**Ch8 鏈結串列 Linked List**

在講解完 vector 後，緊接著就要來介紹第二種 STL 中的資料結構，也就是鏈結串列 linked list。

課程大綱

A. 鏈結串列 Linked List 簡介

B. 鏈結串列的基本操作

C. 不同種 Linked List

D. Linked List @ C++ STL

首先，一樣會先簡介什麼是鏈結串列，接下來，會介紹鏈結串列有哪些基本操作需要學習，再來，會介紹除了最基本的單向鏈結串列外，還有哪些不同類型的鏈結串列，它們各自有什麼樣的特色和強項，最後，一樣會學習使用 C++ STL 裡的鏈結串列。

# 第一節：Linked List 簡介

首先，來看什麼是鏈結串列，以及它的實作原理。

「陣列」與「向量 Vector」資料都放在記憶體的連續空間

A. 難以擴充、擴充耗時

B. 每次擴充大小，都可能伴隨資料遷移

C. 無法事先確定需要的大小，程式是邊執行邊新增

一旦放棄資料放在連續記憶體空間的特性，不要把資料連續放著，新增就會變得比較方便。

這就像平常把東西拿回家，如果每樣東西都有其固定的位置，新增就很麻煩，要把所有東西的位置調整後，才能放新的東西；實際上更常做的是把新的東西往房間裡隨便丟。鏈結串列就是這樣，看記憶體哪邊有空位就直接放入新的資料，不要求它與其他筆資料位置連續。

## 1. 鏈結串列 Linked List 的特色

A. 需要插入一筆新資料時，放在記憶體中的任意位置

B. 插入 / 刪除資料時，不需修改其他資料的位置

a. 不需根據其他筆資料調整，一經插入位置就不再更改

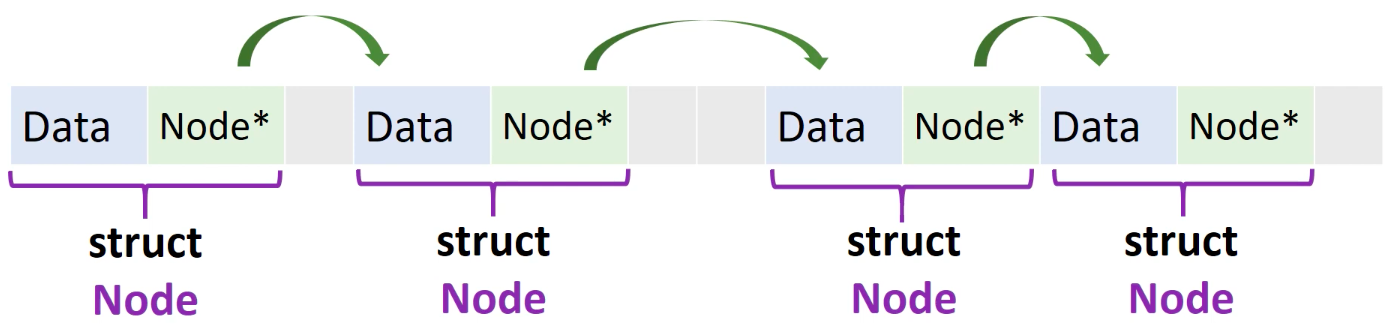
b. 不像向量會面臨到該連續空間不夠大的問題

C. 利用指標紀錄資料的記憶體位置，把每一筆資料串連起來

a. 但也因此不像向量只要往右移動就可以找到下一筆資料

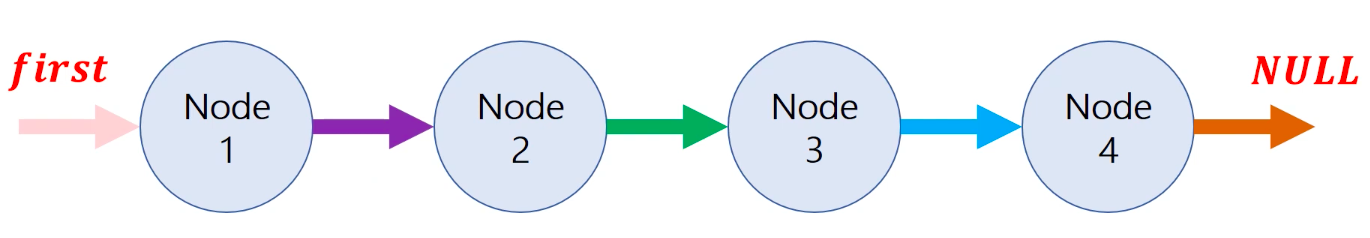
## 2. 節點 node

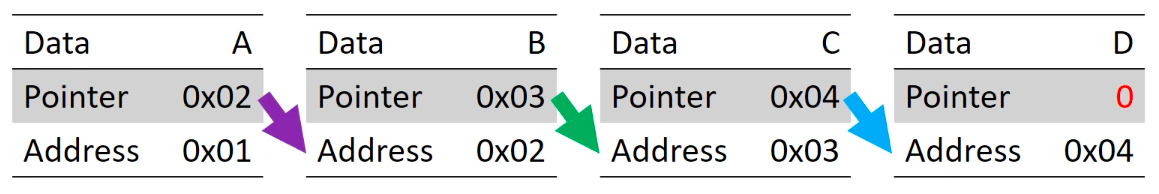
鏈結串列的基本單位叫做「節點 node」，一個 node 中會有兩筆資料，一個存放實際要使用的資料 data，另一個是一個指標 Node\* 指向下一筆資料的位置。



鏈結串列的特色就是透過指標與指標把 node 串聯在一起，不過這會耗費額外空間，因為儲存 node 的空間會比單純儲存資料來的大。

每個節點中都有一個 Data 與一個 Pointer：





第一筆資料 A 的 node 中所含的指標代表第二個節點的位置，我們可以依此前往第二筆資料 B；到了 B 所在的第二個 node 之後，它所含的指標代表第三筆資料在哪個位置；到了第四個 node 之後，發現它的 pointer 是空指標，這代表它是這個鏈結串列中的最後一筆資料。

這種運作方式就像在玩 RPG 的時候，每個關主會說明下一關在哪裡。

## 3. 鏈結串列的開始與結束

A. 何時開始：first 指標紀錄第一個節點的指標

B. 何時結束：空指標，當發現指向空指標，就代表鏈結串列結束



## 4. 鏈結串列 Linked List 的優點與缺點

(1) 優點 Pros

A. 新增 / 刪除資料比陣列簡單

a. 只需要改寫 個節點的資料內容

b. 向量（陣列）中則需要把後面每一筆資料往後移，需要

B. 資料數量是彈性的，不像陣列時常需要搬動資料

(2) 缺點 Cons

A. 要刪除/新增特定位置的資料，須先執行 的搜尋

* 要找到第 5 筆資料，需要從第一筆開始，它指明第二筆在哪裡

後，再去看第三筆、第四筆在哪裡，直到尋訪至第五筆

B. 不支援索引值取值

* 索引值要求連續的記憶體位置

C. 相較陣列，需要「額外的空間儲存指標」以標示節點位置

## 5. 鏈結串列的使用時機

A. 資料的總數量是浮動的，因為每次更動不需移動資料，比較節省時間

B. 新增/刪除資料的操作頻繁

C. 很少「查詢」或「走訪」所有資料的需求

a. 向量：一直向右移動即可

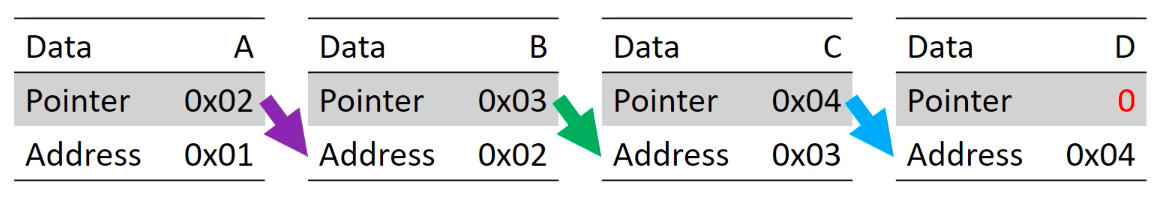
b. 鏈結串列：需要先知道下一筆資料位置在哪裡，再移動到下一筆資料，

會比向量步驟多且較慢

## 6. 鏈結串列的架構

(1) Node 結構

A. Data：存放資料

 B. Pointer：指向另一個 Node

Node 中有兩筆資料，第一筆是資料 Data，因為希望它可以存放各種資料型別，所以採用結構模板撰寫。

|  |  |
| --- | --- |
| Node 結構 | |
| 1  2  3  4  5 | template <typename T>  struct Node {  T Data;  Node \*Next;  }; |

這裡我們用結構 Struct 寫 Node，但用類別 Class 也可以。

(2) Linked List 類別

A. First：指向第一個 Node

B. 各種操作函式：走訪、搜尋、新增、刪除、反轉、...

|  |  |
| --- | --- |
| Linked\_List 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | template <typename T>  class Linked\_List{  private:  Node<T>\* First;  public:  // 建構式  Linked\_List();  // Print\_List 可以把 Linked List 裡面的資料都印出來  // 實際上 STL 中裡的鏈結串列沒有這個功能  void Print\_List();  // 從鏈結串列中找到特定的資料  int Search\_List(T);  }; |

鏈結串列 Linked List 類別同樣採用類別模板以利儲存各種資料，private 成員 First 指標指向第一個 Node。

(3) 指標的宣告與釋放

C 語言

A. 指標 = (指標) malloc(空間大小);

向作業系統要空間後，回傳該空間的記憶體位置給指標

B. free(指標);

a. 釋放指標所指到實體的記憶體空間，相當於刪除該實體

b. 若忘記使用會造成 memory leakage，這塊空間卡著無法再被使用，

程式執行過程中記憶體會不斷洩漏出去

C. 指標 = 0;

a. 讓指標不指到任何一個實體，也就是空指標

b. 要先確認不是空指標才能使用 \* 取值運算

C++

A. 指標 = new 類別(初始化);

宣告出類別/結構實體後，回傳該實體的記憶體位置給指標

B. delete 指標;

釋放指標所指到實體的記憶體空間，相當於刪除該實體

C. 指標 = nullptr;

a. 讓指標不指到任何一個實體（空指標）

b. C++11 之後支援，比較易讀，可以和一般的 0 做區分

(4) malloc 與 new 的差異

A. new 會創造出物件實體，並呼叫建構式

B. malloc 僅配置記憶體空間，不呼叫建構式

創造節點的時候，因為要產生實體，會選擇使用「new」。

## 7. 建立 Node 結構、Linked List 類別，並完成建構式

A. Node 結構

a. Data：存放資料

b. Pointer：指向另一個 Node

B. Linked List 類別

a. First：指向第一個 Node，最一開始串列是空的，First 指到一個空指標

b. 各種操作函式：走訪、搜尋、新增、刪除、反轉...

