DS_HW1_資訊三_嚴聲遠_111703009

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**：
   a. 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。
   b. 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++)。
   c. 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
   d. 計算總和(sum)：造訪每個資料並進行加總

```
class DynamicArray {
private:
    int* arr;       // 指向陣列的指標
    size_t capacity; // 當前容量
    size_t size;     // 當前元素個數

    // 擴充容量的方法，每次將容量加倍
    void resize() {
        capacity *= 2;                          // 將容量加倍
        int* newArr = new int[capacity];        // 分配新的更大陣列
        for (size_t i = 0; i < size; i++) {
            newArr[i] = arr[i];                 // 拷貝舊陣列的內容
        }
        delete[] arr;                           // 釋放舊陣列
        arr = newArr;                           // 將陣列指標指向新陣列
    }

public:
    // 建構子：初始化陣列，預設容量為 1
    DynamicArray() {
        capacity = 1;
        size = 0;
        arr = new int[capacity];
    }
```

```
// 新增元素
void add(int value) {
    if (size == capacity) {
        resize();  // 當容量不足時，擴充容量
    }
    arr[size] = value;
    size++;
}

// 計算陣列內所有元素的總和
int sum() const {
    int total = 0;
    for (size_t i = 0; i < size; i++) {
        total += arr[i];
    }
    return total;
}
```

2. DynamicArray++(DA++)，**透過動態分配記憶體來存儲資料**：
    a. 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。
    b. 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++)。
    c. 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
    d. 計算總和(sum)：造訪每個資料並進行加總

```cpp
// 動態陣列類別
class DynamicArrayIncrement {
private:
    int* arr;       // 指向陣列的指標
    size_t capacity; // 當前容量
    size_t size;     // 當前元素個數

    // 擴充容量的方法，每次將容量加 1
    void resize() {
        capacity += 1;                          // 每次擴充時僅增加 1
        int* newArr = new int[capacity];        // 分配新的更大陣列
        for (size_t i = 0; i < size; i++) {
            newArr[i] = arr[i];                 // 拷貝舊陣列的內容
        }
        delete[] arr;                           // 釋放舊陣列
        arr = newArr;                           // 將陣列指標指向新陣列
    }

public:
    // 建構子：初始化陣列，預設容量為 1
     DynamicArrayIncrement() {
        capacity = 1;
        size = 0;
        arr = new int[capacity];
    }
```

```cpp
// 新增元素
void add(int value) {
    if (size == capacity) {
        resize();  // 當容量不足時，擴充容量
    }
    arr[size] = value;
    size++;
}

// 計算陣列內所有元素的總和
int sum() const {
    int total = 0;
    for (size_t i = 0; i < size; i++) {
        total += arr[i];
    }
    return total;
}
```

3. LinkedList(LL)，**動態分配的單向鏈結串列來創建和管理資料**：
    a. 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
    b. 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。
    c. 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。

d. 計算總和(sum)：造訪每個資料並進行加總

```
class LinkedList {
private:
    struct Node {
        int data;
        Node* next;
    };

    Node* head;
    Node* tail;

public:
    LinkedList() : head(nullptr), tail(nullptr) {}
```

```
    // 新增節點到尾端
    void add(int value) {
        // 動態分配新節點
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = nullptr;


        // 如果鏈表是空的，直接新增第一個節點
        if (head == nullptr) {
            head = tail = newNode;  // 初始化 head 和 tail
        } else {
            // 將新節點連接到當前的尾節點後面
            tail->next = newNode;
            tail = newNode;  // 更新 tail 指向新的最後一個節點
        }
    }

    int sum() const {
        int total = 0;
        Node* current = head;
        while (current != nullptr) {
            total += current->data;
            current = current->next;
        }
        return total;
    }
```

