的搜尋

- 我們可對容器內容做線性搜尋：
  - 在 C++ STL 的 **<algorithm>** 內有 **find** 函式可以做線性搜尋
- 我們可對已經排序好的容器做二元搜尋：
  - 在 C++ STL 的 **<algorithm>** 內有 **lower_bound** 與 **binary_search** 函式可做二元搜尋
  - 二元搜尋法使用到的資料結構特性：
    - 已經排序好
    - 適合隨機存取

陣列適合使用二元搜尋法嗎？

串列適合使用二元搜尋法嗎？

二元搜尋法效率與元素個數有關嗎？還有更好的嗎？

【範例】雜湊

- 二元搜尋的時間複雜度雖然比線性搜尋低，但是還是跟元素個數有關，我們有辦法讓時間複雜度更低嗎？
- **雜湊 (hash) 可以讓搜尋時間複雜度更接近常數時間，也就是不會因為元素個數增加而增加搜尋時間**
- **雜湊表格 (hash table) 使用一個固定大小的陣列去儲存資料，但是儲存的索引由雜湊函數決定**
  - 雜湊函數：將每個元素值對應到一個整數值

使用陣列實作雜湊容器 [1]

- 使用一個可變大小的陣列

```
vector<ElemType> > table;
```

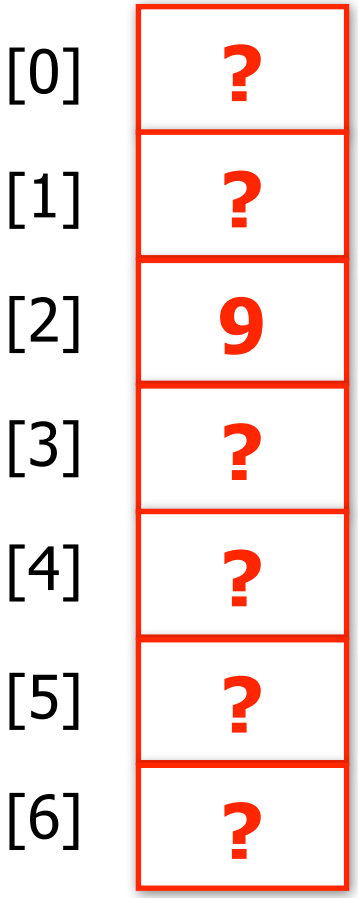
  - 插入元素時，使用雜湊函數將插入的值對應到陣列的編號
  - 搜尋元素時，使用雜湊函數將搜尋的值對應到陣列的編號，然後檢查陣列的元素值是否與搜尋的值相等。相等代表找到，而不相等代表沒找到

雜湊函數： `h(v) = v % 7`

```
table.Insert(v) := table[h(v)] = v;

table.Insert(9);
table.Find(9);
table.Find(4);
```

怎麼保證 **table.find(4)** 不會找到？



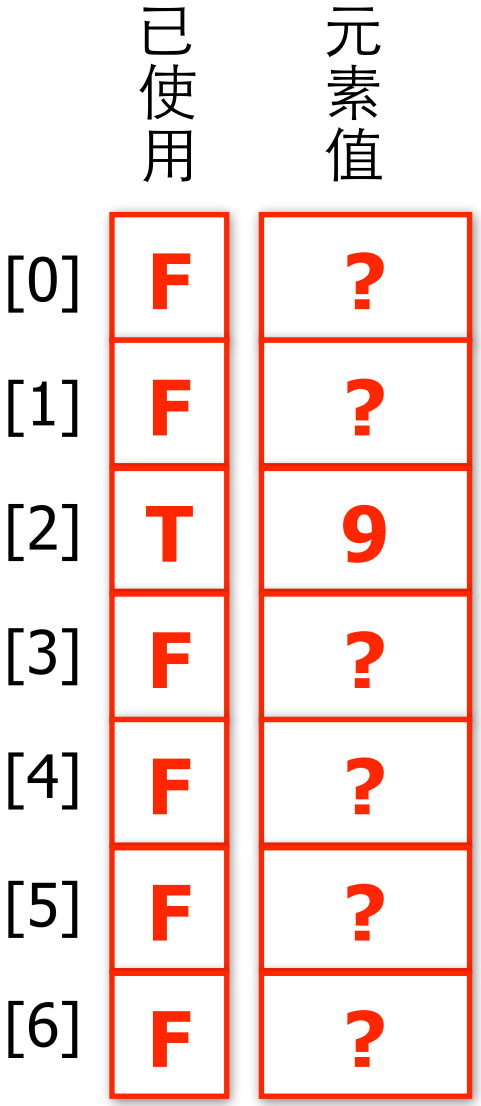
使用陣列實作雜湊容器 [2]

