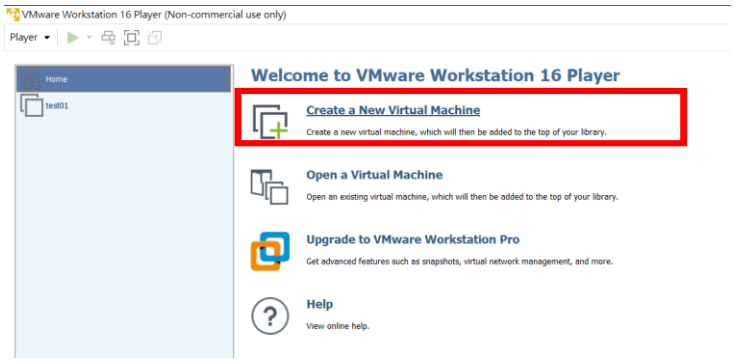
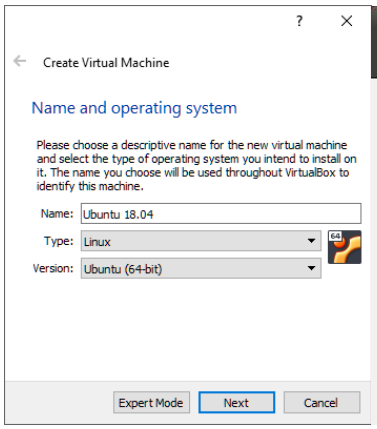
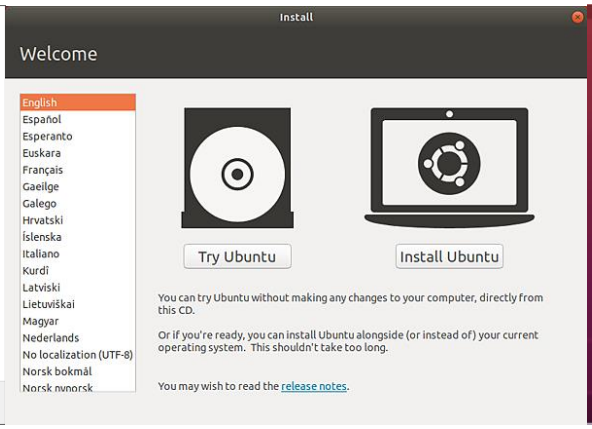
E14071025 趙泓瑞

Part A

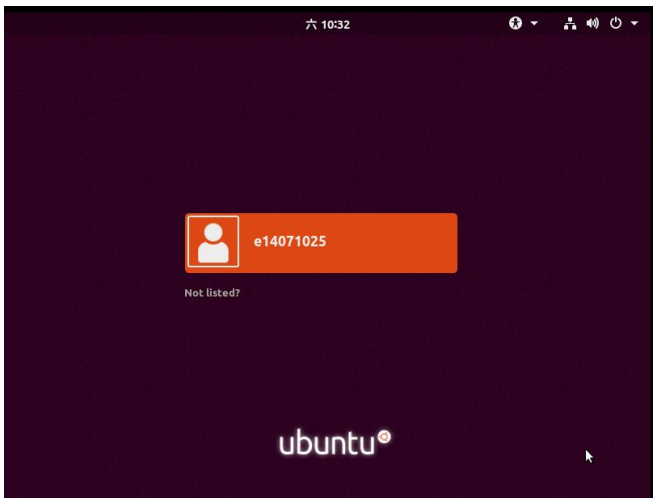
我使用 VM(VMware Workstation 16 Player)下載後，經過最初步的安裝後進入主頁面。點選 Create a New Virtual Machine。







接著需要一系列的安裝過程，從上的選擇作業系統，到語言選擇、安裝大小(我選 Normal installation)、所在地到 user 創立。最後進入安裝 ubuntu18.04 環節。