Linked List 相關的程式碼寫在 Linked List.h 中

(1) Node 結構、Linked List 類別的宣告

|  |  |
| --- | --- |
| Linked List.h | |
| 1  2  3  4  5  6  7  9  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #ifndef LINKED\_LIST\_H\_INCLUDED  #define LINKED\_LIST\_H\_INCLUDED  #include <iostream>  using namespace std;  // Node 結構，也可以放在 Linked List 類別中  template <typename T>  struct Node{  T Data; // 節點的第一個元素：資料 Data  Node\* Next; // 節點的第二個元素：指標 Next  };  // Linked\_List 類別  template <typename T>  class Linked\_list{  private:  Node<T>\* First;  public:  // 建構式  Linked\_list();  };  … // 函式的定義寫在這裡  #endif // LINKED\_LIST\_H\_INCLUDED |

(2) 建構式

|  |  |
| --- | --- |
| 建構式 | |
| 1  2  3  4  5 | template <typename T>  Linked\_list<T>::Linked\_List(){  // 初始化 First，設定為空指標  First = nullptr;  } |

(3) 測試 Linked List

在 main.cpp 中測試剛才寫好的 Linked\_list 類別：

|  |  |
| --- | --- |
| 測試 Linked List | |
| 1  2  3  4  5  6  7  8  9  10 | #include <iostream>  #include "Linked\_List.h"  using namespace std;  int main(){  // 注意變數不要命名成小寫開頭的 list，會與 STL 下的型態混淆  Linked\_List<int> List;  return 0;  } |

# 第二節：鏈結串列的基礎操作

## 1. 走訪鏈結串列

走訪鏈結串列的基本邏輯是從 First 開始一路走到空指標。

完成下列 Linked List 內的函式：

A. Print\_List：印出所有 Linked List 內的資料 Data

B. Search\_List(T)

a. 在 Linked List 內找尋特定資料

b. 回傳該特定資料位在第幾筆

c. 有多筆符合時，只需回傳找到的資料中的「第一筆」

稍後完成 Linked List 中的迭代器後，會再把 Search\_List 的回傳結果型態改為迭代器（而非這裡回傳的整數），因為若回傳一個整數 index，只會知道位在第幾筆，實際取用還是需要經過尋訪才能到達，回傳迭代器則可以直接被使用。

在很多使用 Linked List 寫的資料結構裡，知道某筆資料是「第幾筆」並沒有意義，要取得指向該筆資料的迭代器才能直接進行存取，這也是比起索引值來說更常使用迭代器的原因。

|  |  |
| --- | --- |
| Print\_List 與 Search\_List 的宣告 | |
| 1  2  3  4  5  6  7  8  9 | template <typename T>  class Linked\_list{  private:  Node<T>\* First;  public:  Linked\_list();  void Print\_List();  int Search\_List(T);  }; |

(1) Print\_List 印出所有資料

實作 Print\_List 主要的目的是練習如何「走訪每個節點」，到了每個節點後，裡面的 pointer 都會指明下一個節點的位置。

|  |  |
| --- | --- |
| 印出所有資料：Print\_List | |
| 1  2  3  4  5  6  7  9  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | template <typename T>  void Linked\_List<T>::Print\_List(){  // 例外處理：空 List  if (First == nullptr){  cout << "This list is empty!" << endl;  // 提前結束函式  return ;  }  // 一般情形  Node<T>\* current = First; // current 從鏈結串列的第一筆資料開始  cout << "Data:";    // 還有下一筆資料時，current 繼續往下一個節點移動  while (current->Next != nullptr){  // 印出當前 current 中的資料  cout << current->Data << " ";  // 將 current 往後移動一筆資料  current = current->Next;  }    // 記得還要印出最後一筆資料  // 最後一筆資料的 Next 是空指標，沒有在上面的迴圈裡執行  cout << current->Data << endl;  } |

(2) Search\_List 找尋特定資料

|  |  |
| --- | --- |
| 找尋特定資料：Search List | |
| 1  2  3  4  5  6  7  9  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | template <typename T>  void Linked\_List<T>::Search\_List(T target){  // 例外處理：空 list，回傳 -1 代表沒有找到  if (First == nullptr)  return -1;  // 一般情形  int index = 0;  // 利用 current 指標遍歷陣列  Node<T>\* current = First;  while (current->Next != nullptr){  // 資料與 target 相同時回傳 index  if(current->Data == target)  return index;  // 否則繼續往後執行  current = current->Next;  index++;  }    // 記得檢查最後一筆資料  if(current->Data == target)  return index;  // 如果執行到尾端都沒有找到，同樣回傳 -1  else  return -1;  } |

## 2. 鏈結串列的其它基本操作

接下來，要在鏈結串列裡新增一些基本操作的函式。

A. 新增 / 插入

a. 在特定位置新增資料

b. 在首項新增資料

c. 在尾端新增資料

B. 刪除

a. 在特定位置刪除資料

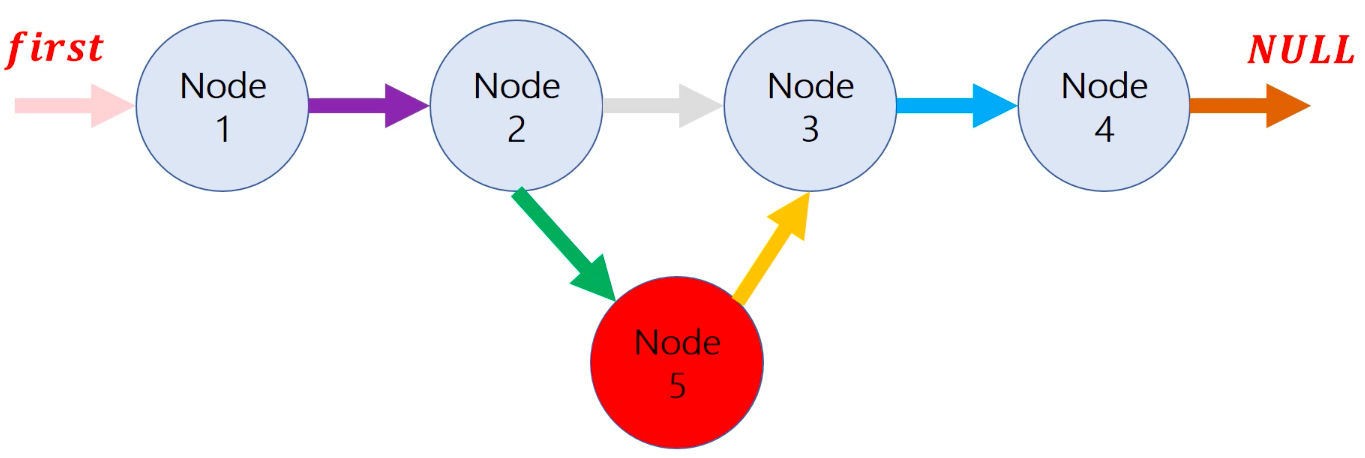
b. 在首項刪除資料

c. 在尾端刪除資料

C. 反轉鏈結串列

D. 清空所有資料

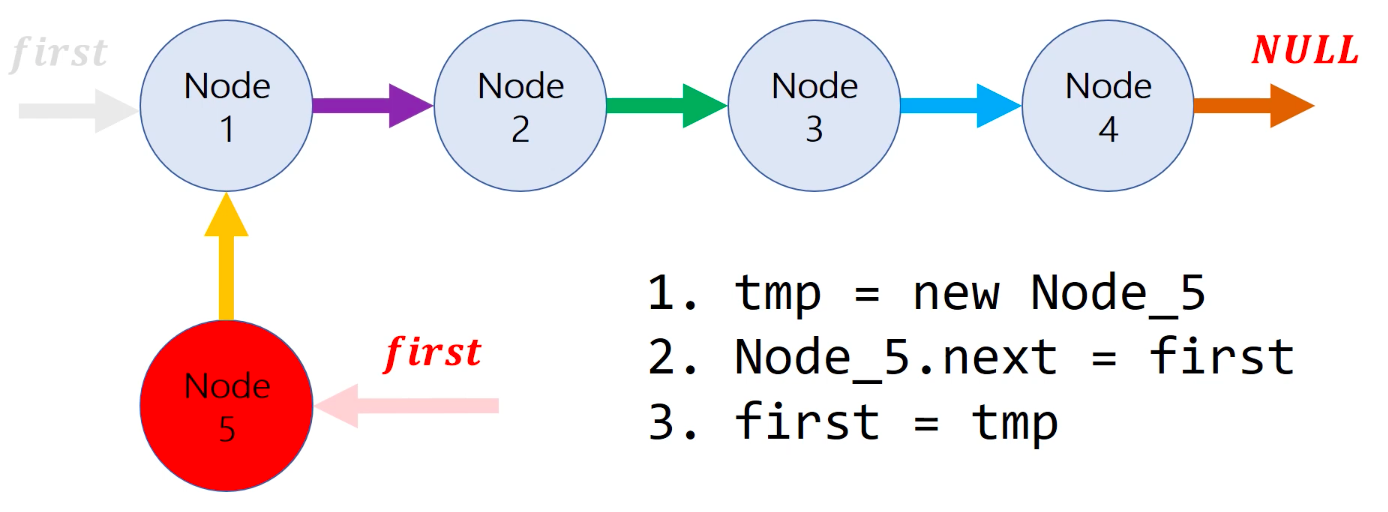
(1) 新增元素至特定位置

要新增一筆資料到鏈結串列上，就要新增一個節點，比方說，要在鏈結串列的節點二和節點三間新增一筆資料，叫節點五：

|  |  |
| --- | --- |
| A. tmp = new Node\_5  B. Node\_5.next = Node\_2.next  C. Node\_2.next = tmp | // 用 new 開出一個新節點，並把位置回傳給 tmp  // 讓第五個節點的 next 指標指向原本 Node\_2 的 next（Node 3）  // 把第二個節點 Node2 的 next 改為指向 Node 5 |

不管鏈結串列有多長，要新增一筆資料需要的時間都是固定的。

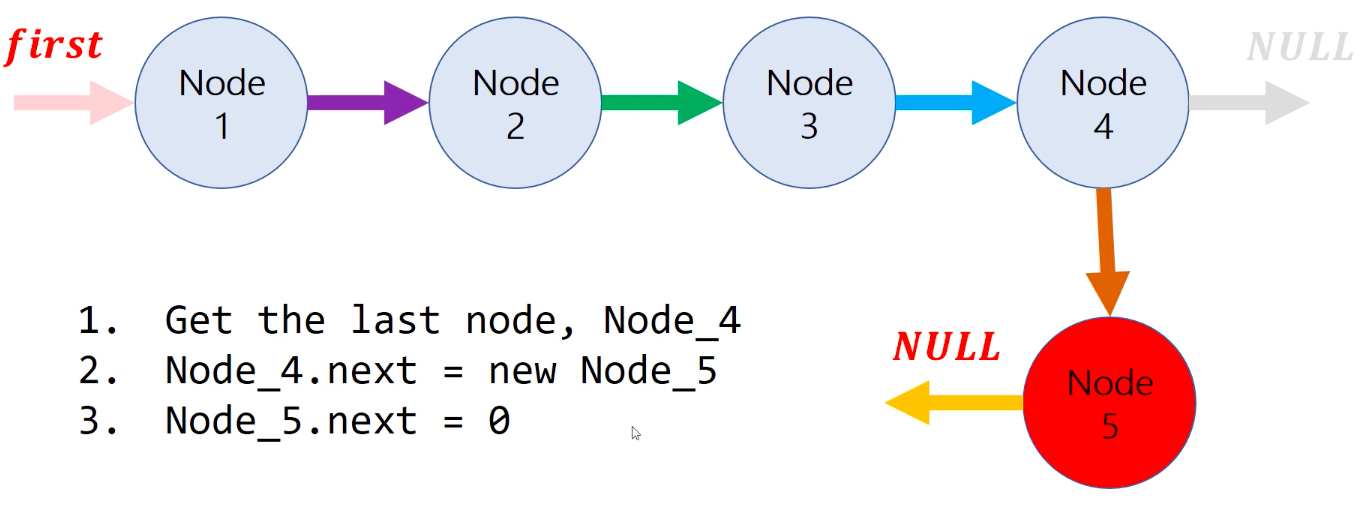
(2) 新增頭項（第一筆資料）



|  |  |
| --- | --- |
| A. tmp = new Node\_5  B. Node\_5.next = first  C. first = tmp | // 用 new 開出一個新節點  // 新增後，Node 5 會是第一筆資料，因此它的 next 要指向原本的第一筆資料 Node 1  // 讓 first 指向 Node 5 |

