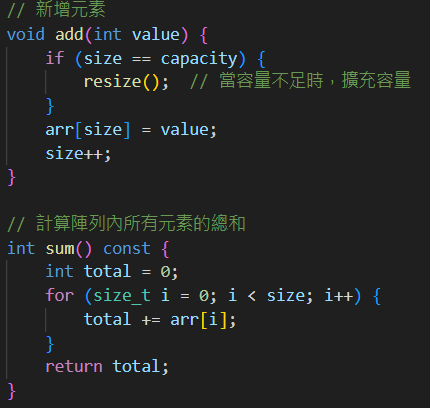
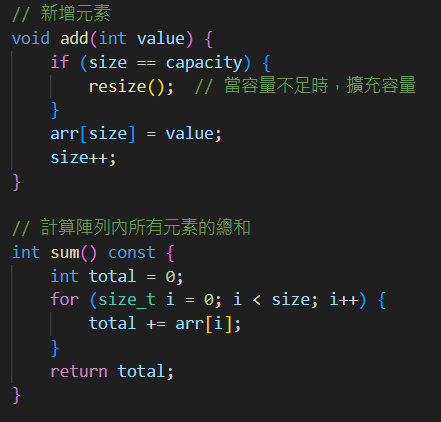
**程式碼運作邏輯設計：**

利用3個不同的腳本(.py)執行四個分別的資料結構 ( DA , DA++ , LL , LL+, cpp)，3個腳本分別負責跑出三張折線圖。

詳細完整程式碼、成果請至：<https://github.com/spaces-lalala/2024DS_HW/tree/Hw1/hw1>

**資料結構程式詳細說明：**

1. DynamicArray(DA)，**透過動態分配記憶體來存儲資料**：
2. 建構子(初始化)：Capacity (容量)= 1，Size (當前元素數量)= 0，動態分配了一個大小為 Capacity 的整數陣列，這個陣列存放新增的元素。
3. 新增元素(add)：當 Size == Capacity (亦即容量不足時)，進行擴充容量(resize) -> 開始填充Value並更新Size(Size++)。
4. 擴充容量(resize)：**每次使容量加倍(Capacity\*=2)** -> 分配新的陣列且copy舊的陣列 -> 釋放舊陣列 ->調整陣列指標
5. 計算總和(sum)：造訪每個資料並進行加總
6. DynamicArray++(DA++)，**透過動態分配記憶體來存儲資料**：
7. 建構子(初始化)：Capacity (容量)= 1，Size(當前元素數量)= 0，動態分配了一個大小為 Capacity 的整數陣列，這個陣列存放新增的元素。
8. 新增元素(add)：當 Size == Capacity (亦即容量不足時)，進行擴充容量(resize) -> 開始填充Value並更新Size(Size++)。
9. 擴充容量(resize)：**每次使容量+1(Capacity+=1)** -> 分配新的陣列且copy舊的陣列 -> 釋放舊陣列 ->調整陣列指標
10. 計算總和(sum)：造訪每個資料並進行加總
11. LinkedList(LL)，**動態分配的單向鏈結串列來創建和管理資料**：
12. 節點(Node)：data：存放實際資料的整數值。