將學號設為使用者名稱後的登入畫面。

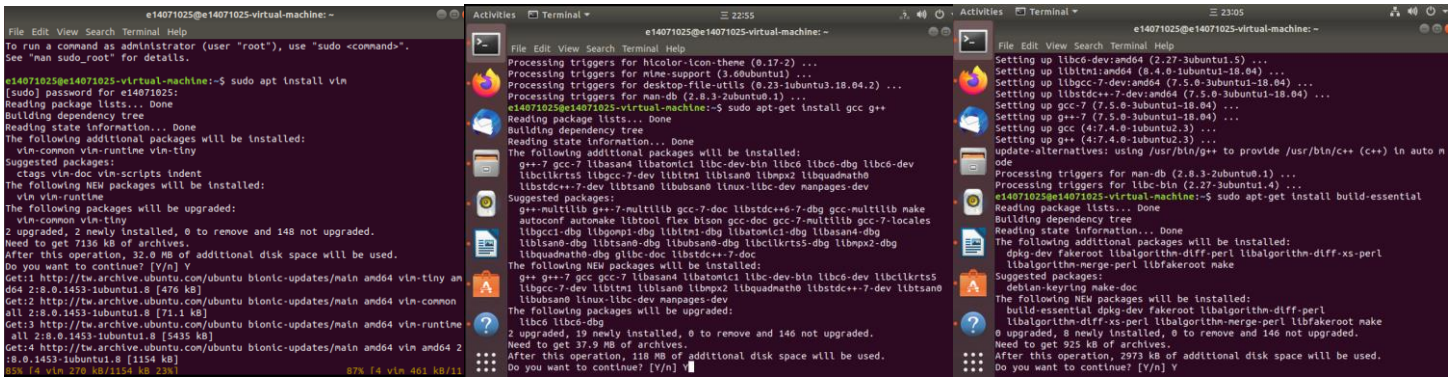
編譯執行過程:

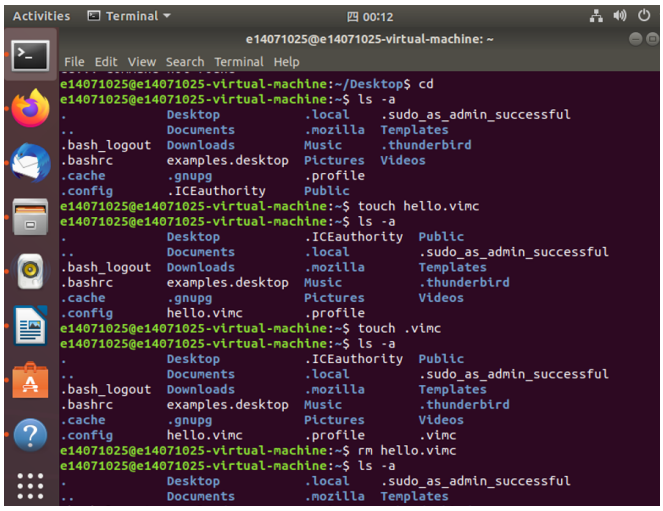
需安裝 Vim 指令以編譯 hello 程式。則需輸入以下三個指令安裝。

sudo apt install vim

sudo apt-get install gcc g++

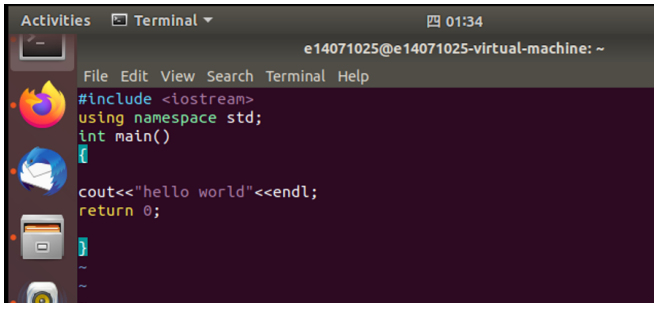
sudo apt-get install build-essential



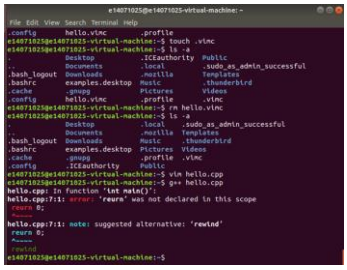


在~中 ls -a 找到 .vimrc 的檔案。如果像我一樣沒有需要自己創造一個。

最後輸入 vim hello.cpp 即可開始編輯。



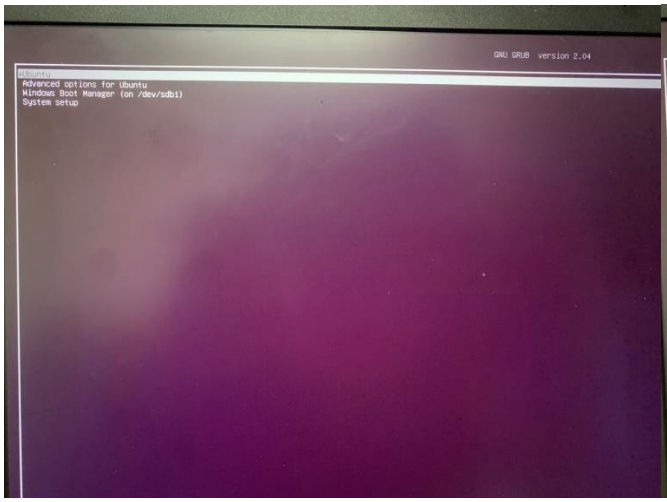
編輯完後輸入指令 g++ hello.cpp 編譯。第一次我 return 打錯所以編譯失敗，產生 error。