這裡不能用剛剛的新增函式！因為會需要更動 first 指標。

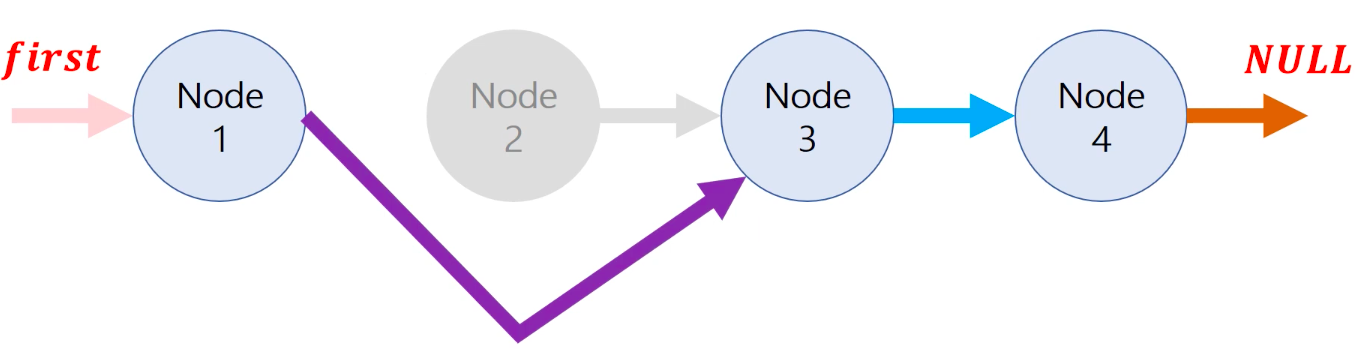
(3) 新增尾項（尾端的資料）



|  |  |
| --- | --- |
| A. Get the last node, Node\_4  B. Node\_4.next = new Node\_5  C. Node\_5.next = nullptr | // 取得最尾端的資料位置 Node 4  // 宣告新的 Node 5，並且讓 Node 4 的 next 指向這個新的 Node 5  // 設定 Node 5 的 next 為空指標 |

新增尾項則「可以」用「新增到特定位置」的函式。

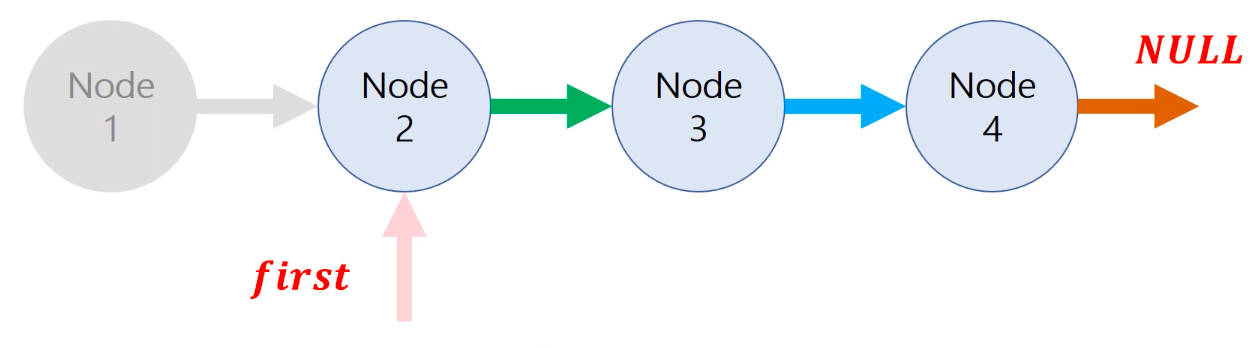
(4) 刪除特定位置上的元素



如果直接把 Node\_1.next（Node\_2）刪掉，那 Node\_3 就再也無法存取，因此在刪掉前，要先把 Node\_3（Node\_2.next）的位置存在 tmp 裡。

|  |  |
| --- | --- |
| 1. tmp = Node\_2.next  2. delete Node\_1.next  3. Node\_1.next = tmp | // 先把 Node 3 的位置存起來  // 把 Node 2 刪掉  // 讓 Node 1 的 next 指到 Node 3 |

(5) 刪除頭項（第一筆資料）

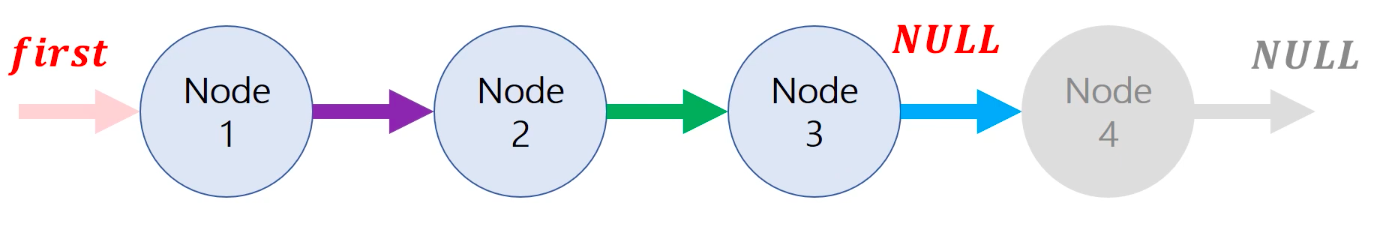


最簡單的方式是讓 first 指標改指到第二筆資料，這樣之後就會從第二筆資料開始讀，但是如果沒有先把 Node 1 的位置存起來，就直接把 first 刪掉，那會導致無法釋放掉 Node 1，產生記憶體洩漏。（也不能最開始就直接把 Node 1 釋放掉，因為這樣就找不到 Node 2 在哪裡）。

|  |  |
| --- | --- |
| 1. tmp = first  2. first = first.next  3. delete tmp | // 先把 first 指標（Node1）存在 first 裡  // 把 first 往右移動一格  // 把 Node 1 釋放掉 |

不可用剛剛的刪除函式！因為要改動 first 指標。

(6) 刪除尾項



|  |  |
| --- | --- |
| 1. Get Node\_3  2. delete Node\_3.next  3. Node\_3.next = 0 | // 取得最後一項 Node 4 的前一個，也就是 Node 3  // 把最後一項 Node 4 刪掉  // 把 Node 3 的 next 設定成空指標，因為 Node 3 成為新的鏈結串列結尾 |