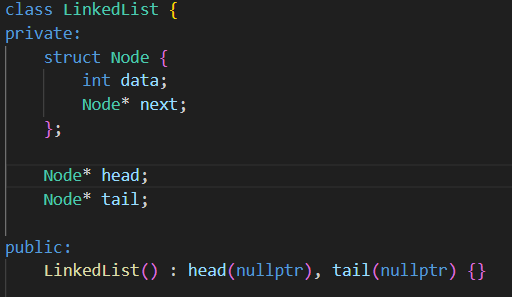
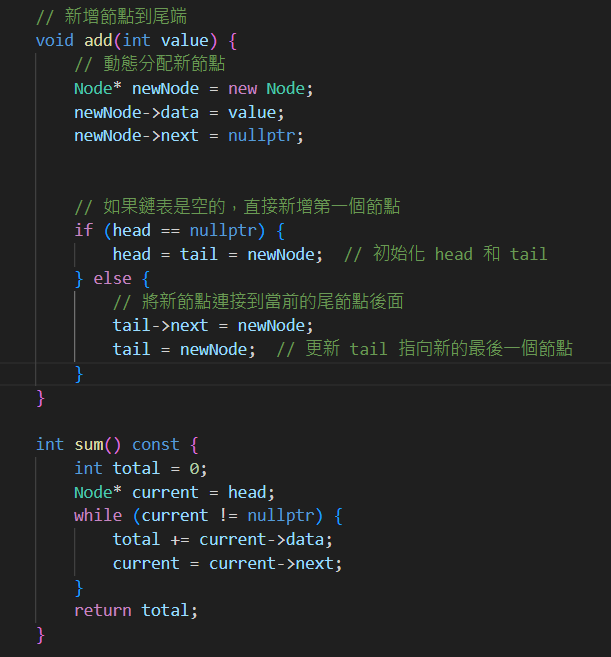
next: 一個指向下一個節點的指標，用於LinkedList中的下一個節點

1. 建構子(初始化)：head (LinkedList第一個節點) 和 tail (LinkedList最後一個節點) 都被設定為 nullptr，表示目前LinkedList為空。
2. 新增元素(add)：**每次新增資料時，會動態分配一個新的節點，並將其加入到鏈結串列的尾端。**

如果鏈結串列是空的，則新節點會成為head (頭節點)和tail(尾節點)。

如果鏈結串列不為空，則將新節點連接到當前tail(尾節點)的後面，並更新尾節點為新節點。

1. 計算總和(sum)：造訪每個資料並進行加總



1. LinkedList++(LL++)，**預先分配了一大塊連續記憶體，並在需要時從這個預先分配的區域中動態分配節點，確保節點按記憶體地址的順序排列**：
2. 節點(Node)：data：存放實際資料的整數值。

next: 一個指向下一個節點的指標，用於LinkedList中的下一個節點

1. 成員變數：

Head：指向鏈結串列的第一個節點。

Tail：指向鏈結串列的最後一個節點。

MemoryBlock：保存一塊已預先分配的連續記憶體區域，用來分配節點。

CurrentOffset：跟蹤當前使用的記憶體偏移量，用來動態分配新節點。

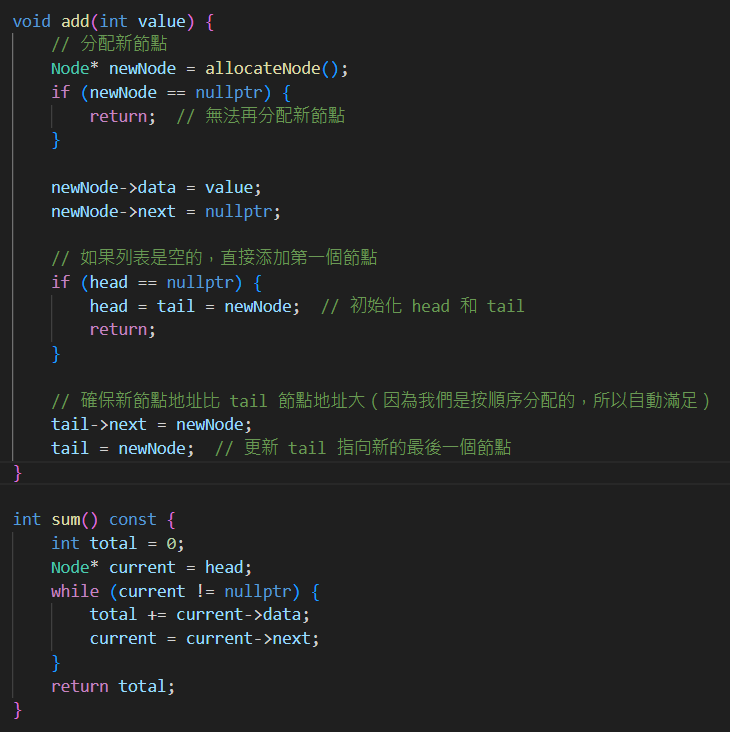
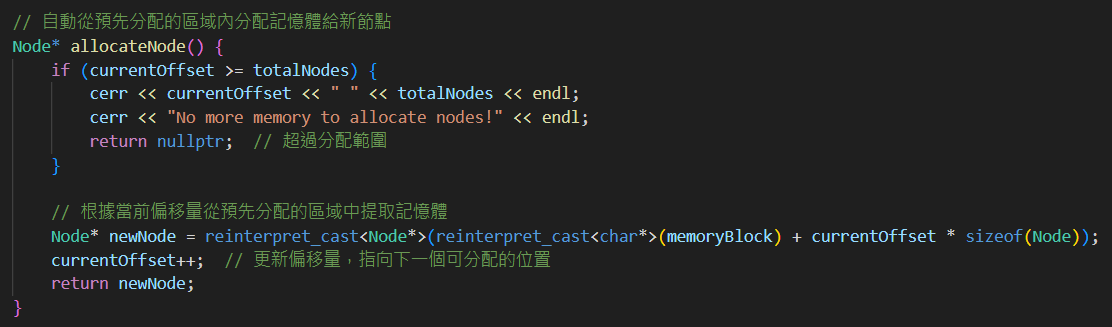
TotalNodes：表示最多可以分配的節點數量。