- 使用一個可變大小的陣列
  - 陣列裡每個格子存兩個資訊：
    - 是否已經使用? (bool)
    - 元素值 (ElemType)

```
vector<pair<bool, ElemType> > table;
```

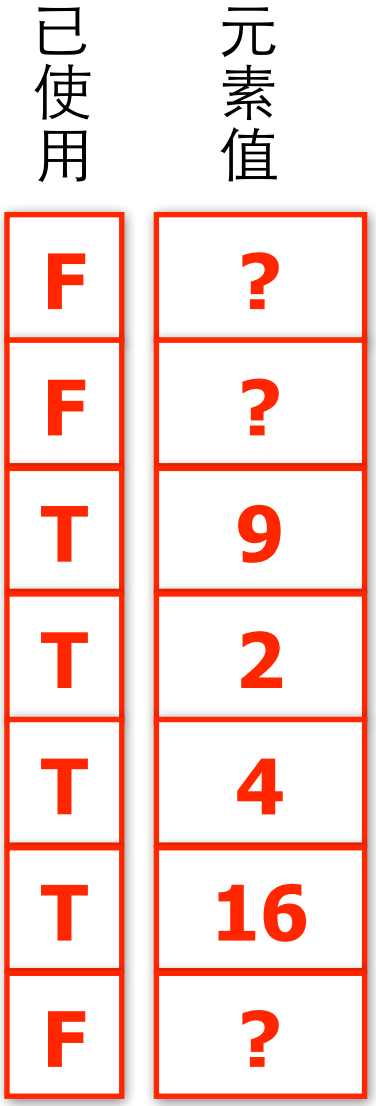
  - 插入元素時，使用雜湊函數將插入的值對應到陣列的編號，然後將元素值置入陣列同時將『已使用』設為 true
  - 搜尋元素時，使用雜湊函數將搜尋的值對應到陣列的編號，然後檢查陣列內『已使用』是否為 true，再比較元素值是否與搜尋的值相等。相等代表找到，而不相等代表沒找到

[碰撞] 如果兩個值對應到同樣的索引怎麼辦？



使用陣列實作雜湊容器 [3]

- 使用一個可變大小的陣列
  - 陣列裡每個格子存兩個資訊
  - 插入元素時，使用雜湊函數將插入的值對應到陣列的編號。檢查要插入的位置是否已被使用，如果已被使用就找下一個可能的位置(例如往下一格) 做一樣的檢查，直到找到未被使用的位置後將元素值置入陣列同時將『已使用』設為 **true**
  - 搜尋元素時，使用雜湊函數將搜尋的值對應到陣列的編號。檢查陣列內的『已使用』是否為 **true**，是的話再比較元素值是否與搜尋的值相等。如果不相等的話要繼續往下一個可能的位置做一樣的檢查，直到該格子是未被使用的位置才能停



依序插入 9, 4, 2, 16

[連續性] 要怎麼刪除一個值？

【範例】使用陣列實作雜湊容器 [4]

- 使用一個可變大小的陣列
  - 陣列裡每個格子存兩個資訊：
    - 使用狀態 **(int)**：
      - 0: 未使用過, 1: 已使用, -1: 曾經使用但被刪除
    - 元素值 **(ElemType)**
  - 插入元素時，先用雜湊函數對應到陣列的編號後，找到可插入的位置。可插入的位置包含『未使用過』還有『曾經使用但被刪除』
  - 搜尋元素時，先用雜湊函數對應到陣列的編號後，依序檢查是否是要找的元素。一直找到『未使用過』的格子後才能停止。
  - 刪除元素時，先用雜湊函數對應到陣列的編號後，找到要刪除的元素，接著將使用狀態設定為『曾經使用但被刪除』

	使用狀態	元素值
[0]	0	?
[1]	0	?
[2]	1	9
[3]	1	2
[4]	-1	4
[5]	1	16
[6]	0	?

