稀疏的對照表 (續)

- 平行陣列 (續)

稀疏的對照表 (續)

- 平行陣列 (續)

  快速轉換：用「二分搜尋法」

稀疏的對照表 (續)

- 平行陣列 (續)

  快速轉換：用「二分搜尋法」

  - C 的 **stdlib.h** 裡面的 **bsearch()**

```
1 #include <stdlib.h>
2 int comp(const void *a, const void *b) { return *(int*)a-*(int*)b; }
  …
3 int d, d0, d1, d2, d3, m=6, idx[]={2,5,7,8,12,13}, *ptr;
4 char tab[]="CABEFE";
  …
5 d = d0*8 + d1*4 + d2*2 + d3;
6 if ((ptr=(int*)bsearch(&d, idx, m, sizeof(int), comp))!=NULL)
7     printf("%c", tab[ptr-idx]);
```

稀疏的對照表 (續)

- 平行陣列 (續)

  快速轉換：用「二分搜尋法」
  - C 的 **stdlib.h** 裡面的 **bsearch()**
  - C++ 的 **algorithm** 裡面的 **lower_bound()**

```
1 #include <algorithm>
2 using namespace std;
  …
3 int d, d0, d1, d2, d3, m=6, idx[]={2,5,7,8,12,13}, *ptr;
4 char tab[]="CABEFE";
  …
5 d = d0*8 + d1*4 + d2*2 + d3;
6 if ((ptr=lower_bound(idx, idx+m, d))&&(*ptr==d))
7     printf("%c", tab[ptr-idx]);
```

稀疏的對照表 (續)

- 平行陣列 (續)

  快速轉換：用「二分搜尋法」

  - C 的 **stdlib.h** 裡面的 **bsearch()**
  - C++ 的 **algorithm** 裡面的 **lower_bound()**
  - 自己撰寫等效的 **lower_bound()**

```
01 int *lower_bound(int *begin, int *end, int target) {
02     int mid, *result=end;
03     while (begin<end) {
04         mid = (end-begin)/2;
05         if (*(begin+mid)>=target)
06             result = end = begin+mid;
07         else
08             begin += mid+1;
09     }
10     return result;
11 }
   …
12 int d, d0, d1, d2, d3, m=6, idx[]={2,5,7,8,12,13}, *ptr;
13 char tab[]="CABEFE";
   …
14 d = d0*8 + d1*4 + d2*2 + d3;
15 if ((ptr=lower_bound(idx, idx+m, d))&&(*ptr==d))
16     printf("%c", tab[ptr-idx]);
```

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼也會很精簡，參加 APCS 檢測的話是值得練習的工具

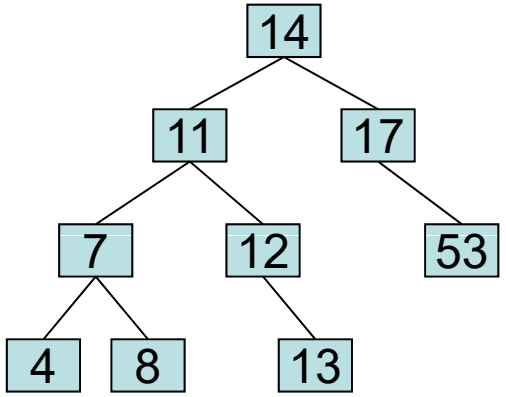
稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  表示所有的索引值



稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

"0 0 1 0"	'C'
"0 1 0 1"	'A'
"0 1 1 1"	'B'
"1 0 0 0"	'E'
"1 1 0 0"	'F'
"1 1 0 1"	'D'

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

```
1 #include <map>
2 #include <string>
3 #include <cstdio>
4 using namespace std;
5 map<string, char> decoder =  { {"0 1 0 1", 'A'}, {"0 1 1 1", 'B'},
6       {"0 0 1 0", 'C'}, {"1 1 0 1", 'D'}, {"1 0 0 0", 'E'}, {"1 1 0 0", 'F'} };
  …
7 char buf[10];
8 fgets(buf, sizeof(buf), stdin); buf[7] = 0;
9 printf("%c", decoder[buf]);
```

