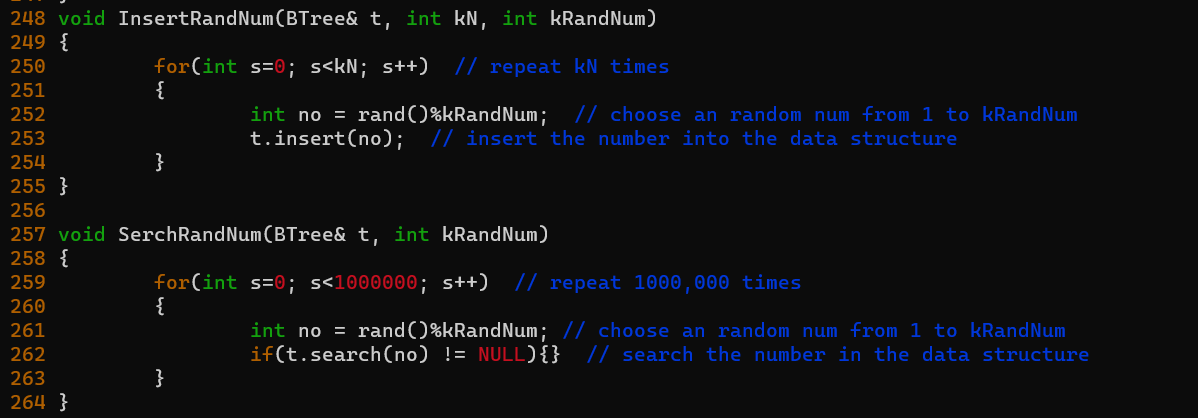
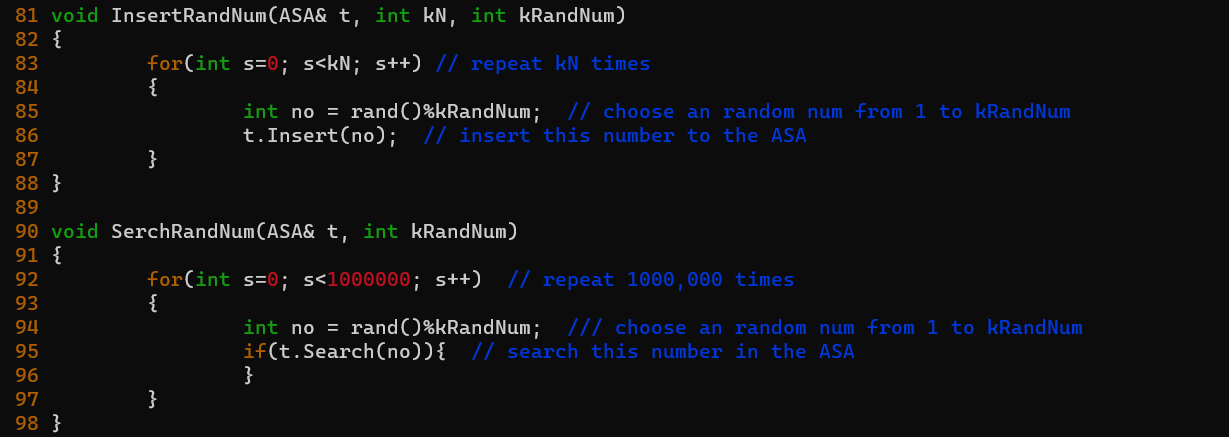
**作業三**

