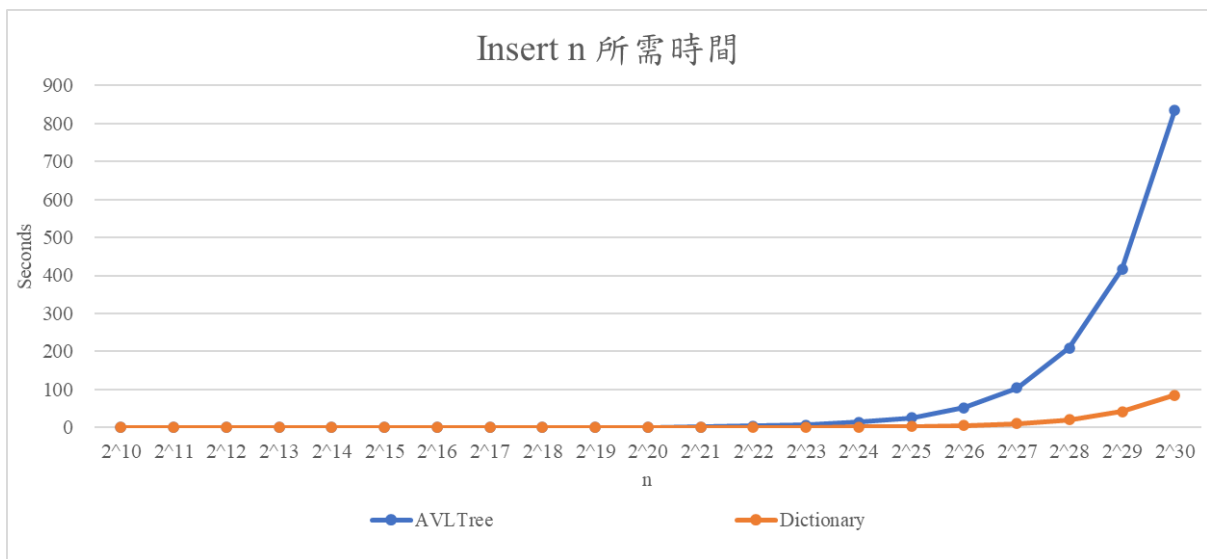


一、 實作目標：比較 hash table 與 balanced binary search tree。

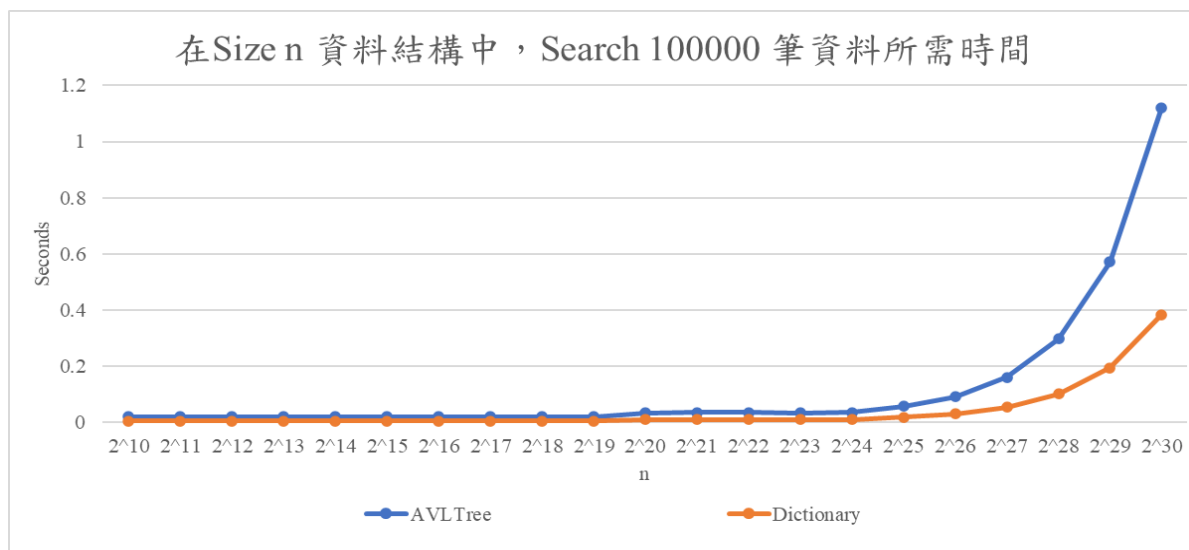
二、 比較方法：

○ 在報告中請畫出每種資料結構**新增**資料所需時間：

- 針對每種資料結構，先產生一個空的資料結構。
- 新增 $n$ 筆資料至該資料結構，並計算總共花費時間。每筆資料都是隨機從 $1 \sim 2^{30}$ 選擇，每個數字被選到的機率都一樣。
- $n = 2^k$  ( $k = 10, 11, 12, \dots, 30$ )。
- 把實驗結果畫成折線圖，圖中有**兩**條折線（分別對應 hash table 與 balanced binary search tree）， $x$ 軸是 $n$ 值， $y$ 軸是新增 $n$ 筆資料到一個空的資料結構所需時間。
- 若 $n$ 太大導致 $y$ 值超過 10 分鐘，請估計 $y$ 值並把估計時間畫在折線圖上。報告中請說明估計方法。