可以用剛剛特定項的刪除函式，相當於該函式的 tmp 處代入 nullptr。

(7) Dummy head

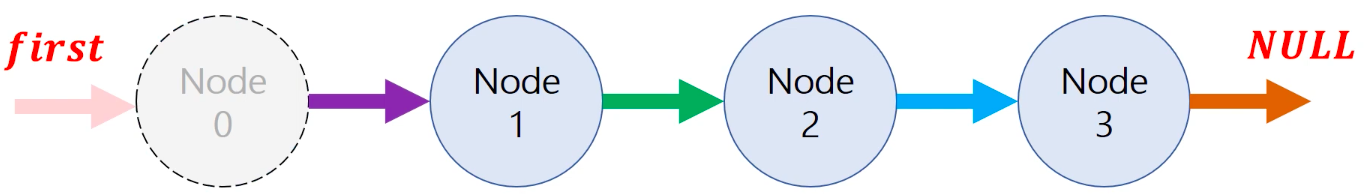
A. 尾端新增 / 刪除：「可以」套用通用的新增 / 刪除函式

B. 頭項新增 / 刪除：「不可」套用通用的新增 / 刪除函式

Dummy head：node 0

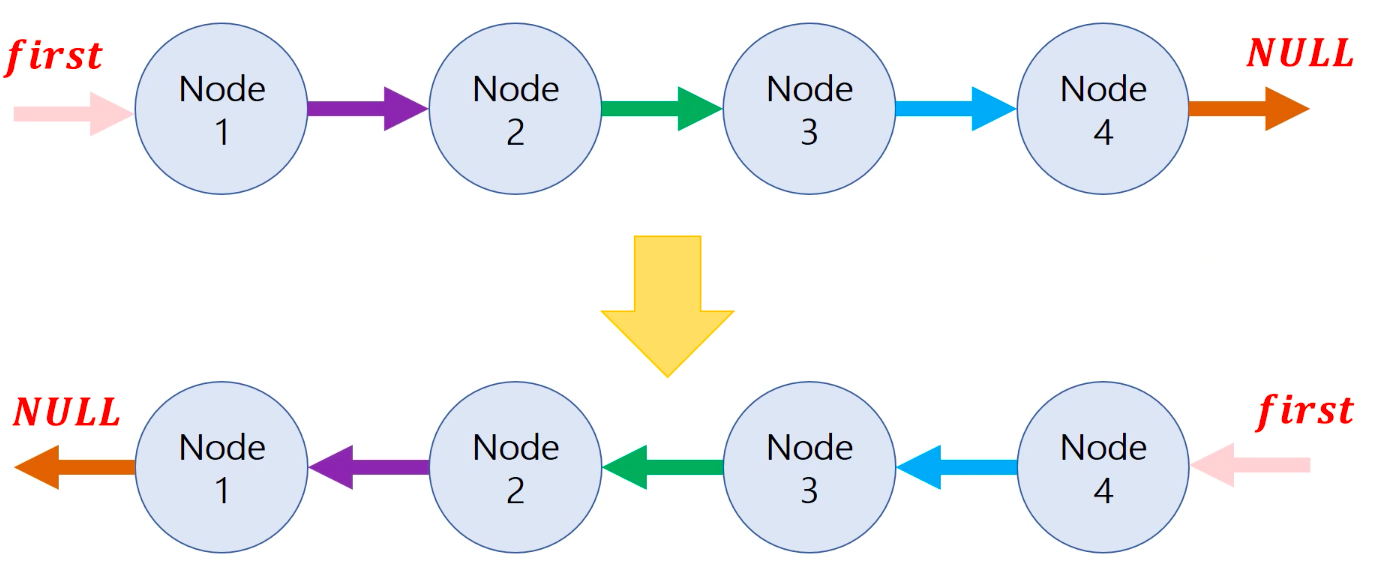
a. 在頭端新增一個空的 node

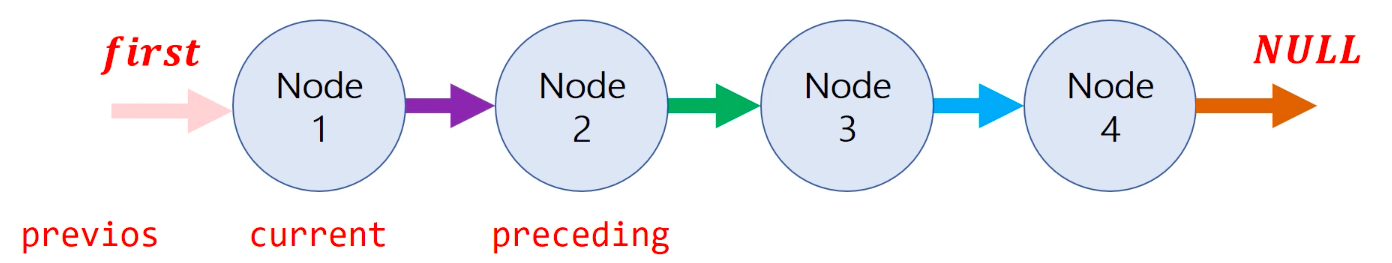
b. 新增後頭端的新增 / 刪除也可以套用通用的函式

****c. 第一筆資料在結構上變成不是第一個節點

(8) 反轉 Linked List

原本資料的順序如果是 1 2 3 4，反轉後會變成 4 3 2 1。

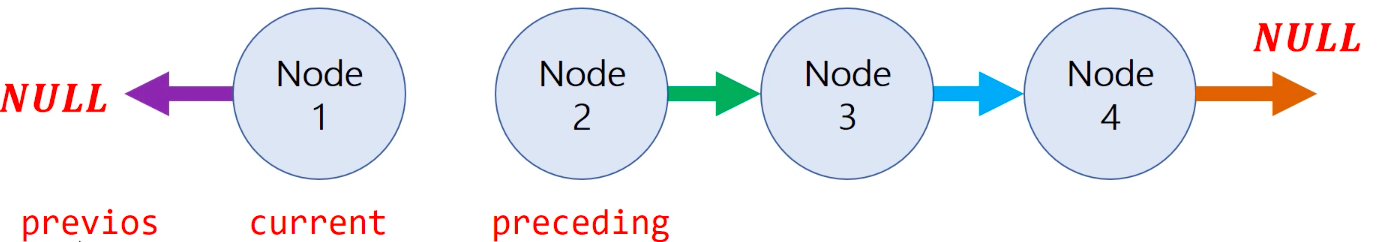


最直觀的做法似乎是把每個指標的方向都倒過來，比如原本如果是 Node 1 指到 Node 2，就改成 Node 2 指向 Node 1，但是一旦讓 Node 2 改指向 Node 1，就再也無法知道 Node 3 的位置了，因此必須把前後節點的位置先記錄起來。

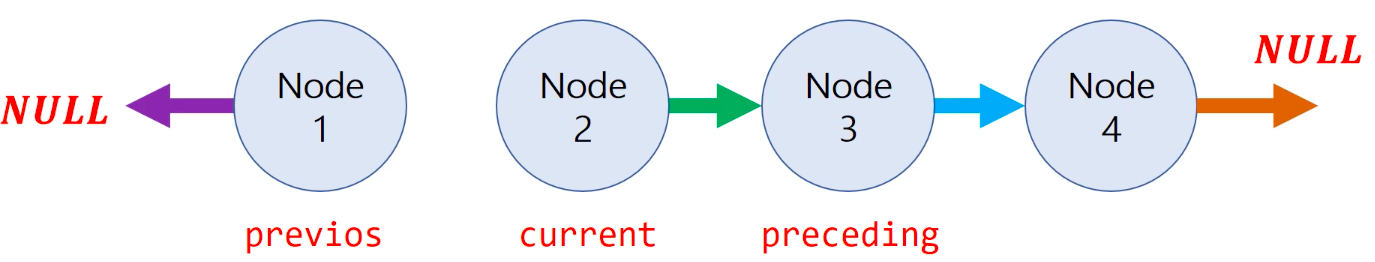
|  |  |
| --- | --- |
| 初始狀態 Initialization | |
| A. previous = nullptr  B. current = first  C. preceding = current -> next | // 現在要反轉的 node 的前一個（第一筆資料的前一筆是空指標）  // 現在要反轉的 node（從第一個節點開始反轉）  // 現在要反轉的 node 的後一個 |

設定好初始狀態後，接著執行下列步驟：

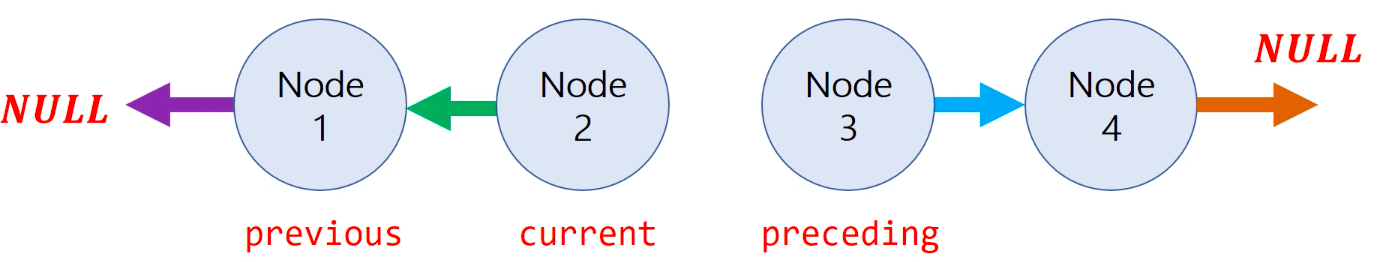
A. current -> next = previous：讓 current 的 next 改指向前方的空指標

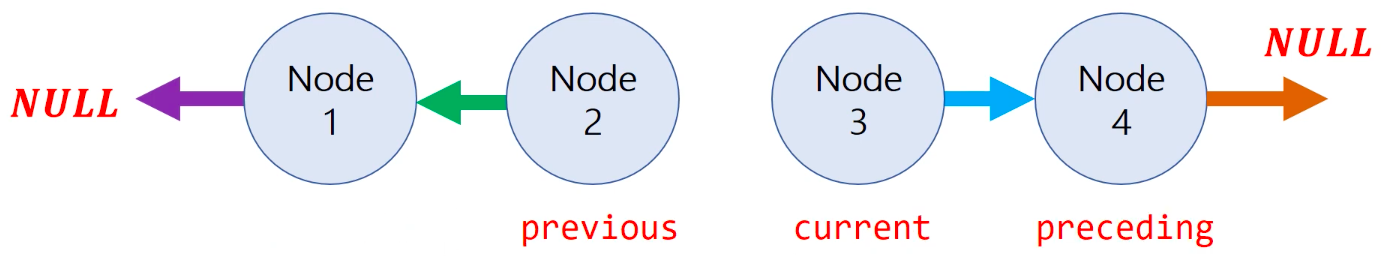


B to D. 把 previous、current、preceding 三個指標分別往後移一個節點

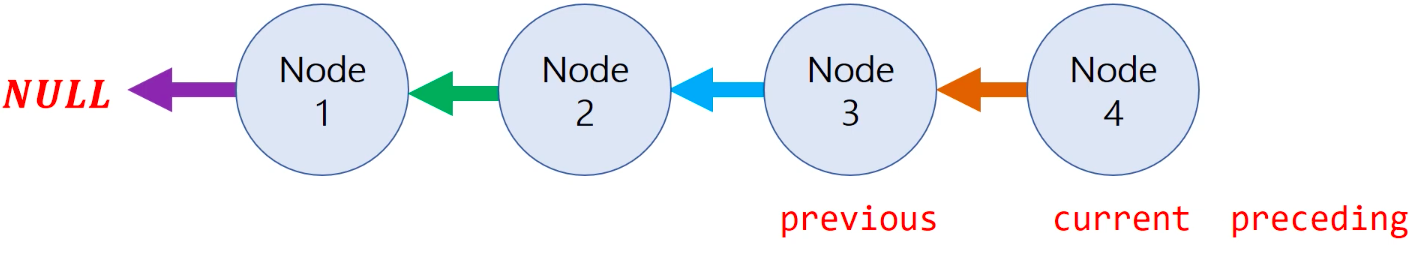


E. 對每個節點執行前面四個步驟（下方圖示是對 Node 2 的處理）

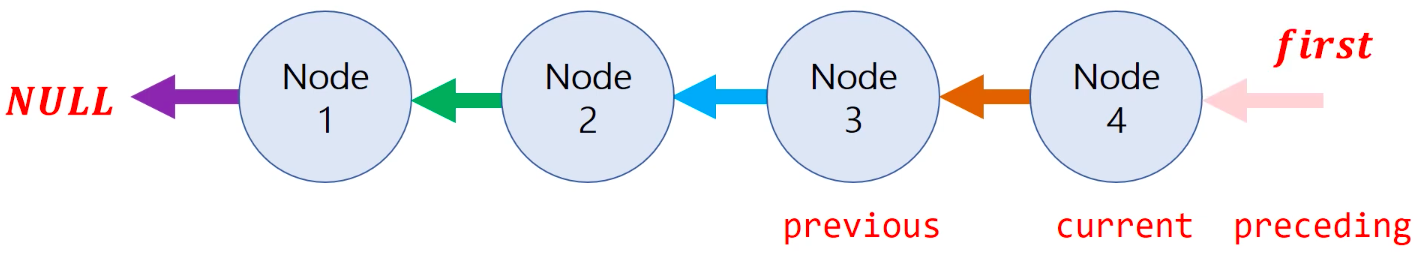




F. 到達最後一個節點時，同樣把 preceding 改指向前一個節點



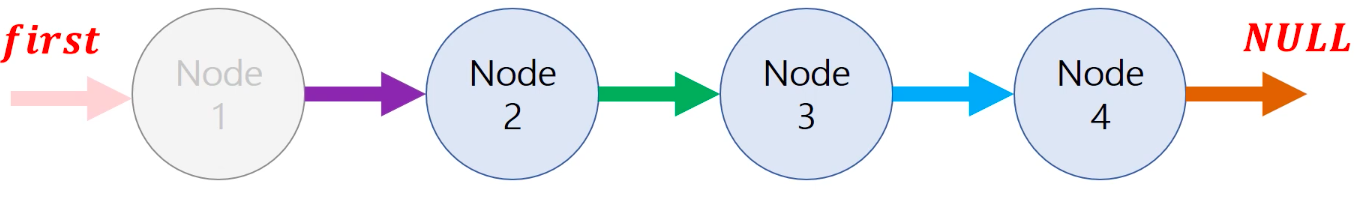
G. 最後，讓 first 指標指到原本的最後一個節點



|  |  |
| --- | --- |
| 反轉現在處理的 node 指的方向（從往後指向 preceding 變成往前指向 previous） | |
| A. current -> next = previous  B. previous = current  C. current = preceding  D. preceding = preceding -> next  E. repeat step A~D  (until preceding == nullptr)  F. current -> next = previous  G. first = current | // current 節點的 next 改指到前一個節點  // B~D: 把三個節點各往後移一格  // 對每個節點重複執行上方步驟  // 直到到達最後一個節點  // 再反轉一次 next 指標方向  // 讓 first 指向原本的最後一項（目前的 current） |