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼
  也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

```
1 #include <map>
2 #include <string>
3 #include <cstdio>                                  decoder["0 1 0 1"] = 'A';
4 using namespace std;
5 map<string, char> decoder =  { {"0 1 0 1", 'A'}, {"0 1 1 1", 'B'},
6       {"0 0 1 0", 'C'}, {"1 1 0 1", 'D'}, {"1 0 0 0", 'E'}, {"1 1 0 0", 'F'} };
  …
7 char buf[10];                                      放資料進去時不需要管順序
8 fgets(buf, sizeof(buf), stdin); buf[7] = 0;
9 printf("%c", decoder[buf]);
```

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼
  也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

```
1 #include <map>
2 #include <string>
3 #include <cstdio>
4 using namespace std;
5 map<string, char> decoder =  { {"0 1 0 1", 'A'}, {"0 1 1 1", 'B'},
6       {"0 0 1 0", 'C'}, {"1 1 0 1", 'D'}, {"1 0 0 0", 'E'}, {"1 1 0 0", 'F'} };
  …
7 char buf[10];
8 fgets(buf, sizeof(buf), stdin); buf[7] = 0;
9 printf("%c", decoder[buf]);
```

decoder["0 1 0 1"] = 'A';

放資料進去時不需要管順序
它自己會紀錄字串索引的順序

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼
  也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

```
1 #include <map>
2 #include <string>
3 #include <cstdio>
4 using namespace std;
5 map<string, char> decoder =  { {"0 1 0 1", 'A'}, {"0 1 1 1", 'B'},
6       {"0 0 1 0", 'C'}, {"1 1 0 1", 'D'}, {"1 0 0 0", 'E'}, {"1 1 0 0", 'F'} };
  …
7 char buf[10];
8 fgets(buf, sizeof(buf), stdin); buf[7] = 0;
9 printf("%c", decoder[buf]);
```

decoder["0 1 0 1"] = 'A';

放資料進去時不需要管順序
它自己會紀錄字串索引的順序
可以很簡單、有效率地讀取資料

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼
  也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

```
1 #include <map>
2 #include <string>
3 #include <cstdio>
4 using namespace std;
5 map<string, char> decoder =  { {"0 1 0 1", 'A'}, {"0 1 1 1", 'B'},
6       {"0 0 1 0", 'C'}, {"1 1 0 1", 'D'}, {"1 0 0 0", 'E'}, {"1 1 0 0", 'F'} };
  …
7 char buf[10];
8 fgets(buf, sizeof(buf), stdin); buf[7] = 0;
9 printf("%c", decoder[buf]);
```

此範例由字串索引字元，當然
也可以設計從整數索引字元

放資料進去時不需要管順序
它自己會紀錄字串索引的順序
可以很簡單、有效率地讀取資料

稀疏的對照表 (續)

- 使用 C++ STL 的 **map**

  雖然語法比較複雜一些，但是概念簡單，應用的程式碼
  也會很精簡，參加 APCS 檢測的話是值得練習的工具

  內部是一個平衡二元搜尋樹 (balanced binary search tree)

  此處概念上就像是一個以字串作為索引的字元資料陣列

```
1 #include <map>
2 #include <string>
3 #include <cstdio>
4 using namespace std;
5 map<string, char> decoder = { {"0 1 0 1", 'A'}, {"0 1 1 1", 'B'},
6       {"0 0 1 0", 'C'}, {"1 1 0 1", 'D'}, {"1 0 0 0", 'E'}, {"1 1 0 0", 'F'} };
  …
7 char buf[10];
8 fgets(buf, sizeof(buf), stdin); buf[7] = 0;
9 printf("%c", decoder[buf]);
```

此範例由字串索引字元，當然
也可以設計從整數索引字元

放資料進去時不需要管順序
它自己會紀錄字串索引的順序
可以很簡單、有效率地讀取資料

C++ 的 map 和 python 的字典 dict 概念是一樣的