1. 建構子(初始化)：**使用 malloc 預先分配了 maxNodes 個節點大小的連續記憶體**，currentOffset 用來追踪目前已經分配的記憶體位置，從而確保每次分配的節點都是從這塊區域中順序分配。
2. **節點分配 (allocateNode)** 當新增節點時，程式會根據 currentOffset 來從預先分配的區域中分配一個節點。如果已分配的節點數達到 totalNodes，就不再允許分配更多節點，並回傳 nullptr。每次分配一個節點後，currentOffset 會自動更新，以指向下一個可分配的位置。
3. 新增元素(add)：**使用 allocateNode() 從預先分配的區域分配一個新節點。**

如果鏈結串列是空的，則新節點會作為第一個節點，並同時更新 head 和 tail。

如果鏈結串列不為空，則將新節點添加到尾部，並更新 tail。

1. 計算總和(sum)：造訪每個資料並進行加總



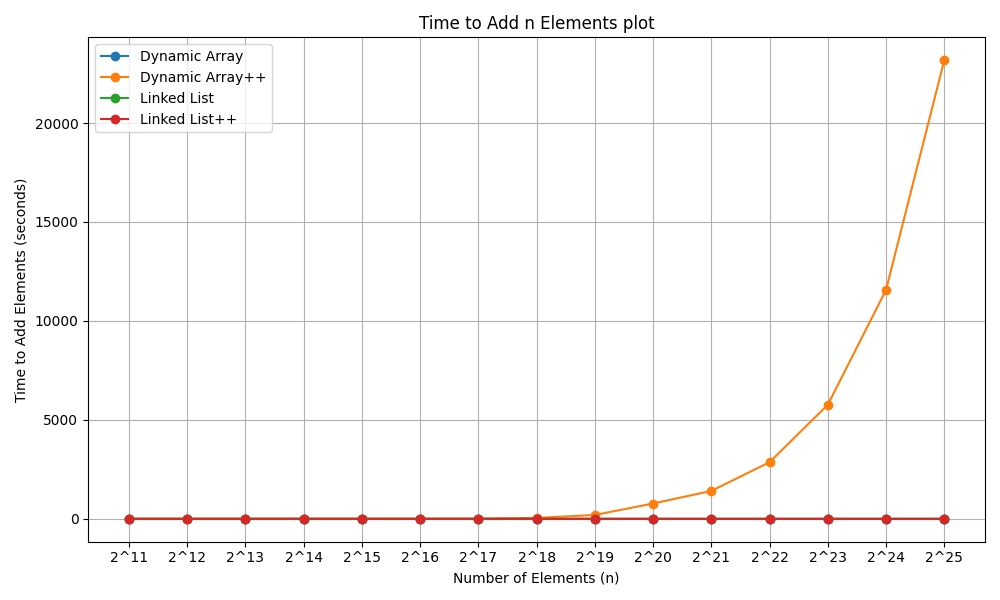
1. 主函式(Main)：在所有的 main() 函式中，程序都通過命令列參數來接收測試的資料量 n 和運行模式 mode。不同的 mode 決定了程式將如何進行效率測試。first/second/third分別做為三張折線圖，再分別做輸出。
2. 腳本：這三個腳本的核心目的是透過執行 C++ 程式來分析不同資料結構（DA、DA++、LL、LL++）的表現。每個腳本都處理不同的測試情境，並且使用 Python 來管理執行過程、處理結果、進行線性回歸補充缺失資料，最後將數據可視化跑出折線圖。