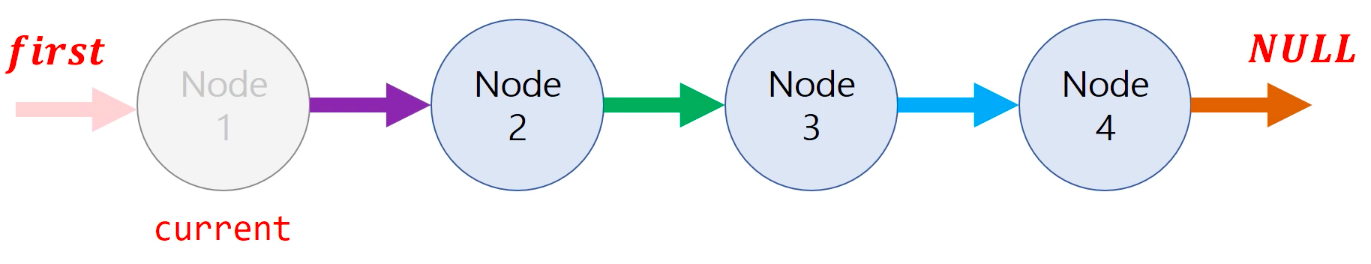
(9) 清空鏈結串列

要把所有資料清空，可以藉由不斷刪除頭項來達成。

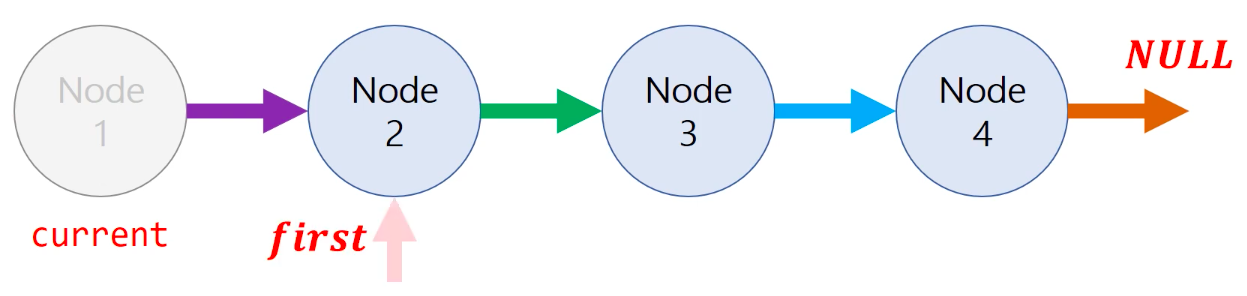


然而如果直接刪掉 Node 1，就不知道 Node 2 在哪裡了，因此每次先把 first 往後移一格記錄下一筆資料的位置，再把目前的節點釋放掉。

A. current 指向第一個節點 first

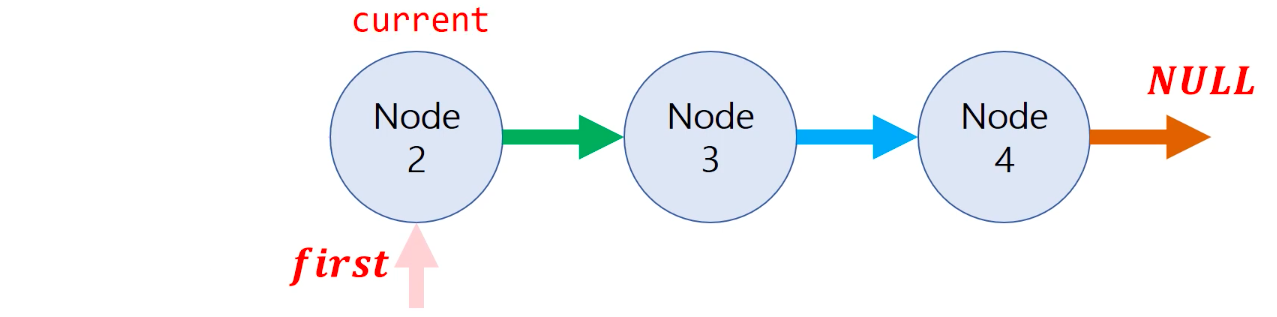


B. first 移向下一筆資料



C. 刪除 current 現在指到的節點

D. 重複 A 到 C 步驟，current 往下一筆資料繼續處理



|  |  |
| --- | --- |
| 清空鏈結串列 | |
| A. current = first  B. first = first->next  C. delete current  D. repeat step A~C | // 從第一筆資料開始刪除  // 讓 first 往後移動去紀錄下一筆資料的位置  // 紀錄後就可以放心把節點刪除  // 直到 first 指向空指標，代表串列已經清空 |

## 3. 實作鏈結串列的函式

(1) 新增頭項與尾項

在特定位置新增或刪除元素會用到迭代器，因此稍後再做，先寫下列兩個函式

A. 新增頭項：Push\_Front

B. 新增尾端元素：Push\_Back

這兩個函式都可以用來不斷新增鏈結串列的資料，新增完後，使用剛剛寫的 print 來印出串列中資料，看看輸出結果是否與預期相符。

|  |  |
| --- | --- |
| Push\_Front：新增頭項（第一筆資料） | |
| 基本邏輯 | |
|  | |
| 宣告 | |
| 1  2  3  4  5  6  7 | template <typename T>  private:  Node<T>\* First;  public:  ...  void Push\_Front(T);  }; |
| 實作 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | template <typename T>  void Linked\_List<T>::Push\_Front(T value){  // Step 1. 開出一個新的 node  Node<T>\* new\_node  // 例外處理：空 list  if (Head == nullptr){  new\_node = new Node<T>{value, 0};  }  // 一般情形  else {  // Step 2. 讓新的 node 的 next 指向第一筆資料的位置 first  new\_node = new Node<T>{value, First};  // 上面用大括號初始化的方法與下列三行的意義相同  // Node<T>\* new\_node = new Node<T>();  // new\_node->Data = value;  // new\_node->Next = First;  // Step 3. 讓 First 改指向新的開頭資料  First = new\_node;  } |

|  |  |
| --- | --- |
| Pop\_Front：刪除頭項（第一筆資料） | |
| 基本邏輯 | |
| 開始執行下面的三個步驟之前，應該先檢查鏈結串列是不是空的： | |
| 宣告 | |
| 1  2  3  4  5  6  7  8 | template <typename T>  private:  Node<T>\* First;  public:  ...  void Push\_Front(T);  void Pop\_Front();  }; |
| 實作 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | template <typename T>  void Linked\_List<T>::Pop\_Front(){  // 例外處理：鏈結串列是空的  if (First == nullptr){  // 不進行處理直接返回  return ;  }  // Step 1. 用 tmp 儲存原本的開頭位置 Node 1  Node<T>\*tmp = First;  // Step 2. 把 First 指向第二筆資料  First = First->Next;  // Step 3. 把要刪除的開頭節點釋放掉  delete tmp;  } |

|  |  |
| --- | --- |
| 測試 Push Front 與 Pop Front | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <iostream>  #include "Linked\_list.h"  using namespace std;  int main(){  Linked\_List<int> List;  // 3 2 1，每一項是從開頭新增  List.Push\_Front(1);  List.Push\_Front(2);  List.Push\_Front(3);  List.Print\_List();  // 把前兩筆資料刪掉，第一次刪除 3，第二次刪除 2  List.Pop\_Front();  List.Pop\_Front();  List.Print\_List();  // 再試著刪除兩筆資料  // 第一次會刪除 1，第二次則因為已經沒有資料所以不做處理  List.Pop\_Front();  List.Pop\_Front();  List.Print\_List();  return 0;  } |
| 執行結果 | |
| Data:3 2 1  Data:1  This list is empty! | |

(2) 新增以下函式

A. 新增尾端元素：Push\_Back

B. 刪除尾端元素：Pop\_Back

函式的取名與 STL 裡相同，只有大小寫的區別。

|  |  |
| --- | --- |
| Push\_Back：新增尾項（尾端的資料） | |
| 基本邏輯 | |
|  | |
| 宣告 | |
| 1  2  3  4  5  6  7  8  9 | template <typename T>  private:  Node<T>\* First;  public:  ...  void Push\_Front(T);  void Pop\_Front();  void Push\_Back(T);  }; |
| 實作 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | template <typename T>  void Linked\_List<T>::Push\_Back(T value){  // Linked list 是空的時候，Push\_Back 跟 Push\_Front 是一樣的  if (First == nullptr){  Push\_Front(value);  }  // 一般情形，先找到最後一筆資料，讓最後一筆資料指到新的節點  else {  // Step 1. 用 current 指標去找到最後一筆資料  Node<T>\* current = First;  while(current->Next != nullptr){  current = current-> Next;  }  // Step 2. 讓最後一筆資料的 Next 指到新的節點  // Step 3. 且新的節點資料是 value、next 指標是空指標  current->Next = new Node<T>{value, nullptr};  }  } |

|  |  |
| --- | --- |
| Pop\_Back：刪除尾項 | |
| 基本邏輯 | |
|  | |
| 宣告 | |
| 1  2  3  4  5  6  7  8  9  10 | template <typename T>  private:  Node<T>\* First;  public:  ...  void Push\_Front(T);  void Pop\_Front();  void Push\_Back(T);  void Pop\_Back();  }; |
| 實作 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | template <typename T>  void Linked\_List<T>::Pop\_Back(){  // 例外處理：鏈結串列是空的  if (First == nullptr){  // 直接返回  return ;  }  else {  Node<T>\* current = First;  // 例外處理：只有一筆資料，刪除完變成空鏈結串列  if(current->Next == nullptr){  delete First;  First = nullptr;  return ;  }  // 一般情況  // Step 1. 找到倒數第二筆資料  while(current->Next->Next != nullptr){  current = current-> Next;  }  // Step 2. 此時 current 是倒數第二筆資料  // 刪除 current->Next，即最後一筆資料  delete current->Next;  // Step 3. 將串列結尾設定為空指標  current->Next = nullptr;  }  } |

|  |  |
| --- | --- |
| 測試 Push Back 與 Pop Back | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include <iostream>  #include "Linked\_list.h"  using namespace std;  int main(){  Linked\_List<int> List;  // 1 2 3，每一項是從尾端新增  List.Push\_Back(1);  List.Push\_Back(2);  List.Push\_Back(3);  List.Print\_List();  // 把後兩筆資料刪掉，第一次刪除 3，第二次刪除 2  List.Pop\_Back();  List.Pop\_Back();  List.Print\_List();  // 再試著刪除兩筆資料  // 第一次會刪除 1，第二次則因為已經沒有資料所以不做處理  List.Pop\_Back();  List.Pop\_Back();  List.Print\_List();  return 0;  } |
| 執行結果 | |
| Data:1 2 3  Data:1  This list is empty! | |

(3) 新增以下函式

A. 清空所有資料：Clear

B. 反轉所有資料的方向：Reverse

注意例外處理，比如鏈結串列是空的，或者只有一筆資料等情形。

|  |  |
| --- | --- |
| Clear 和 Reverse 的宣告 | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | template <typename T>  private:  Node<T>\* First;  public:  ...  void Push\_Front(T);  void Pop\_Front();  void Push\_Back(T);  void Pop\_Back();  void Clear();  void Reverse();  }; |

|  |  |
| --- | --- |
| Clear：清空所有資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | template <typename T>  void Linked\_List<T>::Clear(){  // 宣告一個指標來儲存要釋放掉的節點  Node<T>\* current;  // 在鏈結串列還有資料時繼續處理  while(First != nullptr){  // 紀錄這次要釋放的節點  current = First;  // First 往右一格，不再指到第一筆資料  First = First->Next;  // 釋放掉節點 current  delete current;    }  } |