- 實驗結果：Hash Table 在 Python 中內建是 Dictionary，balanced binary search tree 使用 Python package “bintrees”實作 AVLTree。插入表現 Dictionary 都比 AVLTree 快。驗證 Hash Table 時間複雜度是  $O(1) < \text{balanced binary search tree } O(\log n)$ 。
- 在報告中請畫出每個資料結構**搜尋**資料所需時間：
  - 針對每種資料結構，先產生一個空的資料結構。
  - 新增 $n$ 筆資料至該資料結構。每筆資料都是隨機從 $1 \sim 2^{30}$ 選擇，每個數字被選到的機率都一樣。
  - 在該資料結構中搜尋十萬筆資料，並計算所需時間。每筆資料都是隨機從 $1 \sim 2^{30}$ 選擇，每個數字被選到的機率都一樣。
  - $n = 2^k$  ( $k = 10, 11, 12, \dots, 30$ )。
  - 把實驗結果畫成折線圖，圖中有兩條折線（分別對應 hash table 與 balanced binary search tree）， $x$ 軸是 $n$ 值， $y$ 軸是搜尋十萬筆資料所需時間。
  - 若 $n$ 太大導致 $y$ 值超過 10 分鐘，請估計 $y$ 值並把估計時間畫在折線圖上。報告中請說明估計方法。



- 實驗結果：搜尋表現 Dictionary 都比 AVLTree 快。驗證 Hash Table 時間複雜度是  $O(1)$  < balanced binary search tree  $O(\log n)$ 。

### 三、標準或常見函式庫

- hash table 是來自標準（或常見）函式庫，請附上資料來源網址並截圖說明實作使用 hash table（或是時間複雜度為  $O(1)$ ）。
- 實驗程式碼（含新增與搜尋的程式碼範例）與使用說明。

dict

The Average Case times listed for dict objects assume that the hash function for the objects is sufficiently robust to make collisions uncommon. The Average Case assumes the keys used in parameters are selected uniformly at random from the set of all keys.

Note that there is a fast-path for dicts that (in practice) only deal with str keys; this doesn't affect the algorithmic complexity, but it can significantly affect the constant factors: how quickly a typical program finishes.

Operation	Average Case	Amortized Worst Case
kind	$O(1)$	$O(n)$
Copy[3]	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(n)$
Set Item[1]	$O(1)$	$O(n)$
Delete Item	$O(1)$	$O(n)$
Iteration[3]	$O(n)$	$O(n)$

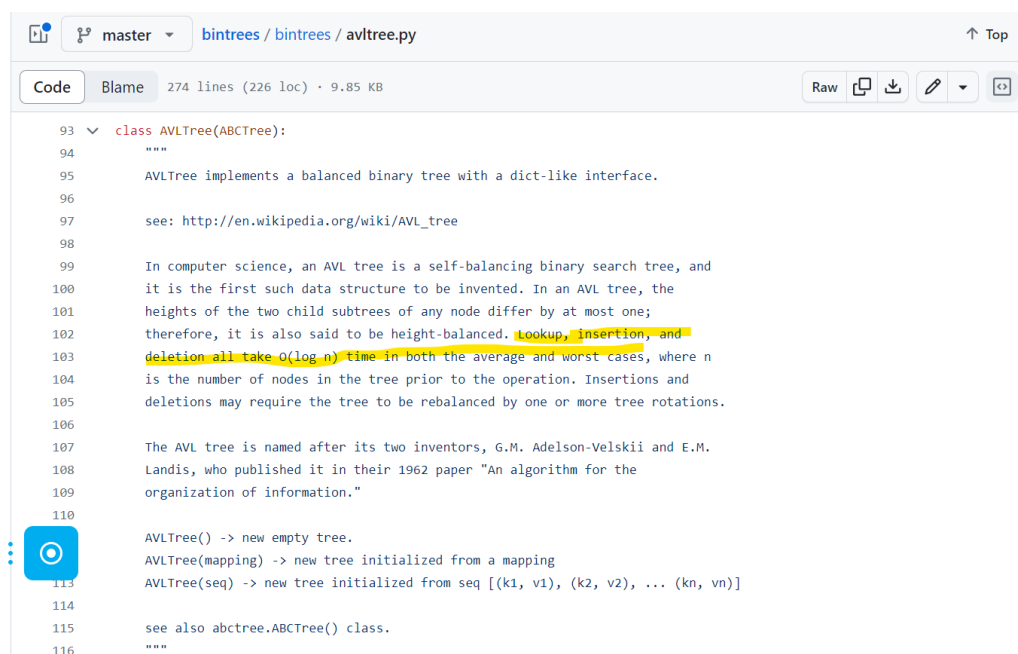
<https://wiki.python.org/moin/TimeComplexity>

```

1 # insert to the dictionary
2 a = dict(one=1, two=2, three=3)
3 b = {'one': 1, 'two': 2, 'three': 3}
4 c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
5 d = dict([('two', 2), ('one', 1), ('three', 3)])
6 e = dict({'three': 3, 'one': 1, 'two': 2})
7 f = dict({'one': 1, 'three': 3}, two=2)
8 print(a)
9 print(b)
10 print(c)
11 print(d)
12 print(e)
13 print(f)
14
15 g={}
16 g[3]='value3'
17 g[1]='value1'
18 g[2]='value2'
19 print(g)
20
21 # search a key and return its value
22 print(a['two'])
23 print(a.get('two'))