**資科三 109208064 俞懿**

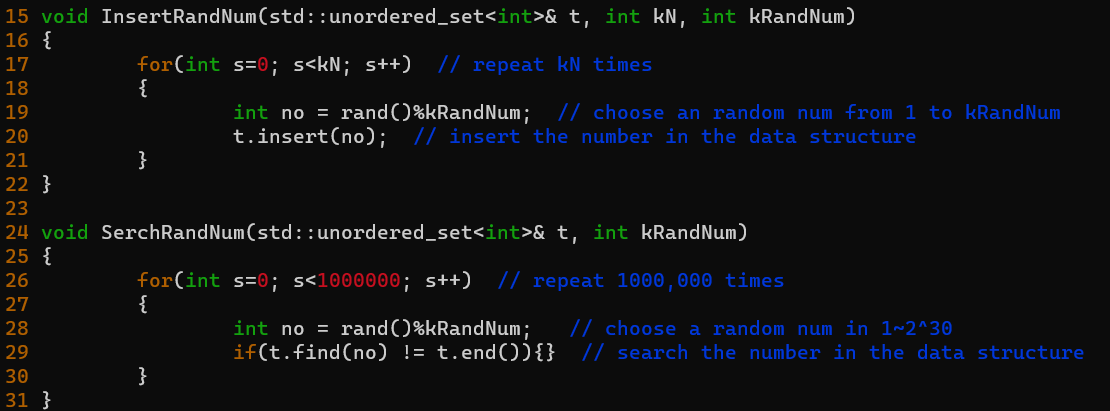
* 作業目的：比較B-tree (或B+tree)、treap、skip list、array of sorted arrays、hash table。
* （30%）每種資料結構**新增**資料所需時間：
  + 實驗結果折線圖。x軸是新增的資料數量，y軸是新增該數量筆資料到一個空的資料結構所需時間。
  + 其中ASA只能測到2^28次方，而Skip List、Treap、Hash Table只能測到2^27，B-Tree只能測到2^26次方。
  + 為了估計剩下不足的資料，我計算**最後一筆資料**對**倒數第二筆資料**花費時間的**倍數** (Tn / Tn-1)，以此倍數作為下一回合花費時間的成長率，例如**Tn+1估計數值 = Tn** \* (**倍數**)、**Tn+2 = Tn+1 \* 倍數**。估計結果如下 :
  + 花費的時間越多代表速度越慢。由上圖可知，此五種資料結構的新增資料的速度排名如下 :
    1. B-Tree
    2. Hash Table
    3. Skip List
    4. Array Of Sorted Array (ASA)
    5. Treap
* （30%）每個資料結構**搜尋**資料所需時間：
  + 實驗結果折線圖。x軸是新增的資料數量，y軸是新增該數量筆資料到一個空的資料結構所需時間。
  + 其中ASA只能測到2^28次方，而Skip List、Treap、Hash Table只能測到2^27，B-Tree只能測到2^26次方。
  + 為了估計剩下不足的資料，我計算**最後一筆資料**對**倒數第二筆資料**花費時間的**倍數** (Tn / Tn-1)，以此倍數作為下一回合花費時間的成長率，例如**Tn+1估計數值 = Tn** \* (**倍數**)、**Tn+2 = Tn+1 \* 倍數**。
  + 估計實驗結果如下圖。其中Skip List所需花費的時間最多，遠高出其他資料結構，因此另外放了圖二，去除Skip List可以更清楚地比較另外四種資料結構搜尋所需花費的時間。
  + 由上圖可知，此五種資料結構的搜尋的速度排名如下 :
    1. Treap
    2. Hash Table
    3. Array Of Sorted Array (ASA) / B-Tree *(當資料數<2^22時ASA贏)*
    4. B-Tree / Array Of Sorted Array (ASA) *(當資料數<2^22時B-Tree贏)*
    5. Skip List
* 資料結構程式碼來源（未提供則整份作業以0分計算）
  + 1. B-Tree : <https://www.geeksforgeeks.org/insert-operation-in-b-tree/>
    2. Skip List : <https://www.geeksforgeeks.org/skip-list-set-3-searching-deletion/>
    3. Treap : <https://www.geeksforgeeks.org/implementation-of-search-insert-and-delete-in-treap/>
* 解釋如何實作array of sorted arrays（以註解呈現）
  + <https://github.com/yvonne90190/Data-Structure-HW/blob/main/hw3/arrayOfSortedArray/ASA.cc>
* 實驗程式碼（含新增與搜尋的程式碼範例）與使用說明。
  + 完整檔案: <https://github.com/yvonne90190/Data-Structure-HW/tree/main/hw3>
    1. B-Tree :
       - BTree t ; // declare a BTree name t
       - t.insert(no); // insert the number “no” in BTree t
       - t.search(no); // search the number “no” in the BTree t, return a bool



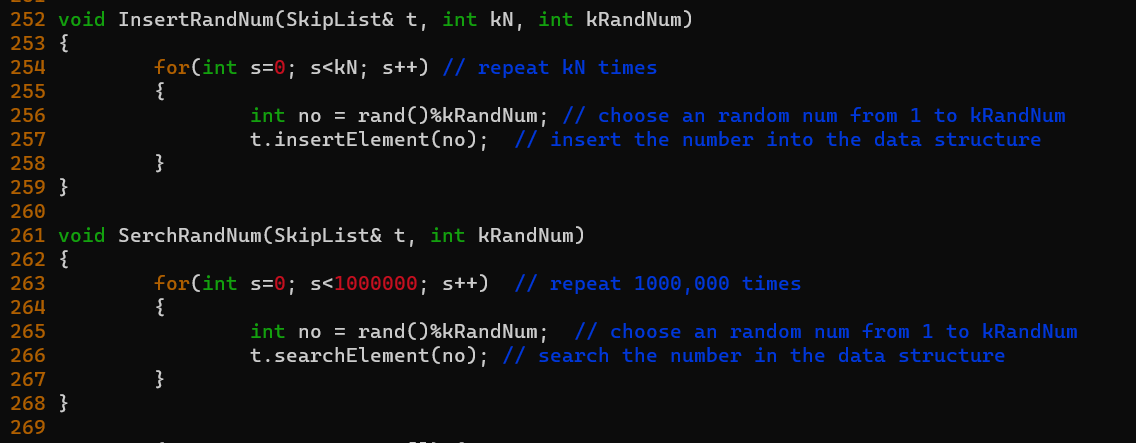
* + 1. Array of Sorted Array (ASA):
       - ASA t ; // declare a ASA name t
       - t.insert(no); // insert the number “no” in ASA t
       - t.search(no); // search the number “no” in the ASA t, return a bool