**實驗圖分析：**

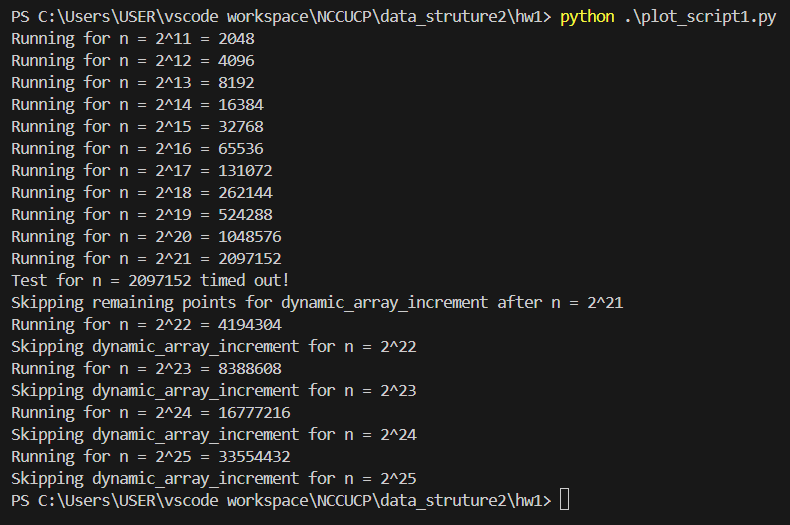
詳細折線圖與數據一樣請至：<https://github.com/spaces-alala/2024DS_HW/tree/Hw1/hw1>