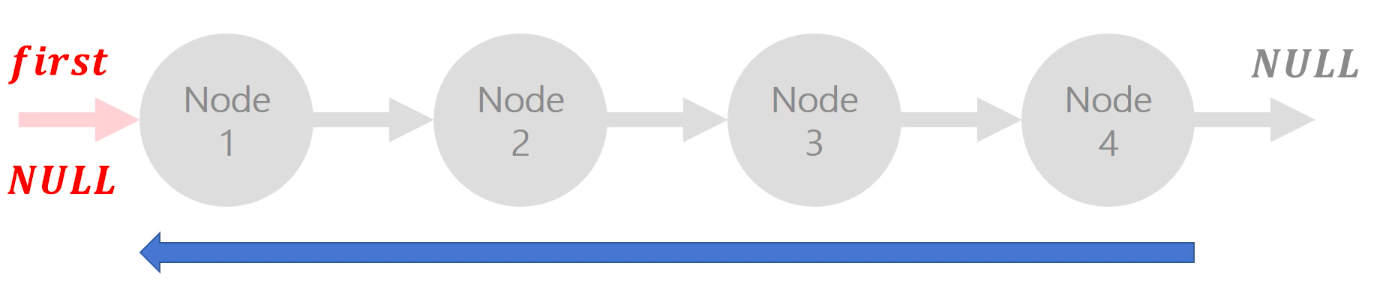
|  |  |
| --- | --- |
| Reverse：反轉所有資料的方向 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | template <typename T>  void Linked\_List<T>::Reverse(){  // 例外處理：空串列  if (First == nullptr)  return ;  // 例外處理：只有一筆資料，反轉後與原來相同，因此不需處理  if (First->Next == nullptr)  return ;  // 宣告三個指標紀錄目前處理的節點與前後節點  Node<T>\* previous = nullptr;  Node<T>\* current = First;  Node<T>\* preceding = First->Next;  // 在 current 到最後一個節點之前進行下列處理  while(preceding!=nullptr){  // 改變 current 的指標指向  current->Next = previous;  // 三個節點各往右一格  previous = current;  current = preceding;  preceding = preceding->Next;  }  // 處理尾端的節點  current->Next = previous;  // 把 First 指向尾端的 current  First = current;  } |

從上面的程式碼中，可以發現反轉鏈結串列的複雜度是 。

|  |  |
| --- | --- |
| 測試 Reverse 與 Clear | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #include <iostream>  #include "Linked\_list.h"  using namespace std;  int main(){  Linked\_List<int> List;  // 1 2 3  List.Push\_Back(1);  List.Push\_Back(2);  List.Push\_Back(3);  List.Print\_List();  // 反轉鏈結串列並印出  List.Reverse();  List.Print\_List();  // 清空鏈結串列資料  List.Clear();  List.Print\_List();  return 0;  } |
| 執行結果 | |
| Data:1 2 3  Data:3 2 1  This list is empty! | |

# 第三節：不同種 Linked List

接下來要介紹各種不同的鏈結串列。

剛才寫的鏈結串列是單向的，每個節點都只連到一個另一個節點，因此在做搜尋或尋訪的時候，只能從第一個節點、第二個節點、第三個節點、...一路往下走，沒有辦法回頭，這是因為每個節點都「只含有指向下一個節點的指標」，沒有辦法知道上一個節點在哪裡。

這會產生一些問題，比如想知道 Node 4 的上一筆資料 Node 3 在哪裡時，竟然只能重新從頭開始找，而不能從 Node 4 倒退回去。

只能往後移動的鏈結串列叫做「單向鏈結串列Single Linked List」。

## 1. 各種不同的鏈結串列

A. Single Linked List

a. 新增/刪除最後一個元素需要

b. 只有最後一個元素可以用 last 指標解決（事實上倒數第二筆資料的

next 要指向空指標，因此還是需要從頭開始找）

B. 雙向鏈結串列 Double Linked List

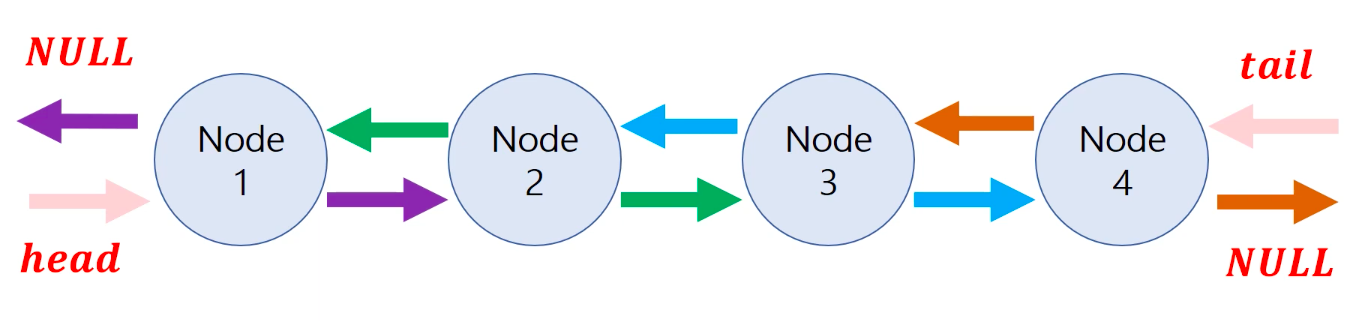
C. 環向鏈結串列 Circular Linked List

(1) 雙向鏈結串列

所有的節點中都含有兩個指標

A. previous 指向上個節點

B. next 指向下個節點

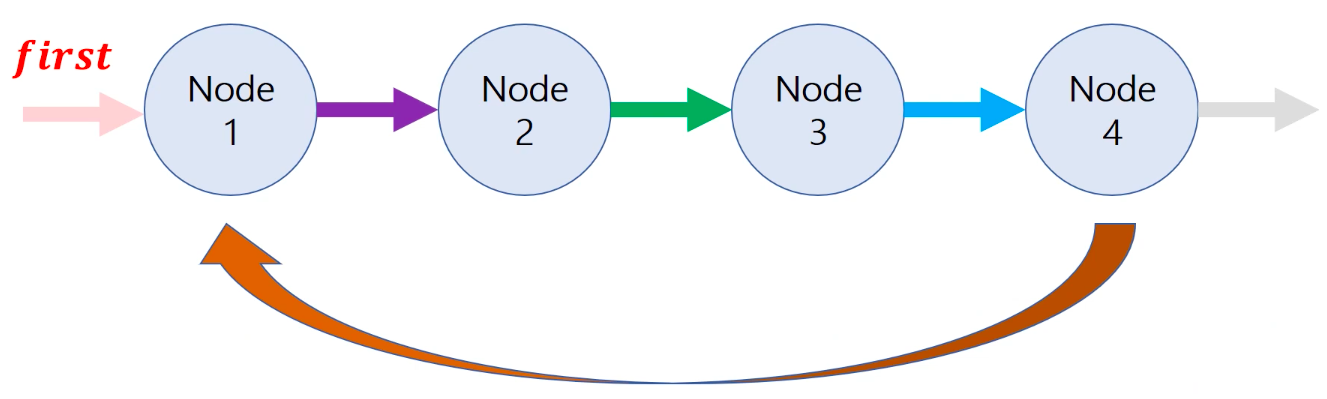


在雙向鏈結串列中，如果想知道 Node 2 的前一筆資料，往 previous 走就可以了。

另外，雙向鏈結串列中有兩個指標分別指向鏈結串列的最開頭（head）和最結尾（tail），這樣可以解決單向鏈結串列中「尾端刪除資料」前置作業需要 的問題。

(2) 環狀鏈結串列

「環狀鏈結串列」是將單向鏈結串列的「最後一個指標的 next」指向第一個節點，這樣的好處是可以用同樣的函式不斷處理資料，因為處理完尾端節點後，就會自動回到 Node 1。



另外，環狀鏈結串列最適合的操作是「旋轉」，比如想讓節點的順序從 1 2 3 4 變成 2 3 4 1，在環狀的鏈結串列中，只要讓 first 指標改指向 Node 2 就可以了，所以當要進行旋轉的操作時，環狀鏈結串列是很好的選擇。

在環狀鏈結串列中，要怎麼知道已經走完一圈了？當發現目前 node 的記憶體位置與 first 相同，就代表已經走完一圈，要停止執行，避免形成無窮迴圈。

(3) 鏈結串列與陣列的比較

|  |  |  |
| --- | --- | --- |
| 鏈結串列與陣列的比較 | | |
|  | 鏈結串列 | 陣列 |
| 記憶體需求  記憶體大小  元素的資料型態  插入/刪除複雜度  空間配置  存取方式  存取速度 | 無需連續的記憶體  須額外空間儲存指標  可不同  可隨時調整  只能依序存取  慢 | 需連續的記憶體  無須額外空間  相同  固定的大小  隨機(索引值)及依序存取  快 |

A. 新增與刪除資料的比較

在向量中，為了把所有資料存在連續的位置，在新增或刪除資料時都會非常不方便；鏈結串列則不要求資料放在連續的記憶體空間，因此新增與刪除資料較快，不用改動到其他資料。

對於鏈結串列來說，如果已經事先知道要插入或刪除的資料的記憶體位置，就只需要 ，而陣列因為需要移動其他資料，所以需要 。

B. 使用空間的比較

鏈結串列需要額外的空間儲存指標，所以空間比陣列來得大，整數的大小是 4 個 byte，整數指標也是 4 個 byte。因此在陣列中儲存一個整數只要一個 byte，鏈結串列則需要 8 個 byte，也就是說整數的情況下需要的空間是陣列的兩倍。

事實上，鏈結串列的每個節點存的資料型態也可以不一樣，比如一個 int 指到一個 float，陣列則因為每筆資料的空間大小固定，一般來說只能儲存一樣的資料型別。

C. 空間配置方式的比較

空間配置方面，鏈結串列可以隨時調整，要新增一筆資料時，在記憶體空間的任何一個位置開一個空間放新的資料就好；陣列則有固定的大小，即使是用動態陣列，也要到全新的地方去挖出更大的空間。

D. 存取資料的比較

存取方式方面，要在鏈結串列裡找到第 50 筆資料，就一定得從第一筆、第二筆、第三筆、...一路往下找；陣列則因為資料放在連續的空間，可以直接計算出第 50 筆資料存放的空間，而且計算只需要 。

存取速度方面，鏈結串列每到一個節點，就要把這個節點的 next 指標拿出來看，才能知道下一筆資料的記憶體位置在哪裡，會比陣列可以直接往下一個位置存取來的慢很多。這就像在 RPG 裡，要先問過每個關的關主才會知道下個關主在哪裡，又要問過下個關主才會知道下下個關主在哪裡。

總結來說，鏈結串列的優點是新增和刪除快，缺點是存取速度慢。

## 2. 把單向鏈結串列修改成雙向鏈結串列

A. node 內：prev 與 next

B. Linked\_List 類別內：head 與 tail

C. 在鏈結串列類別中新增迭代器，可直接由 vector 程式碼修改，並支援

a. 比較運算子：==、!=

b. ++、--

c. +、\*

D. 新增外部函式 find(container, target)

a. 回傳找到的第一個元素迭代器

b. 若無，回傳空指標

接下來要做的就是實際寫出雙向鏈結串列。在雙向鏈結串列的 node 內，會有一個 prev 指標指向上一個節點，一個 next 指標指向下一個節點；而在 Linked\_List 類別內，會有 head 指向第一筆資料，tail 指向最後一筆資料。

再來要新增迭代器，這裡的迭代器與向量中相似，需要支援一些常用的操作。

A. ++ 和 -- 代表移到上一筆資料或下一筆資料

B. + 是往後移多少筆資料，\* 是取值運算子。

再來，新增一個在 STL 裡面很常見的函式 find，因為它是外部函式，所以不在我們的容器類別裡面，它有兩個引數，第一個是要搜尋的容器 container，第二個則是要找的目標，找完後，會回傳第一個資料的迭代器，如果沒有找到，以鏈結串列為例會回傳空指標。