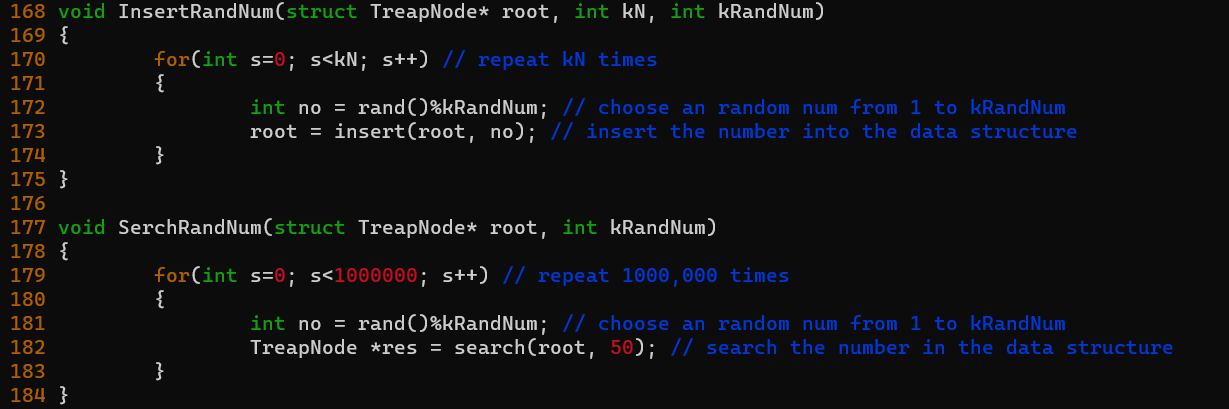
* + 1. Hash Table (std::unordered\_set<int>) :
       - std::unordered\_set<int> t ; // declare a std::unordered\_set<int> name t
       - t.insert(no); // insert the number “no” in t, by std function std::unordered\_Set::insert()
       - t.find (no); // search the number “no” in t, by std function std::unordered\_Set::find(), return a bool



* + 1. Skip List :
       - SkipList t ; // declare a SkipList name t
       - t.insertElement(no); // insert the number “no” in SkipList t
       - t.searchElement(no); // search the number “no” in the SkipList t, return a bool



* + 1. Treap :
       - TreapNode root ; // declare a TreapNode name root
       - root = insert(root, no); // insert the number “no” in root
       - TreapNode \* res = search(root, no); // search the number “no” in the root, return the node\* where the “no” is, if do not exist, return NULL



* （10%）心得、疑問、與遇到的困難

這此遇到的困難是在辨別與使用網路上找到的程式碼，因為網路上的資料結構和老師所教的資料結構有寫差距，例如: GreekForGreek 網站上的Hash Table (<https://www.sanfoundry.com/cpp-program-hash-tables-singly-linked-list/>) 是需要insert 一個key 跟一個value，查詢時需要用key去搜尋，然而我們在課堂上所學的是insert一個value，讓hash table自己跑hash function找到對影的key，儲存在資料結構上，而search 是直接用value查詢。

後來我看到了c++ 內建的STL，set、map、unorderer\_set、unordered\_map等等，研究了許久，發現他們都有hash table的性質，其中map 是如上述GreekForGreeks 的Hash Table一樣，有key 跟 value，而set是只有value，感覺set或許是比較理想的實作hash table的方式，在經過多次和同學討論後，我們在課堂下課和老師詢問，最終確定可以用unordered\_set實作。在這個過程中，我多方查詢資料、與同學討論，讓我受益良多。

另外，在寫array of sorted array程式碼時，我在網路上查詢到很多演算法已做成std library的function了，例如 : binarySearch() 、 merge()，因此在寫這份程式碼時，我覺得我像站在巨人的肩膀上，寫得很快 !

但我在每次要新增一個空得array時卡了很久，最後發現只需要先push\_back 再clean一次，就可以得到一個空得array了。不過這個方法讓我的程式碼看起來有點冗長，我想我還會多向同學請教一下他們的寫法 !