1. 第一張折線圖：新增n筆所需時間

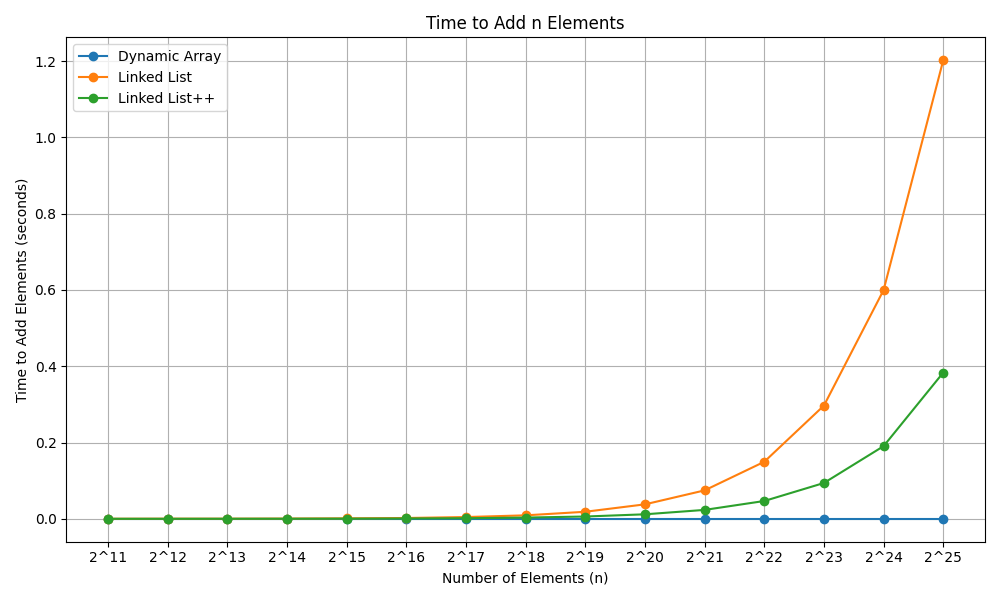
折線圖：



實際運行圖(時間過長有進行skip及預測)：



觀察csv後發現，由於DA++數值過大，壓縮了其他資料結構的顯示，所以我做了一個去除DA++的版本。

折線圖(without DA++)：

分析：

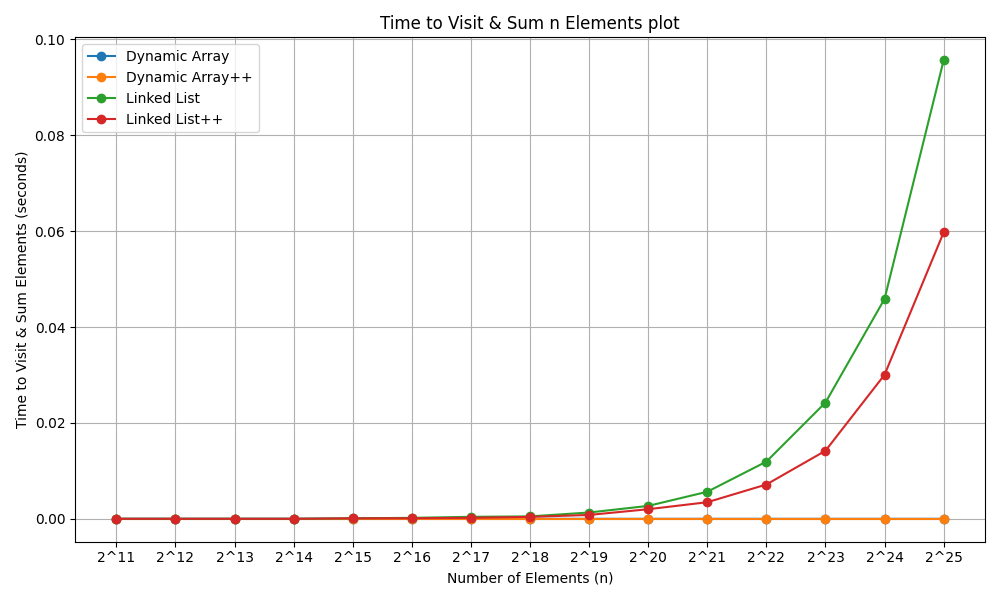
第一個圖 (包含 DA、DA++、LL、LL++)：

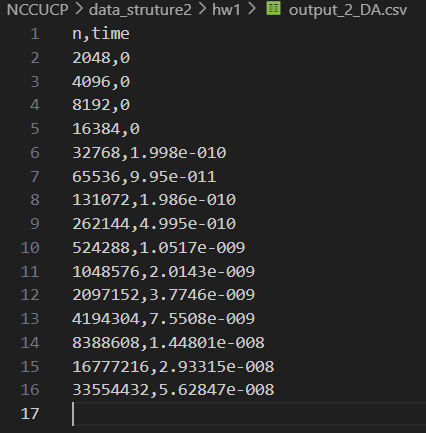
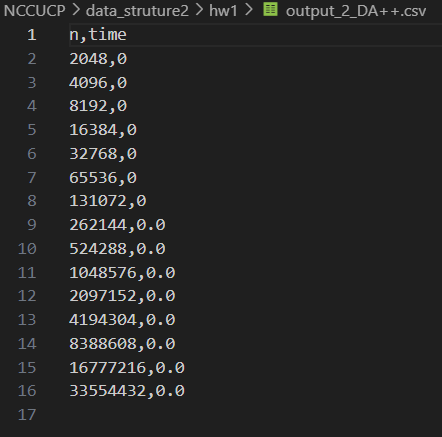
可以主要看出DA++這條線呈現一個急速向上的趨勢。DA++ 的每次容量增加是固定的增量（僅增加 1），而不是倍數增長。這使得當資料量較大時，頻繁的內存重新分配導致所需時間快速上升，效率非常差。