最後，新增以下兩個函式，利用迭代器指定要新增 / 刪除的位置

A. 在特定位置新增資料

B. 在特定位置刪除資料

剛剛因為還沒有寫出迭代器，因此沒有辦法在特定位置新增或刪除資料，在鏈結串列中給出索引值是沒有意義的，因為索引值只代表第幾筆資料，但是實際上還是得從第一筆資料一路往下找；相對的，如果給的是迭代器，因為透過迭代器知道記憶體位置，就可以直接進行新增或刪除。

(1) 修改 Linked\_List 類別

|  |  |
| --- | --- |
| struct Node 與 class Linked\_List | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | template <typename T>  struct Node{  T Data;  Node\* Prev;  Node\* Next;  }  template <typename T>  class Linked\_List{  private:  Node<T> \*First;  public:  class Iterator{  private:  // 注意! 修改成 Node<T>\* 資料類型  Node<T>\* iter;  public:  // 注意! 初始化資料類型是 Node<T>\*  Iterator(Node<T>\* = nullptr);  void operator++();  void operator--();  void operator++(int);  void operator--(int);  void operator==(Iterator);  void operator!=(Iterator);  bool operator=(Iterator);  Iterator operator+(int);  Iterator operator-(int);  T& operator\*();  };  ...  } |

(2) 迭代器類別中函式的定義

|  |  |
| --- | --- |
| Iterator 的建構式 | |
| 1  2  3  4  5 | template <typename T>  Linked\_List<T>::Iterator::Iterator(Node\* pointer){  // 指定指標 pointer 作為 iter 的初始值  iter = pointer;  } |

|  |  |
| --- | --- |
| Iterator::++operator | |
| 1  2  3  4  5  6  7 | template<typename T>  void Linked\_List<T>::Iterator::operator++(){  // 讓 iter 往後一筆資料（改指向 iter->Next）  if(iter != nullptr){  iter = iter->Next;  }  } |

|  |  |
| --- | --- |
| Iterator::--operator | |
| 1  2  3  4  5  6  7 | template<typename T>  void Linked\_List<T>::Iterator::operator--(){  // 讓 iter 往前一筆資料（改指向 iter->Prev）  if(iter != nullptr){  iter = iter->Prev;  }  } |

|  |  |
| --- | --- |
| Iterator::operator++ | |
| 1  2  3  4  5  6  7 | template<typename T>  void Linked\_List<T>::Iterator::operator++(int){  // 讓 iter 往後一筆資料（改指向 iter->Next）  if(iter != nullptr){  iter = iter->Next;  }  } |

|  |  |
| --- | --- |
| Iterator::operator-- | |
| 1  2  3  4  5  6  7 | template<typename T>  void Linked\_List<T>::Iterator::operator--(int){  // 讓 iter 往前一筆資料（改指向 iter->Prev）  if(iter != nullptr){  iter = iter->Prev;  }  } |

|  |  |
| --- | --- |
| Iterator::operator== | |
| 1  2  3  4  5 | template<typename T>  bool Linked\_List<T>::Iterator::operator==(Iterator iter2){  // 與 vector 中相同  return iter == iter2.iter;  } |

|  |  |
| --- | --- |
| Iterator::operator!= | |
| 1  2  3  4  5 | template<typename T>  bool Linked\_List<T>::Iterator::operator!=(Iterator iter2){  // 與 vector 中相同  return iter != iter2.iter;  } |

|  |  |
| --- | --- |
| Iterator::operator= | |
| 1  2  3  4  5 | template<typename T>  void Linked\_List<T>::Iterator::operator=(Iterator iter2){  // 與 vector 中相同  iter = iter2.iter;  } |

|  |  |
| --- | --- |
| Iterator::operator\* | |
| 1  2  3  4  5 | template<typename T>  void Linked\_List<T>::Iterator::operator\*(){  // 取出 iter 下的 Data 變數  return iter->Data;  } |

|  |  |
| --- | --- |
| Iterator::operator+ | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | template <typename T>  void typename Linked\_List<T>::Iterator Linked\_List<T>::Iterator::operator+(int offset){  Iterator result(iter);  for (int i=0;i<offset;i++){  // 例外處理：result 已經移動到尾段後的空指標  if(result->iter == nullptr)  return result;  // 一般情況  // result 下的節點變成 result 下的節點指到的下一個節點  result->iter = result->iter->Next;  }  return result;  } |

|  |  |
| --- | --- |
| Iterator::operator- | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | template <typename T>  void typename Linked\_List<T>::Iterator Linked\_List<T>::Iterator::operator-(int offset){  Iterator result(iter);  for(int i=0;i<offset;i++){    // 例外處理：result 已經移動到第一筆資料前的空指標  if(result->iter == nullptr)  return result;  // 一般情況  // result 下的節點變成 result 下的節點指到的前一個節點  result->iter = result->iter->Prev;  }  return result;  } |

(3) 修改 Linked\_List 中的函式寫法，改為「雙向鏈結串列」

|  |  |
| --- | --- |
| class Linked\_list | |
| 1  2  3  4  5  6  7  8  9  10 | template <typename T>  class Linked\_List{  private:  // 指標成員修改成 Head 跟 Tail  Node<T>\* Head;  Node<T>\* Tail;  public:  class Iterator{...}  ...  } |

|  |  |
| --- | --- |
| Double 版 Reverse | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | template <typename T>  void Linked\_List<T>::Reverse(){  // 例外處理：空 List  if(Head == nullptr)  return ;  // 例外處理：只有一筆資料  // Head 不是 nullptr 的情況下還與 Tail 相同，代表指到同一節點  if(Head == Tail)  return ;  // First 改成 Head  Node<T>\* previous = nullptr;  Node<T>\* current = Head;  Node<T>\* preceding = Head->Next;  // 開始反轉前，把 Tail 指向第一筆資料  Tail = Head;    while (preceding != nullptr){  current->Next = previous;  // current 的 Prev 也要反轉成指向後一筆資料  current->Prev = preceding;  previous = current;  current = preceding;  preceding = preceding->Next;  }  current->Next = previous;  current->Prev = preceding;  // Head 改指向最後一筆資料  // 也可以把 Head 跟 Tail 做交換 swap  Head = current;  } |

|  |  |
| --- | --- |
| Double 版 Clear | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template <typename T>  void Linked\_List<T>::Clear(){  Node<T>\* current;  // 之前程式碼中的First 都要改成 Head  while(Head!=nullptr){  current = Head;  Head = Head->Next;  delete current;  }  // Tail 也要設定成空指標  Tail = nullptr;  } |

|  |  |
| --- | --- |
| Double 版 Push\_Back | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | template <typename T>  void Linked\_List<T>::Push\_Back(T value){  // 例外情形：空 list  if (Head == nullptr){  Node<T>\* new\_node = new Node<T>{value,nullptr,nullptr};  Head = new\_node;  Tail = new\_node;  }  // 一般情形：直接在最後一筆資料 tail 後面加上新的 Node  else {  //{Data, Prev, Next}  Tail->Next = new Node<T>{value, Tail, nullptr};  Tail = Tail->Next;  }  } |

|  |  |
| --- | --- |
| Double 版 Pop\_Back | |
| A. 空 List 不處理  B. 如果只有一筆資料，要改變 Head  C. 一般情形，當有兩筆資料以上，只需改變 Tail | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | template <typename T>  void Linked\_List<T>::Pop\_Back(){  // 例外處理：空 List  if (Head == nullptr)  return ;  // 例外處理：只有一筆資料  if (Head == Tail){  delete Tail;  Head = Tail = nullptr;  // 結束處理  return ;  }  // 一般情形  // Tail 改指向倒數第二筆資料  Tail = Tail->Prev;  // 把最後一筆資料刪掉  delete Tail->Next;  // 把倒數第二筆資料(現在的 tail) 的 Next 指向空指標  Tail->Next = nullptr;  } |

|  |  |
| --- | --- |
| Double 版 Push\_Front | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | template <typename T>  void Linked\_List<T>::Push\_Front(T value){  if (Head == nullptr){  Node<T>\* new\_node = new Node<T>{value, nullptr, nullptr};  Head = new\_node;  Tail = new\_node;  }  // 一般情形  else {  Head->Prev = new Node<T>{value, nullptr, Head};  Head = Head->Prev;  }  } |

|  |  |
| --- | --- |
| Double 版 Pop\_Front | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | template <typename T>  void Linked\_List<T>::Pop\_Front(){  // 例外處理：空 list  if(Head == nullptr){  return ;  }  // 例外處理：只有一筆資料  if(Head == Tail){  delete Head;  Head = Tail = nullptr;  return ;  }  // 一般情況  Head = Head->Next;  delete Head->Prev;  Head->Prev = nullptr;  } |

|  |  |
| --- | --- |
| Double 版建構式 | |
| 1  2  3  4  5 | template <typename T>  Linked\_List<T>::Linked\_List(){  Head = nullptr;  Tail = nullptr;  } |

至於 Print\_List 和 Search\_List 兩個函式，只要把裡面的 First 都改成 Head 即可。

到這邊為止，Linked List 類別已經成功改成了「雙向鏈結串列」，且其中加入了迭代器，讀者可以自行使用前幾節中的程式碼測試。

(4) 外部函式 Find

|  |
| --- |
| 宣告 |
| 為了要讓 Find 可以取用 Head 和 Tail，要把函式宣告成 Linked\_List 類別的 friend |
| template <typename T>  class Linked\_List{  private: ...  public:  ...  // 因為用到迭代器，所以要放在迭代器的宣告後面  // 注意 friend 關鍵字要放在 template 後面  // T 要另外取名成 T2 避免與 Linked\_List 的 template T 衝突  template<typename T2> friend typename Linked\_List<T2>::Iterator  Find(Linked\_List<T2>,T2);  } |

|  |  |
| --- | --- |
| 外部函式 Find 的實作 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | template<typename T>  typename Linked\_List<T>::Iterator Find(Linked\_List<T> List, T data){  // 宣告一個指向空指標的迭代器  // 因為在外部，要寫明 Iterator 是在 Linked\_List 類別下  typename Linked\_List<T>::Iterator null\_iter(nullptr);  // 例外處理：空 list  if (List.Head == nullptr){  return null\_iter;  }  // 宣告迭代器指向 list 的開頭  // 多使用到 T 時，前方要加上 typename 關鍵字  typename Linked\_List<T>::Iterator current(ListHead);  while (current != null\_iter){  if (\*current == data)  return current;  current++;  }  // 沒有找到時同樣回傳 current，此時 current 中的 iter 是空指標  return current;  } |

|  |  |
| --- | --- |
| 測試 Find | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | int main(){  Linked\_List<int> List;    List.Push\_Back(1);  List.Push\_Back(2);  List.Push\_Back(3);  List.Push\_Back(4);  List.Push\_Back(5);  List.Push\_Back(6);  List.Push\_Back(7);  List.Push\_Back(8);  Linked\_List<int>::Iterator iter1 = Find(List, 3); // iter1 在 3 的位置  Linked\_List<int>::Iterator iter2 = Find(List, 7); // iter2 在 7 的位置  // 從 iter1 輸出到 iter2，注意「前閉後開」的適用  for(;iter1!=iter2;iter1++){  cout << \*iter1 << " ";  }  } |
| 執行結果 | |
| 3 4 5 6 | |