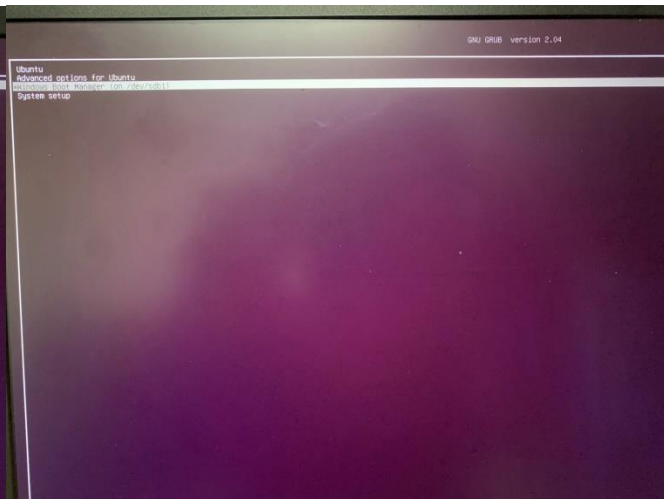


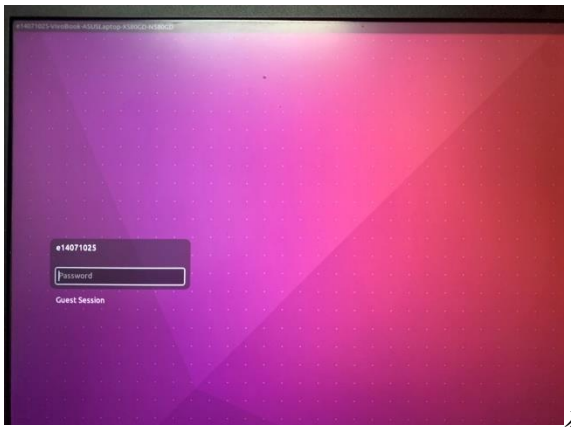
修正後，輸入編譯指令即會產生一個 a.out 檔。

輸入指令./ a.out，即可輸出 hello world。

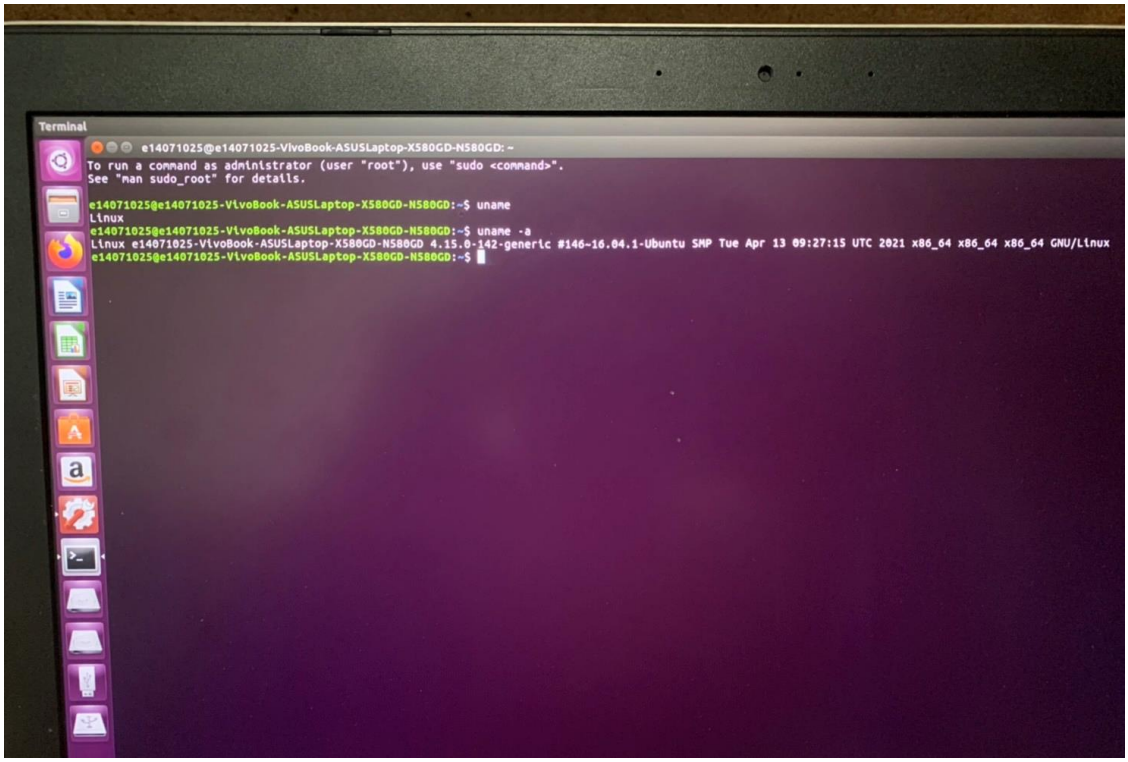
加分:雙系統選單畫面如下







左側為登入畫面



雙系統安裝的版本為 ubuntu16.04 版，用指令顯示安裝版本。

安裝遇到問題

1. 一開始下載完 ubuntu 時不知道先需要裝 VM 才能打開，因此瞎忙了一陣子。自以為電腦無法開啟 iso 檔。
2. 對於要如何在 Linux 系統編譯程式，不知道要使用裝 vim 或是 vi 去完成。
3. 加分 Rufus 做安裝光碟時，原本有割空間出來，結果意外把硬碟整個吃掉，哭倒。
4. 我把雙系統做在硬碟時，不知道怎麼找到電腦的 bios，花了好久，最後知道要關機並且按住 F2 重開。
5. 在雙系統時，install ubuntu 時卡住，才知道跟 nvidia 顯卡相衝。需要按 e 找到 linux 那行最後加上 nomodeset 即可。

Part B

red-black tree 是一種 extended binary tree，每一個節點不是紅就是黑。紅黑樹中，第一，根(root)和所有的 external node 都必須是黑色。第二，從 root 到 external node 的路徑中都沒有連續的紅點，紅點中間必須插入黑點。第三，路徑過程中經過的黑點個數都相同。相同的定義也可以套用在程式 C++裡的 pointer。