第二個圖 (包含 DA、LL、LL++)：

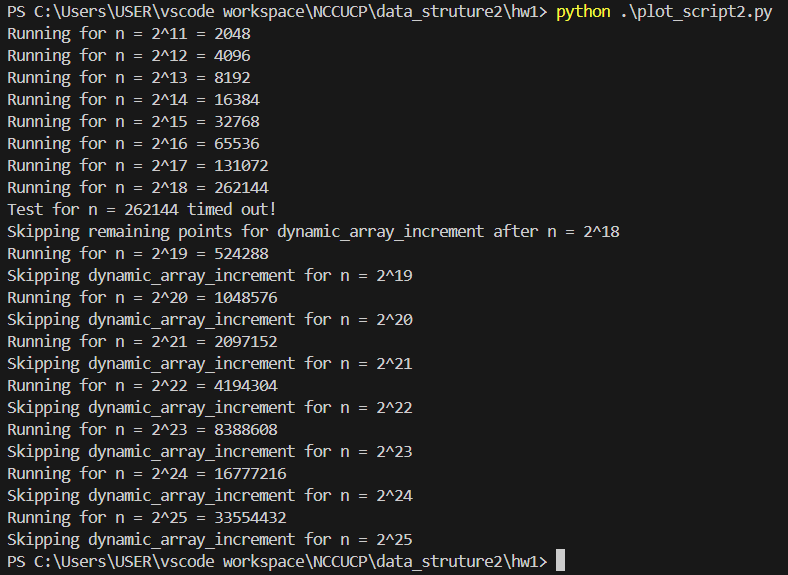
**DA**表現非常穩定，新增資料的時間幾乎保持在極低的時間。這是因為標準動態陣列使用的策略是以倍數成長空間容量，避免頻繁的內存重新分配，因此隨著資料量增加，仍能保持良好的效率。而LL 隨著資料量的增長，線性成長的時間成本是預期之內的，因為 Linked List 的插入操作本身就伴隨著額外的記憶體分配和指標操作。但LL++表現相對LL佳且穩定，由於LL++採取的是預先分配了一大塊連續記憶體，並在需要時從這個預先分配的區域中動態分配節點，可能使其在測試中加快了記憶體操作。

1. 第二張折線圖：新增後加總n筆資料所需時間

折線圖(DA與DA++重疊)：

DA與DA++在加總時間皆相對LL及LL++極小，故在此折線圖呈現重疊且近乎為0。

實際運行圖(時間過長有進行skip及預測)：



分析：

從圖中可以發現DA與DA++的效率非常好，此兩個資料結構以造訪與加總的角度來看其實應沒有太多差距，實際實驗出來的成果也是如此。是由於在DA與DA++裡的記憶體是連續分配的，CPU 的快取能有效命中，遍歷和加總元素的操作效率極高。而對比LL，LL中的元素在記憶體中不是連續存放的，遍歷每個元素時需要跳轉到不同的記憶體位置，這會導致較多的cache misses，大幅增加了加總操作的時間。每次訪問一個節點都需要額外的時間來讀取節點的指標，這使得加總操作相較於陣列慢得多。最後，LL++中的節點是從一塊連續的記憶體中分配的，這意味著 CPU 可以預測到下一個節點的位置，從而提前將其讀取到快取中。這大大提升了快取命中率，減少了讀取每個節點的記憶體延遲所以相對LL會花較少時間，而雖然節點是連續存放的，但LinkedList的基本結構仍然依賴指標來連接節點。在每次遍歷時，程式仍然需要解引用這些指標來訪問下一個節點，這使得鏈結串列相比動態陣列仍然有較大的額外開銷，故所需時間較多。

1. 第三張折線圖：計算每新增第213個資料所需時間