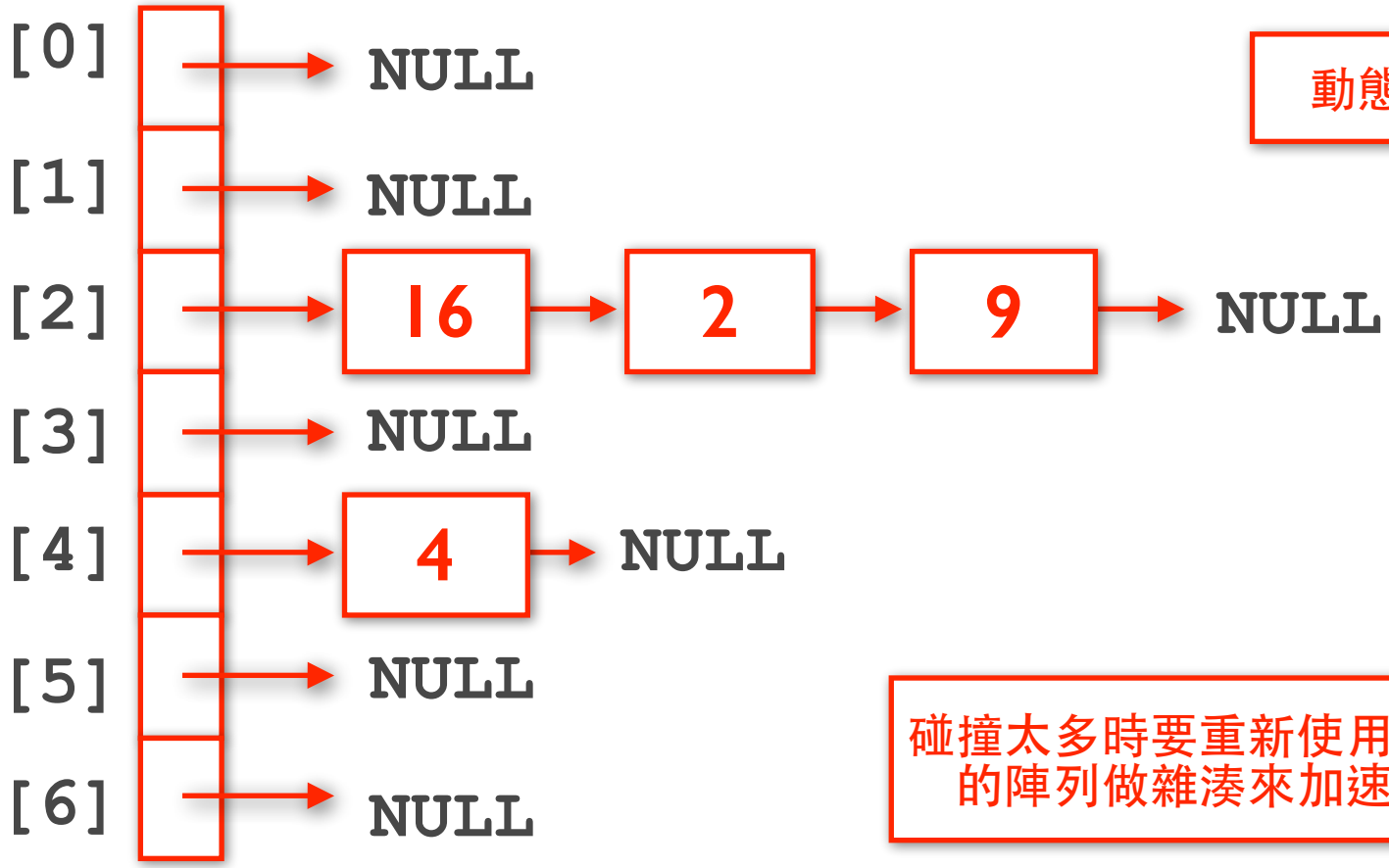
依序插入 9, 4, 2, 16 後刪除 4

如果沒有未使用過的的格子怎麼辦？

[範例] hash.cpp

【範例】串列雜湊表格

```
vector<list<ElemType> > table;
```



動態刪除

碰撞太多時要重新使用比較大的陣列做雜湊來加速搜尋

依序插入 9, 4, 2, 16

[範例] linked_hash.cpp

STL 的雜湊

- 在 **C++03** 以前, **STL** 並沒有公開的雜湊相關容器或類別。在 **C++11** 以後, **STL** 提供了下列的模版作為雜湊容器或類別使用：
  - **hash** 類別模版：用來作為雜湊函式物件類別模版
    - 可以產生常見型態的平均分布雜湊函式

```
hash<string> h;

cout << h("Hello") << endl;
cout << h("world") << endl;
```

  - **unorder_set** 類別模版：用雜湊做的集合容器
  - **unorder_map** 類別模版：用雜湊做的映射容器

【補充】在 Dev C++ 使用 C++11

- 在 **Dev C++** 內開啟 **C++11** 的功能：
  - Tools > Compiler Options > Settings > Code Generation > Language standard > 選擇 ISO C++11 或 GNU C++ 11
  - ISO C++11 指的就是 C++11 的標準版，而 GNU C++11 則還加上了 GNU 編譯器對 C++11 的一些擴充功能。

集合與映射容器

- **集合的概念在於集合容器內的每個元素都只會出現一次，當你插入與某已有元素比較結果為相同的新元素時，集合並不會變大。**
  - 例如想統計一篇文章中出現過多少種字，我們就可以將每個字都插入到集合容器中，看最後容器大小就知道有多少種不同的字