紅黑樹中還有其他特質，例如 P 及 Q 在同一個 binary search tree 中，P 及 Q 的路徑長度中 length(P)<length(Q)*2。

紅黑樹的高度 H，N 為 internal nodes 的數量，以及 rank R。則會有以下規律:

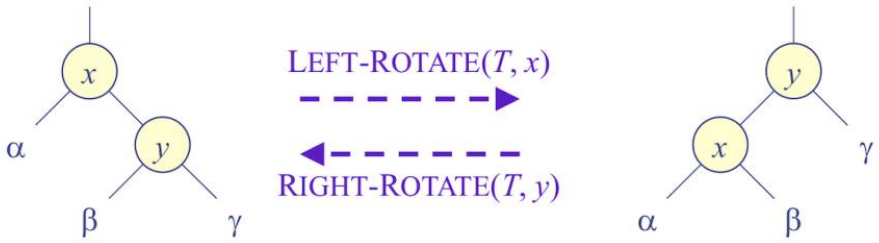
(a)$H \leqq 2*R$

(b)$N \geqq 2^R - 1$

(c) $H \leqq 2*log_2(N+1)$

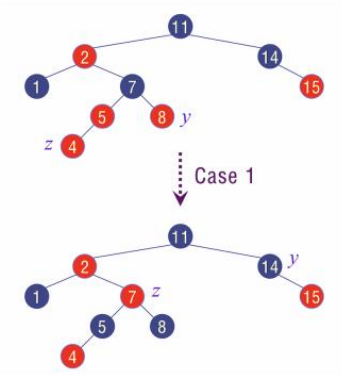
(1)

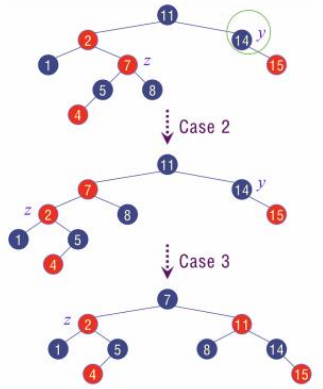
- left rotation(lr):把自己的 right children 放到自己現在的位置，則自己變成原 right children 的 left children。 right children 的 left children 則變成自己的 right children。
- right rotation(rr): 相似的流程，是把自己的 left children 放到自己所在位置，則自己成為原 left children 的 right children。left children 的 right children 則變成自己的 left children。



