

程式設計研討專題二

110403518 林晉宇 11/16

一、紅黑樹

C++ STL 的 map 跟 set 容器背後就是用紅黑樹實現的，故這裡直接使用 set 來實作。

時間複雜度: Search $O(\log n)$ / Insert $O(\log n)$ / Delete $O(\log n)$

Pseudo code:

```
input n
for i in [0,n-1]:
    input x
    set.insert(x)

input m
while(m--):
    input operation, num
    if operation=='s':
        if (num exist):    print "Found"
        else:              print "Not Found"
    else if operation=='d':
        if (num exist):    print "Delete Success", set.erase(num)
        else:              print "Delete Failed"
    else if operation=='l':
        if (num exist):    print "Insert Failed"
        else:              print "Insert Success", set.insert(num)
```

二、 Making Binary Search Dynamic

時間複雜度分析:

1. **Search:** 遍歷每一個陣列，假設第 i 個陣列是滿的，其長度為 2^i ，因其為排序好的陣列，故可以二分搜，時間為 $O(i)$ ，而 i 的範圍為 $(0, \log n)$ ，故搜尋的時間複雜度為 $O(\log^2(n))$ 。
2. **Insert:** 當有新元素加入時，直接把該元素加入進 A_0 陣列，如果已經存在一 A_0 陣列，則把兩個 merge 後形成 A_1 ，依此類推，直到不用再 merge 為止，假設我們需要把 A_0, A_1, \dots, A_{m-1} merge，這樣的時間複雜度為 $O(2^m)$ ，而最糟情況 $m=k$ ，則時間複雜度為 $O(n)$ 。但我們可以發現當操作一系列的 insert 後， n_0 每次都會被改變， n_1 每兩次被改變...etc，每次改變代表被 merge 一次，所以 m 次的 insert \rightarrow 時間複雜度可視為 $mO(\log n) \rightarrow O(\log n)$ 。
3. **Delete:** 先找到不為空的最小陣列($n_i \neq 0$)，從此陣列開始找，我們可能需要 k (深度，即 $\log(n)$)個陣列，所以時間複雜度為 $O(\log n)$ 。

Credit: [17-2 Making binary search dynamic - CLRS Solutions \(walkccc.me\)](https://walkccc.me/CLRS/Solutions/17-2-Making-binary-search-dynamic/)

Pseudo code:

pair search(num):

遍歷每一層，二分搜每一層

If 找到 num: return {num 的位置}

else: return {-1,-1}

void merge(vector<int> a, vector<bool> b, h 欲合併的深度):

if h >= 現在的深度: 新增一層空陣列

if h 層為空: 直接將 a 塞進 h 層

bool insert(num):

if 該數字存在: return false

else if 該數字被標記為刪除(false): 更改標記成 true

else:

vector<int> a; a.push_back(num);

vector<bool> b; b.push_back(true);

merge(a,b,0) //0 代表第 0 層

bool delete(num):

pos=search(num)

if 該點不存在: return false

else 該點存在: exist[pos]=false //把該點標記不存在

if pos 所在層數有一半的點已經被刪除:

if 上一層為空: 則把當前這一半全部塞進上一層

else: merge(a,b,上一層)

main:

input N

for i~N: input x, insert(x)