4. 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**：

   a. 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

   b. 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。

c. 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。

d. **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。

e. 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。

f. 計算總和(sum)：造訪每個資料並進行加總

```
class LinkedListSorted {
private:
    struct Node {
        int data;
        Node* next;
    };

    Node* head;
    Node* tail;
    void* memoryBlock;  // 保存整塊預先分配的記憶體區域
    size_t currentOffset;  // 用於跟踪已使用的記憶體區域
    size_t totalNodes;  // 總共能分配的節點數

public:
    LinkedListSorted(size_t maxNodes) : head(nullptr), tail(nullptr), currentOffset(0), totalNodes(maxNodes) {
        // 預先分配足夠大的連續記憶體區域
        memoryBlock = malloc(maxNodes * sizeof(Node));
        if (memoryBlock == nullptr) {
            cerr << "Memory allocation failed!" << endl;
            exit(1);
        }
    }
```

```
// 自動從預先分配的區域內分配記憶體給新節點
Node* allocateNode() {
    if (currentOffset >= totalNodes) {
        cerr << currentOffset << " " << totalNodes << endl;
        cerr << "No more memory to allocate nodes!" << endl;
        return nullptr;  // 超過分配範圍
    }

    // 根據當前偏移量從預先分配的區域中提取記憶體
    Node* newNode = reinterpret_cast<Node*>(reinterpret_cast<char*>(memoryBlock) + currentOffset * sizeof(Node));
    currentOffset++;  // 更新偏移量，指向下一個可分配的位置
    return newNode;
}
```

```
void add(int value) {
    // 分配新節點
    Node* newNode = allocateNode();
    if (newNode == nullptr) {
        return;  // 無法再分配新節點
    }

    newNode->data = value;
    newNode->next = nullptr;

    // 如果列表是空的，直接添加第一個節點
    if (head == nullptr) {
        head = tail = newNode;  // 初始化 head 和 tail
        return;
    }

    // 確保新節點地址比 tail 節點地址大（因為我們是按順序分配的，所以自動滿足）
    tail->next = newNode;
    tail = newNode;  // 更新 tail 指向新的最後一個節點
}

int sum() const {
    int total = 0;
    Node* current = head;
    while (current != nullptr) {
        total += current->data;
        current = current->next;
    }
    return total;
}
```

5. 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。
6. 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。
7. 預測：

   利用線性回歸來填補因測試超時而缺失的資料點。取現有數據後使用 scikit-learn 的 LinearRegression 模型進行回歸訓練。

```python
# 紀錄哪些資料點超時並需要預測
missing_points = {
    "dynamic_array": {"n": [], "predicted_time": []},
    "dynamic_array_increment": {"n": [], "predicted_time": []},
    "linked_list": {"n": [], "predicted_time": []},
    "linked_list_sorted": {"n": [], "predicted_time": []}
}

# 記錄哪些資料結構已經跳過剩餘計算
skipped_structures = {
    "dynamic_array": False,
    "dynamic_array_increment": False,
    "linked_list": False,
    "linked_list_sorted": False
}
```

```python
# 執行所有 C++ 程式並生成資料
for name, executable in cpp_programs.items():
    # 如果該資料結構之前已經被跳過，則不再執行，直接將後續點加入 missing_points
    if skipped_structures[name]:
        print(f"Skipping {name} for n = 2^{k}")
        missing_points[name]["n"].append(n_value)
        continue  # 繼續處理下一個資料結構

    # 執行程式並檢查是否成功
    success = run_cpp_program(executable, n_value, output_files[name], timeout=7200)

    # 若超過時間限制，則記錄需要預測的點並標記為跳過後續
    if not success:
        print(f"Skipping remaining points for {name} after n = 2^{k}")
        missing_points[name]["n"].append(n_value)
        skipped_structures[name] = True  # 標記此資料結構為跳過
```

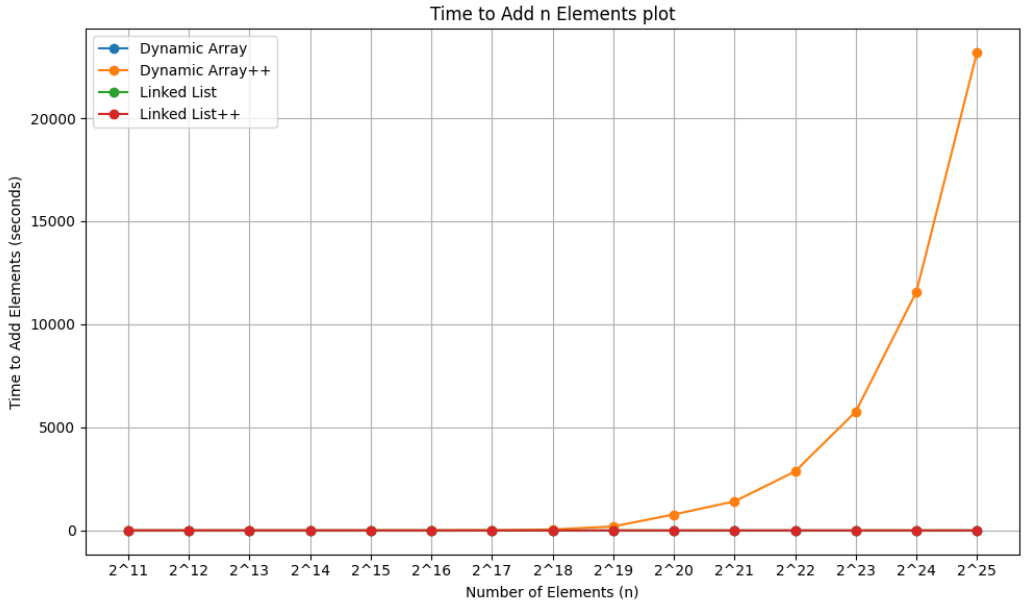
```python
# 預測缺失資料點並將預測值加入 CSV
for name, output_file in output_files.items():
    # 使用現有資料進行回歸模型預測
    model = predict_missing_points(output_file)

    if model and missing_points[name]["n"]:
        for n_value in missing_points[name]["n"]:
            predicted_time = model.predict(np.array([[n_value]]))[0]
            with open(output_file, 'a') as f:
                f.write(f"{n_value},{predicted_time}\n")

            # 保存預測資料點
            missing_points[name]["predicted_time"].append(predicted_time)
    else:
        missing_points[name] = {"n": [], "predicted_time": []}
```