```

- balanced binary search tree 是來自標準（或常見）函式庫，請附上資料來源網址並截圖說明實作使用 balanced binary search tree（或是時間複雜度為 $O(\log n)$ ）。
- 實驗程式碼（含新增與搜尋的程式碼範例）與使用說明。



The screenshot shows a GitHub file view for `avltree.py` in the `bintree` repository. The file is 274 lines long, with 226 lines of code and 9.85 KB in size. The code defines an `AVLTree` class that implements a balanced binary search tree. The class docstring includes a description of AVL trees, a reference to the Wikipedia page, and a list of methods: `AVLTree()` for a new empty tree, `AVLTree(mapping)` for a tree initialized from a mapping, and `AVLTree(seq)` for a tree initialized from a sequence. The code also mentions that lookups, insertions, and deletions all take  $O(\log n)$  time.

```
93 class AVLTree(ABCTree):
94     """
95     AVLTree implements a balanced binary tree with a dict-like interface.
96
97     see: http://en.wikipedia.org/wiki/AVL_tree
98
99     In computer science, an AVL tree is a self-balancing binary search tree, and
100     it is the first such data structure to be invented. In an AVL tree, the
101     heights of the two child subtrees of any node differ by at most one;
102     therefore, it is also said to be height-balanced. Lookup, insertion, and
103     deletion all take O(log n) time in both the average and worst cases, where n
104     is the number of nodes in the tree prior to the operation. Insertions and
105     deletions may require the tree to be rebalanced by one or more tree rotations.
106
107     The AVL tree is named after its two inventors, G.M. Adelson-Velskii and E.M.
108     Landis, who published it in their 1962 paper "An algorithm for the
109     organization of information."
110
111     AVLTree() -> new empty tree.
112     AVLTree(mapping) -> new tree initialized from a mapping
113     AVLTree(seq) -> new tree initialized from seq [(k1, v1), (k2, v2), ... (kn, vn)]
114
115     see also abctree.ABCTree() class.
116     """
```

<https://github.com/mozman/bintree/blob/master/bintree/avltree.py>



The screenshot shows a Python script that imports the `AVLTree` class from the `bintree` module. It creates an `AVLTree` object and demonstrates inserting three key-value pairs: ('key3', 'value3'), ('key1', 'value1'), and ('key2', 'value2'). It then prints the tree object, searches for the key 'key1' using `__getitem__`, `get`, and direct indexing.

```
1 from bintree import AVLTree
2
3 tree = AVLTree()
4
5 # insert(key, value)
6 tree.insert('key3', 'value3')
7 tree.insert('key1', 'value1')
8 tree.insert('key2', 'value2')
9 print(tree)
10
11 # search for a key
12 print(tree.__getitem__('key1'))
13 print(tree.get('key1'))
14 print(tree['key1'])
```

#### 四、心得、疑問、與遇到的困難

在找 Python 文件時，balanced binary search tree 比較不好找，pypi 上有 package 但推薦大家使用另一種容器 `sortedcontainers`(<https://pypi.org/project/bintree/>)，一開始在比較兩者不同但覺得樹狀結構比較符合需求所以仍用 `AVLTree` 實作。