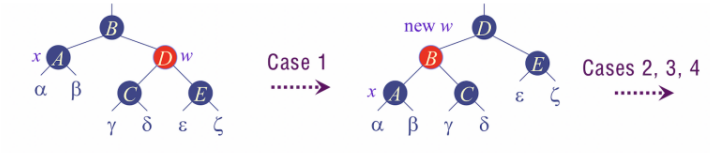
- 因紅黑樹是 binary search tree 分枝，所以搜尋法相同。都是與現在的 node 比較大小，若要尋找的比現在所在的 node 大時，往 node 的 right children 路徑移動。相反，若要尋找的比現在所在的 node 小，就往 node 的 left children 路徑移動，則若與 node 值相同就是找到。若走到 external leaf node(NULL)時，代表要目標值不在這個樹裡。

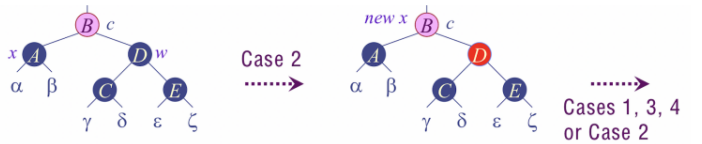
- 插入方式是比較 node 的值，若要插入的數比現在所在的 node 大時，往此 node 的 right children 走，若要插入的數比現在所在的 node 小時便往此 node 的 left children 走，直到 children 為 null，就把要插入的值在此空的位置且為紅色，最後要檢查是否符合紅黑樹的基本三個規則。
  若不符合時，分為三種狀況。第一個是 insert node 與他的 parent 都是紅色且 insert node 的 uncle 是紅色，這就要把 insert node 跟他的 uncle 變色，這會讓連續兩個紅色 node 移到 grandparent node 與其 parent(node A)，且此時 insert node 的 uncle 會是黑色且 uncle 會與此 node 與他們 parent node 在同側。這同時也就是第二種狀況，此時就把 node A 做 left rotation(right rotation)，會變成 insert node 的 uncle 會是黑色且 uncle 會與此 node 在他們 parent node 的不同側這同時也就是第三種狀況，接著再對 node A 做 left rotation(right rotation)再將 node 顏色改變就能得到結果。

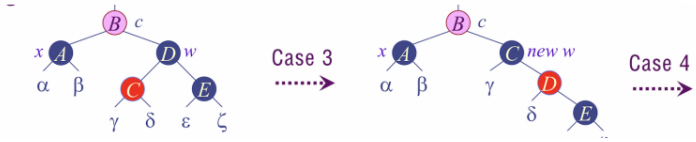




- 刪除會有四種狀況，第一種是 delete node 的 sibling 是紅色，這時就對 parent node 做 left rotation(right rotation)，會使 delete node 的 sibling 是黑色，若 sibling 的兩個 children 都是黑色同時也是第二種狀況，則直接把 sibling 與他的兩個 children 變色，如此便有可能結束，或進入其他種狀況，若 sibling 的 left children 是紅色 right children 是黑色同時也是第三種狀況，此時對 sibling 做 left rotation(right rotation)，這會變成 delete node 的 sibling 是黑色且 right children 是紅色同時也是第四種狀況，此狀況就直接對 parent 做 left rotation(right rotation)，便能得到答案，這只會有第一二種狀況會重複若是能到第三或四種狀況便已完成。







(2)與其他資料結構比較優缺點

- 比較 AVLtree 時，因為 AVL tree 的要求是嚴格的平衡，都是最多兩次樹旋轉來實現 rebalance，但是 AVL tree rotation 是 O(log(n))，rbtree 的效率是 O(1)，所以紅黑樹復衡效率較高。由於 AVL 高度平衡，所以 AVL 的 search 效率較高，所以實際上若做 search 時，次數遠遠多於 insert 和 delete，所以該選擇 AVL，若 search、insert、delete 次數幾乎一樣時，應該選擇紅黑樹。

- 比較 interval tree 時，interval tree 會把原本 node 變成一段區間，這方法在表現 event 上很耐用，能清楚知道哪些 event 會重疊到，但由於需要使用一個 key 來記錄此 node 下的所有 node 的 maxm，因此空間需要較多，使用 key 來存位置，所以在搜尋上則是紅黑樹較有效率，因此 search 的 order 維持在 log(n)。