實驗圖分析：

詳細折線圖與數據同前文連結：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：

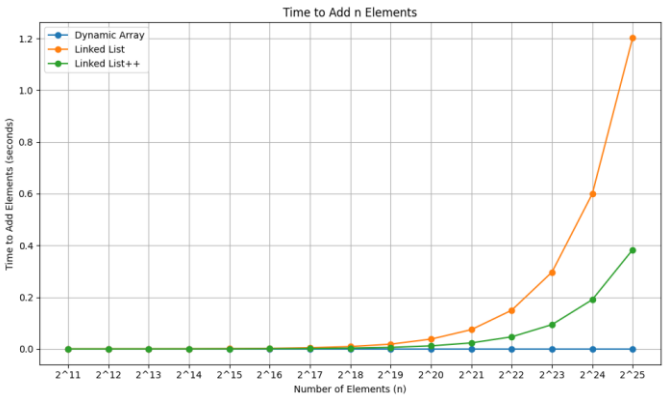


實際運行圖(時間過長有進行 skip 及預測)：

```
PS C:\Users\USER\vscode workspace\NCCUCP\data_struture2\hw1> python .\plot_script1.py
Running for n = 2^11 = 2048
Running for n = 2^12 = 4096
Running for n = 2^13 = 8192
Running for n = 2^14 = 16384
Running for n = 2^15 = 32768
Running for n = 2^16 = 65536
Running for n = 2^17 = 131072
Running for n = 2^18 = 262144
Running for n = 2^19 = 524288
Running for n = 2^20 = 1048576
Running for n = 2^21 = 2097152
Test for n = 2097152 timed out!
Skipping remaining points for dynamic_array_increment after n = 2^21
Running for n = 2^22 = 4194304
Skipping dynamic_array_increment for n = 2^22
Running for n = 2^23 = 8388608
Skipping dynamic_array_increment for n = 2^23
Running for n = 2^24 = 16777216
Skipping dynamic_array_increment for n = 2^24
Running for n = 2^25 = 33554432
Skipping dynamic_array_increment for n = 2^25
PS C:\Users\USER\vscode workspace\NCCUCP\data_struture2\hw1>
```

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++)：

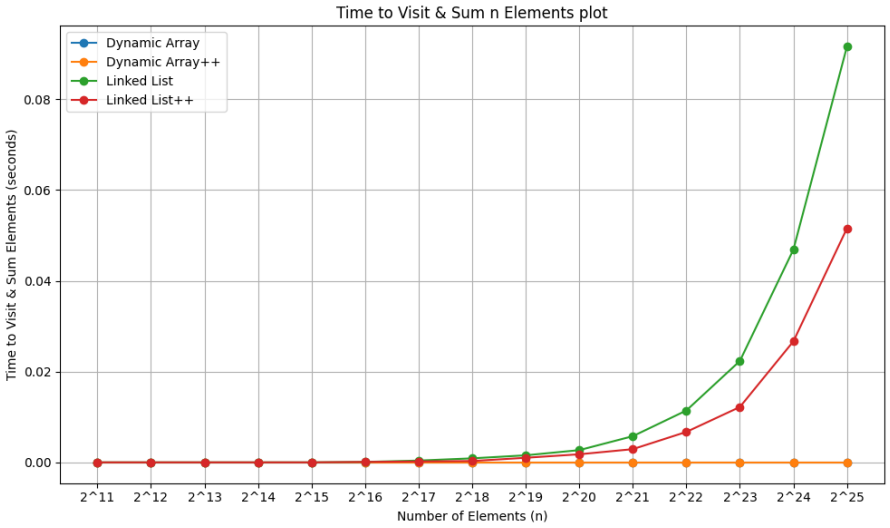
DA 表現非常穩定，新增資料的時間幾乎保持在極低的時間。這是因為標準動態陣列使用的策略是以倍數成長空間容量，避免頻繁的內存重新分配，因此隨著資料量增加，仍能保持良好的效率。而 LL 隨著資料量的增長，線性成長的時間成本是預期之內的，因為 Linked List 的插入操作本身就伴隨著額外的記憶