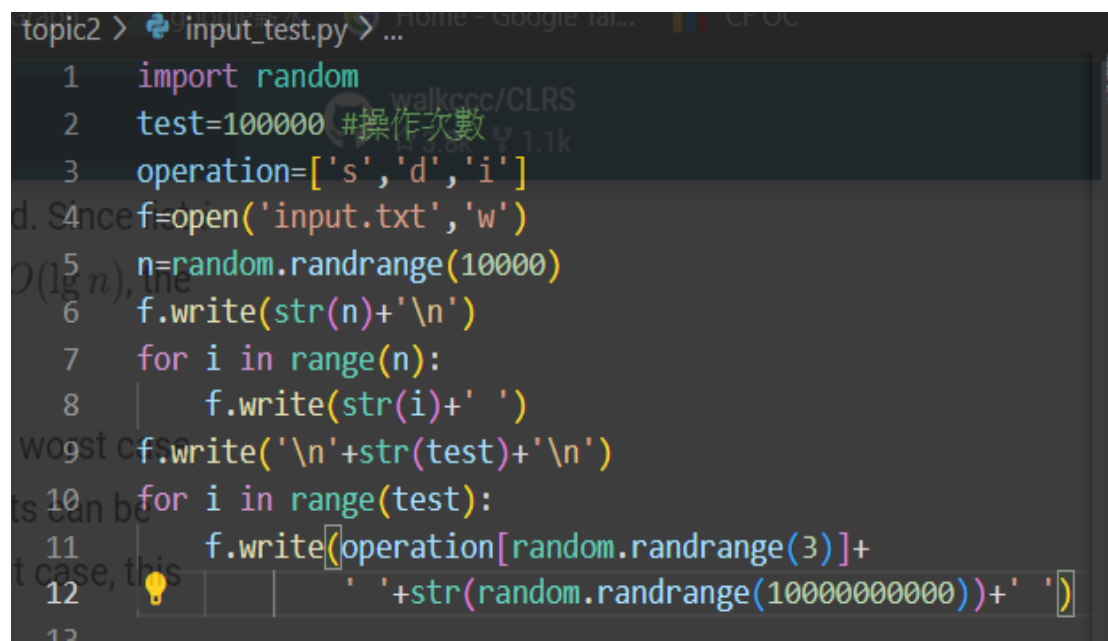
```

input t
while(t--):
    input operation, num
    if operation=='s':
        if search(num): print "Found"
        else: print "Not Found"
    else if operation=='i':
        if insert(num): print "Insert Success"
        else: print "Insert Failed"
    else if operation=='d':
        if delete(num): print "Delete Success"
        else: print "Delete Failed"
return 0

```

三、 時間比較

產生側資: 使用 python 產生側資，初始有 10000 個數字，再來 100000 個隨機操作。



```

topic2 > input_test.py > ...
1 import random
2 test=100000 #操作次數
3 operation=['s','d','i']
4 f=open('input.txt','w')
5 n=random.randrange(10000)
6 f.write(str(n)+'\n')
7 for i in range(n):
8     f.write(str(i)+' ')
9 f.write('\n'+str(test)+'\n')
10 for i in range(test):
11     f.write(operation[random.randrange(3)]+
12             ' '+str(random.randrange(10000000000))+' ')
13

```

1. 紅黑樹(set): 0.212 秒(100000 筆操作)

```
100000 Insert Success
100001 Red Black Tree: 0.212 S
```

2. Dynamic Binary Search: 0.59 秒(100000 筆操作)

```
100000 Insert Success
100001 Dynamic Binary Search: 0.59 S
```

1. 紅黑樹(set): 2.688 秒(1000000 筆操作)

```
1000000 Not Found
1000001 Red Black Tree: 2.688 S
```

2. Dynamic Binary Search: 6.093 秒(1000000 筆操作)

```
1000000 Not Found
1000001 Dynamic Binary Search: 6.093 S
```

1. 紅黑樹(set): 19.869 秒(10000000 筆操作)

```
10000000 Not Found
10000001 Red Black Tree: 19.869 S
```

2. Dynamic Binary Search: 41.766 秒(10000000 筆操作)

```
10000000 Not Found
10000001 Dynamic Binary Search: 41.766 S
```

四、 結論

從時間複雜度可以看出，dynamic binary search 的 search 需要花比較多的時間，所以 search 用的越多次，時間與紅黑樹的差距越顯著，透過 python 產生的側資實際比較，可以發現紅黑樹與 dynamic binary search 的時間大概差接近三倍，所以可知 c++ 的 map 及 set 容器算十分有效率的，以後可以直接用。