- 比較 order statistic trees 時，order statistis trees 可以藉由在每個 node 多存一個 key 讓 search 的 order 維持在 log(n)，能夠讓尋找效率提高，但存取 key 值需要做一倍的空間，且需要對 rank 運算，在 insert 與 delete 上需花費較多時間，因此在實際運用上若 search 的次數遠遠多於 insert 和 delete，則應該選擇 AVL，但若 insert、search 和 delete 次數差不多，就該選擇紅黑樹。

解釋 rbtree.c 程式碼內容:

- Left rotation :使用 y 存取 x 的 right node(最後會成為 y 的 parent)，在確認 x 的 right children 不是空的時才能把 right children 到 x 的 link 連起來，若 x 的 parent 是 null 代表 x 是 root，因此在 rotate 完後 root 會變成 y，若不是就把 x 的 parent 與 y 連起來，最後再把 x 變為 y 的 left children，兩者 link 連接起來。

- Insert:先檢查樹裡有沒有 node，假如沒有就把他變成 root node，若有則從 root node 開始比若要插入的數比現在所在的 node 小時便往此 node 的 left children 走，直到 children 為空的，便把要插入的值在這個空，再把迴圈內最後比較的 node 設為他的 parent 並且 link 起來，且將顏色設定為紅色，以防踩到 black hight 的規則，最後在用 insertFixUp 解決連續兩紅點的問題。

- insertFixUp :使用 y 來存取 x 的 uncle，先檢查第一個情況是不是 uncle 是紅色，此時要把 insert node 跟他的 parent 與他 parent 的 parent 與他的 uncle 變色，會讓連續兩個紅色 node 移到 grandparent node 與 grandparent node 的 parent 也就是把 z 變成原本的 grandparent 位置，這會到第二種狀況, 也就是 else 內的 z 的 uncle 會是黑色且 uncle 會與此 node 在他們 parent node 的同一側，此時就把 z 的 parent 與 grandparent 變色(為了不讓連續兩紅發生)再對 z 的 parent 與 grandparent 做 rotation(維持 black hight balance)，就可得到答案。若到第三種狀況就是 else 內 z 的 uncle 會是黑色且 uncle 會與此 node 在他們 parent node 的不同側，只需要把 z 的 parent 與 grandparent 變色(為了不讓連續兩紅發生)再 rotating z 的 grandparent 以保持 black hight balance。

- Inorder: 先往左走到最底(碰到 NULL 時)，輸出自己的位置，再往右走。因為 left children>node> right children，如此就可以輸出由小到大的值。

3.討論 rb tree 在 Linux 系統的應用，並解釋 rb tree 的特性所造成的影響

- rbtree 在 Linux 系統的應用

在 Linux 中有用到 rbtree 的有 deadline、anticipatory、CFQ I/O 排程和 CD/DVD 驅動，都是以 rb tree 來進行請求跟蹤。舉例來說，EXT3 檔案系統是使用 rb tree 來管理目錄，高精度計時器（high-resolutiontimer）用 rb tree 組織定時請求，還有虛擬儲存管理系統是使用 rb tree 進行 Virtual Memory Areas 的管理等等。此外檔案描述符密碼鑰匙，等級級令牌桶排程的網路資料包也是用 rb tree 資料進行組織和管理。

- 解釋 rb tree 的特性所造成的影響

由於在 linux 中沒有 key 域，因此結構體獲取資料資訊是通過 CONTAINER_OF 這個巨集達成，在 rb_node 結構體[__attribute__((aligned(sizeof(long))))]包 sizeof(long)，藉由位元組對齊方法發現在結構 address 後還有兩個沒有用到，所以可藉此表示顏色，若要取出顏色則看最後一位即可。若要取出地址將兩位清零就可，抑或是使用 rbtree 的 delete 與 insert 處理問題，並用 rotation implement。此外 Linux 為了有好的 cache locality，對 rbtree 針對達到效能最好，使用較少的間接層。由於 rbtree 的 balance 以及在 rotate 兩次內便能完成 insert 或 delete，且所消耗的 resource 不多，使其能在 Linux 上很好的功效。

參考資料:

https://kknews.cc/zh-tw/code/8gy295e.html

Fundamentals of Data Structures in C++, 2/e

演算法相關內容

http://reborn2266.blogspot.com/2012/01/red-black-trees-rbtree-in-linux.html

https://www.itread01.com/content/1550480764.html