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。

2. 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。



實際數據(DA、DA++)

NCCUCP > data_struture2 > hw1 > output_2_DA.csv

```
n,time
2048,0
4096,0
8192,0
16384,0
32768,0
65536,0
131072,0
262144,1.001e-010
524288,4.637e-010
1048576,7.61e-010
2097152,1.6142e-009
4194304,3.2213e-009
8388608,6.1225e-009
16777216,1.26389e-008
33554432,2.51547e-008
```

NCCUCP > data_struture2 > hw1 > output_2_DA++.csv

```
n,time
2048,0
4096,0
8192,0
16384,0
32768,1e-010
65536,0
131072,5.849324804548685e-11
262144,1.0852878464818763e-10
524288,2.085998578535892e-10
1048576,4.0874200426439233e-10
2097152,8.090262970859985e-10
4194304,1.6095948827292111e-09
8388608,3.210732054015636e-09
16777216,6.413006396588486e-09
33554432,1.2817555081734186e-08
```

實際運行圖(時間過長有進行 skip 及預測)：

```
PS C:\Users\USER\vscode workspace\NCCUCP\data_struture2\hw1> python .\plot_script2.py
Running for n = 2^11 = 2048
Running for n = 2^12 = 4096
Running for n = 2^13 = 8192
Running for n = 2^14 = 16384
Running for n = 2^15 = 32768
Running for n = 2^16 = 65536
Running for n = 2^17 = 131072
Running for n = 2^18 = 262144
Test for n = 262144 timed out!
Skipping remaining points for dynamic_array_increment after n = 2^18
Running for n = 2^19 = 524288
Skipping dynamic_array_increment for n = 2^19
Running for n = 2^20 = 1048576
Skipping dynamic_array_increment for n = 2^20
Running for n = 2^21 = 2097152
Skipping dynamic_array_increment for n = 2^21
Running for n = 2^22 = 4194304
Skipping dynamic_array_increment for n = 2^22
Running for n = 2^23 = 8388608
Skipping dynamic_array_increment for n = 2^23
Running for n = 2^24 = 16777216
Skipping dynamic_array_increment for n = 2^24
Running for n = 2^25 = 33554432
Skipping dynamic_array_increment for n = 2^25
PS C:\Users\USER\vscode workspace\NCCUCP\data_struture2\hw1>
```