- **映射的概念比集合多了一些資訊：映射容器內每個元素除了用來做比較的鍵值 (key) 外，還可以附帶一個內容值 (value)。**
  - 例如我們可以用電話號碼做鍵值，用該電話的擁有者資料做內容值。之後我們可以快速使用電話號碼來搜尋電話的擁有者資料
- **集合跟映射都是種可以用來作快速搜尋的容器**

STL 雜湊的集合與類別

- **C++11 內的雜湊容器有兩種：**
  - **<unordered_set>** 內的 **unordered_set** 類別模組：無序型集合容器
  - **<unordered_map>** 內的 **unordered_map** 類別模組：無序型映射容器
- **因為是雜湊容器，使用迭代器依序印出這兩個容器內的元素值時，並不會依照值大小順序。**
  - 下一章會介紹 **<set>** 與 **<map>** 是有序型的
- **每一種鍵值都只會出現一次，不會有兩個元素值比較結果是相等的。**
  - 當你插入已有的值時，是不會有任何效果

	線性搜尋 **(linear search)**	二元搜尋 **(binary search)**	雜湊 **(hash)**
相關函式	**std::find,** **std::find_if**	**std::lower_bound,** **std::upper_bound,** **std::binary_search**	**std::hash** (函式類別模版)
適用容器	一般容器： **std::array** **std::vector** **std::deque** **std::forward_list** **std::list**	排序後的一般容器 (如左格)	特殊容器： **std::unordered_set,** **std::unordered_map**
搜尋時間	與元素個數成正比	排序的成本 + 與元素個數的對數成正比	可能與元素個數無關，但 與碰撞程度相關
插入新增或修改元素時間	與使用容器相關 (無額外負擔)	與使用容器相關 + 維持排序的成本	可能與元素個數無關，但 可能需要重新配置雜湊 (與個數成正比)

使用二元搜尋樹