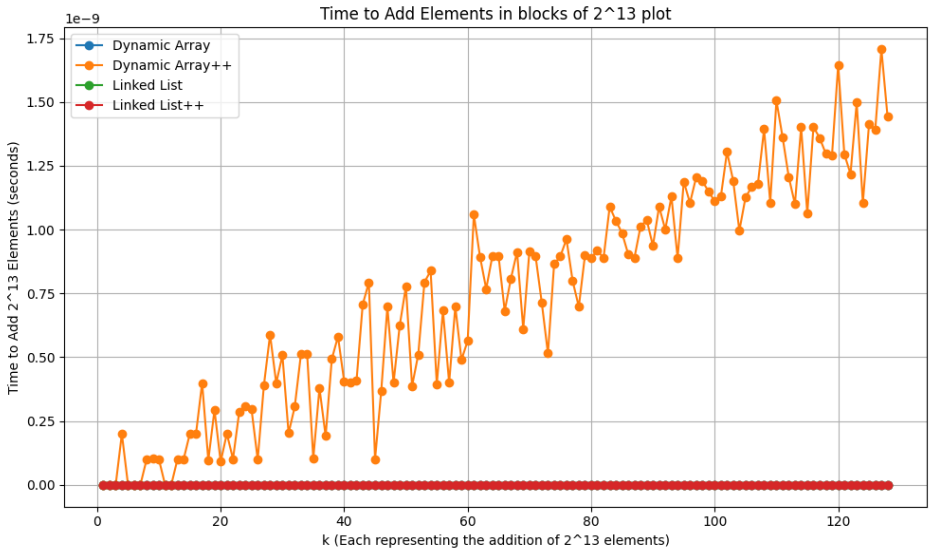
分析：

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。

3. 第三張折線圖：計算每新增第 2^{13} 個資料所需時間

折線圖：



除了 DA++以外，DA、LL、LL++所有數據接近乎於 0，故三條線重疊。

節錄實際數據：

NCCUCP > data_struture2 > hw1 > output_3_DA.csv

```
100,0
101,0
102,0
103,0
104,0
105,0
106,0
107,0
108,0
109,0
110,0
111,0
112,0
113,0
114,0
115,0
116,0
117,0
118,0
119,0
120,0
121,0
122,0
123,0
124,0
125,0
126,0
127,0
128,0
```

NCCUCP > data_struture2 > hw1 > output_3_DA++.csv

```
100,1.1108e-009
101,1.1288e-009
102,1.3047e-009
103,1.1911e-009
104,9.958e-010
105,1.1265e-009
106,1.1669e-009
107,1.1786e-009
108,1.3935e-009
109,1.1063e-009
110,1.5053e-009
111,1.3616e-009
112,1.206e-009
113,1.0997e-009
114,1.4013e-009
115,1.0636e-009
116,1.4019e-009
117,1.3592e-009
118,1.297e-009
119,1.2906e-009
120,1.6439e-009
121,1.2955e-009
122,1.2178e-009
123,1.4976e-009
124,1.1047e-009
125,1.4139e-009
126,1.39e-009
127,1.7064e-009
128,1.4413e-009
```

NCCUCP > data_struture2 > hw1 > output_3_LL.csv

```
100,0
101,0
102,0
103,0
104,0
105,0
106,0
107,0
108,0
109,0
110,0
111,0
112,0
113,0
114,0
115,0
116,0
117,0
118,0
119,0
120,0
121,0
122,0
123,0
124,0
125,0
126,0
127,0
128,0
```

NCCUCP > data_struture2 > hw1 > output_3_LL++.csv

```
100,0
101,0
102,0
103,0
104,0
105,0
106,0
107,0
108,0
109,0
110,0
111,0
112,0
113,0
114,0
115,0
116,0
117,0
118,0
119,0
120,0
121,0
122,0
123,0
124,0
125,0
126,0
127,0
128,0
```

分析：四個資料結構所花的時間其實皆相當接近 0，只有 DA++能看出所需時間逐漸向上的趨勢，由於 DA++每次都需要進行一次重新分配(capacity++)，隨著

元素數量的增長，時間逐漸增加。DA、LL、LL++在新增上大部分不需要進行頻繁的記憶體擴充操作。在這些情況下，新增元素的操作只需要簡單地將元素插入現有的記憶體中，時間開銷相對小。

問題：

1. LinkedList實作方法：

   嘗試實作linkedlist++時，第一個想法是想要利用一直讓系統自己隨機找位置並檢查位置直到比目前tail大。但僅做到要插入第7個資料時就會陷入無限迴圈，系統會一直找尋相同的幾個位置，但都沒有比tail大所以會掉入無限迴圈而無法Add。第二個方法是先創建好所需的記憶體在裡面按照順序進行Add並有指標指向，雖然此方法可實作，但若回顧兩種方法會發現，若兩種方法皆可實作且以新增資料的角度來看，第一種方法會比正常的LinkedList久，而第二種則較少。雖然可能都是LL++，但創建的方式也影響了實驗的結果，而不是資料結構本身。

2. DA++在第三章折線圖的震盪現象：

   由於DA++每次Capacity++，我原先認為DA++的線會一路向上呈現線性而不是有上下震盪的呈現，查詢GPT得到「雖然每次擴充只增加 1 個單位，但操作系統的記憶體分配器在管理小塊記憶體時，可能會有不同的處理方式。例如，當動態陣列頻繁地在記憶體中進行小範圍的擴充時，系統可能會分配到不同位置的記憶體塊，這會導致額外的記憶體分配開銷，有時可能比之前的開銷大，導致這種震盪現象。 2. Cache 行為與對齊問題 CPU 在存取記憶體時會依賴快取，當資料位置不對齊或記憶體區塊換行到不同的快取線時，存取速度可能會有所不同。每次新增一個元素時，當快取快滿時，CPU 可能會進行額外的快取管理或清除操作，這會導致時間上的變化。」，看起來很合理但

我無法確認是否還有其他如背景程式、隨機數等等外部問題影響，由於就算有震盪，數字還是極小的，任何小影響都有可能影響結果。