(5) Insert 和 Erase

|  |  |
| --- | --- |
| Insert 和 Erase 的宣告 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | template <typename T>  class Linked\_List{  private: ...  public:  // 因為新增的函式要取用到 iter  // 把 Linked List 設為迭代器的 friend  class Iterator {  friend Linked\_List;  ...  }  // 在特定迭代器的前面插入資料  void Insert(Iterator, const T&);  // 把特定迭代器指到的資料刪除  void Erase(Iterator);  template<typename T2> friend typename Linked\_List<T2>::Iterator  Find(Linked\_List<T2>,T2);  } |

上面的宣告中，注意 Insert 傳入的引數型態使用 const T& value。

const T& 型態基本上是一個 T 型態的變數，因為傳參考呼叫效率較佳，因此先在後方加 &，變成 T&，然而，我們又不希望因為傳參考，使得該資料的值被函式改動，因此再將其設為常數 const，成為「const T&」。

|  |  |
| --- | --- |
| Insert | |
| A. it 是迭代器  B. it.iter 是迭代器指到的 node  C. it.iter 下面才有 Data、Prev 和 Next  D. STL 裡提供的 Insert 函式是將資料插入到在使用者提供的迭代器的位置，  也就是該筆資料會出現在迭代器的前面  a. 原先資料為 A B，使用者傳入指向 B 的迭代器  b. 插入後結果為 A C B | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | template <typename T>  void Linked\_List<T>::Insert(Iterator it, T value){  // TODO：這個版本還不能拿來新增頭項或尾項  // 因為需要修改 head 和 tail  // 等到寫完 begin 和 end 函式後，會再回來修改  // A B，要把 C 插在 B 前  // A C B，迭代器 it 要指到 B  Node<T> \*new\_node = new Node<T>{value, it.iter->Prev, it.iter}  // A->Next = C  it.iter->Prev->Next = new\_node;  // B->Prev = C  it.iter->Prev = new\_node;  } |

|  |  |
| --- | --- |
| Erase | |
| A. 先讓 A->Next 指向 C  B. C->Prev 指向 A | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | template <typename T>  void Linked\_List<T>::Erase(Iterator it){  // A B C, B provided by user = it  // A C  // 不能拿來刪除頭項或尾項    // it.iter->Prev 是 A  // it.iter 是 B  // it.iter->Next 是 C  it.iter->Prev->Next = it.iter->Next;  it.iter->Next->Prev = it.iter->Prev;  // 刪除 it  delete it.iter;  } |

|  |  |
| --- | --- |
| 測試 Insert 與 Erase | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | int main(){  Linked\_List<int> List;    List.Push\_Back(1);  List.Push\_Back(2);  List.Push\_Back(3);  List.Push\_Back(4);  List.Push\_Back(5);  List.Push\_Back(6);  List.Push\_Back(7);  List.Push\_Back(8);  Linked\_List<int>::Iterator iter1 = Find(List, 3);  Linked\_List<int>::Iterator iter2 = Find(List, 7);  // 在 4 這筆資料前面插入 100  List.Insert(Find(List, 4),100);  // 印出所有資料  for(;iter1!=iter2;iter1++){  cout << \*iter1 << " ";  }  cout << endl;  // 先找到 4 這筆資料的位置後，用 Erase 刪掉  List.Erase(Find(List, 4));  iter1 = Find(List, 3);  // 印出所有資料  for(;iter1!=iter2;iter1++){  cout << \*iter1 << " ";  }  } |
| 執行結果 | |
| 3 100 4 5 6  3 100 5 6 | |

(6) 加入以下「外部」函式操作迭代器，並修改之前的函式

A. Advance(it,n)：迭代器 it 前後移動 n 步

B. Distance(it1,it2)：計算兩迭代器間的距離

C. Begin()：第一個元素的迭代器

D. End()：最後一個元素下一個位置的迭代器，在 Linked List 是空指標

E. Prev(it)：迭代器 it 前面的迭代器

F. Next(it)：迭代器 it 後面的迭代器

Advance 函式在 STL 中是用函式模板撰寫的，因為我們希望它能夠處理 Vector、Linked List 等各種類型的容器。

這些函式都在外部，平時操作迭代器時，無論要操作的容器種類為何，很多時候都是使用這些外部寫好的 ++、-- 等函式或運算子重載。

為了保持版面整潔，另開一個標頭檔 Function.h，並且在 Linked\_List.h（不是 main.cpp！）裡加上 include "Function.h"，這樣使用者在main.cpp 中引入 Linked\_List.h 即可，不用考慮要不要另外引入Function.h（引入Linked\_List.h 時會連帶被自動引入）。

|  |  |
| --- | --- |
| Function.h | |
| 1  2  3  4  5  6 | #ifndef FUNCTION\_H\_INCLUDED  #define FUNCTION\_H\_INCLUDED  … // 程式碼寫在這裡  #endif // FUNCTION\_H\_INCLUDED |

|  |  |
| --- | --- |
| Advance | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | template<typename T>  void Advance(T& iter, const int& distance){  // distance 是正數時，iter 往後移動  if (distance > 0){  for (int i=0;i<distance;i++){  iter++;  }  // 也可以寫成 iter = iter+distance;  }  // distance 是負數時，iter 往前移動  else {  for(int i=0;i>distance;i--){  iter--;  }  }  } |

|  |  |
| --- | --- |
| Distance | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template<typename T>  void Distance(T iter\_1, T iter\_2){  // 宣告 int result 儲存結果  int result = 0;  // 讓 iter\_1 一直往後移動，直到與 iter\_2 相等  while (iter\_1 != iter\_2){  iter\_1++;  result += 1;  }  return result;  } |

使用 auto 與 decltype 推導出回傳資料型別

A. auto：C++11 後引入，自動推定資料型別

a. auto declarator = initializer;

b. 在「auto i = 0;」中，因為 0 是 int，auto 會決定 i 是 int 型別

B. decltype：根據給定的 expression 決定型別

a. int i = 0; decltype(i) j;

b. 上面的程式碼中，j 會是從 i 抓出的資料型別，即 int

|  |  |
| --- | --- |
| 自動判斷回傳型態的函式宣告方式 | |
| 1  2  3  4  5 | template <typename Container, typename Index>  auto DoSomething(Container& c, Index i) -> decltype(c[i])  {  return c[i];  } |

如果我們希望在 C++ 中可以自動推斷出回傳的資料型別，可以像上面的程式碼一樣，在「函式名稱前面加上 auto」、「引數列後面加上 decltype」，這裡的 auto 不是用來自動判斷，它指的是交由後置的 decltype 來抓出資料型別，然後把抓出的型別再放到 auto 的位置。

|  |  |
| --- | --- |
| Begin | |
| 1  2  3  4  5  6  7 | template<typename T>  auto Begin(T container) -> decltype(container.Begin())  {  // 先假設 container 中有 Begin 函式，利用 container 中的 Begin  // 因為不同類型 container 取得開頭值的方法不同  return container.Begin();  } |

|  |  |
| --- | --- |
| End | |
| 1  2  3  4  5 | template<typename T>  auto End(T container) -> decltype(container.End()){  // 假設 container 中有 End 函式，利用 container 中的 End  return container.End();  } |

|  |  |
| --- | --- |
| Prev | |
| 1  2  3  4  5 | template<typename T>  T Prev(T iter){  iter--;  return iter;  } |

|  |  |
| --- | --- |
| Next | |
| 1  2  3  4  5 | template<typename T>  T Next(T iter){  iter++;  return iter;  } |

回到 Linked\_List.h 中加上 Begin 與 End 函式。

|  |  |
| --- | --- |
| class Linked\_List | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | template <typename T>  class Linked\_List{  private:  ...  public:  ...  class Iterator{...}  ...  Iterator Begin();  Iterator End();  Iterator Head\_Iter();  Iterator Tail\_Iter();  template<typename T2> friend typename Linked\_List<T2>::Iterator  Find(...)  } |

注意「前閉後開」，Begin 指到的是第一筆資料 Head，End 指到的是容器外的空指標。

|  |  |
| --- | --- |
| Linked\_List::Begin | |
| 1  2  3  4 | template <typename T>  typename Linked\_List<T>::Iterator Linked\_List<T>::Begin(){  return Iterator(Head);  } |

|  |  |
| --- | --- |
| Linked\_List::End | |
| 1  2  3  4 | template <typename T>  typename Linked\_List<T>::Iterator Linked\_List<T>::End(){  return Iterator(nullptr); // 鏈結串列的 End 是空指標  } |

|  |  |
| --- | --- |
| Linked\_List::Head\_Iter | |
| 1  2  3  4 | template <typename T>  typename Linked\_List<T>::Iterator Linked\_List<T>::Head\_Iter(){  return Iterator(Head); // 注意回傳型態是一個迭代器 Iterator  } |

|  |  |
| --- | --- |
| Linked\_List::Tail\_Iter | |
| 1  2  3  4 | template <typename T>  typename Linked\_List<T>::Iterator Linked\_List<T>::Tail\_Iter(){  return Iterator(Tail); // 注意回傳型態是一個迭代器 Iterator  } |

|  |  |
| --- | --- |
| 測試 Begin 與 End | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | int main(){  Linked\_List<int> List;    List.Push\_Back(1);  List.Push\_Back(2);  List.Push\_Back(3);  List.Push\_Back(4);  List.Push\_Back(5);  List.Push\_Back(6);  List.Push\_Back(7);  List.Push\_Back(8);  // 用 Linked\_List 下的 Begin 和 End 方法印出資料  for(auto iter=List.Begin();iter!=List.End();iter++){  cout << \*iter << " ";  }  cout << endl;  // 用外部函式 Begin 和 End 印出資料  for(auto iter=Begin(List);iter!=End(List);iter++){  cout << \*iter << " ";  }  cout << endl;  // 宣告迭代器 iter 指向 1  auto iter = Begin(List);  // iter 往後三筆資料，指向 4  Advance(iter,3);  cout << \*iter << endl;  // iter 往後負二筆資料（也就是往前兩筆資料），指向 2  Advance(iter,-2);  cout << \*iter << endl;  // iter\_1 指向 5，iter\_2 指向 3  auto iter\_1 = Find(List, 5);  auto iter\_2 = Find(List, 8);  cout << Distance(iter\_1, iter\_2);    } |
| 執行結果 | |
| 1 2 3 4 5 6 7 8  1 2 3 4 5 6 7 8  4  2  3 | |

(7) 回頭修改 Insert、Erase 函式

|  |  |
| --- | --- |
| Begin+End 版 Insert | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | template<typename T>  void Linked\_List<T>::Insert(Iterator it, const T& value){  // 插入在開頭  if (it == Begin()){  Push\_Front(value);  return ;  }  // 插入在尾端  it (it == End()){  Push\_Back(value);  return ;  }  ... // 其餘程式碼  } |

|  |  |
| --- | --- |
| Begin+End 版 Erase | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | template<typename T>  void Linked\_list<T>::Erase(Iterator it){  // 刪除頭項  if (it == Begin()){  Pop\_Front();  return ;  }  // 刪除尾項  // it 是 End() 空指標，或者是尾項時，都執行刪除尾項  if (it == End() || it == Iterator(Tail)){  Pop\_Back();  return ;  }  ... // 其餘程式碼  } |

|  |  |
| --- | --- |
| 加上 Remove 函式 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | template<typename T>  void Linked\_List<T>::Remove(T value){  // 利用迭代器 iter 遍歷鏈結串列  for (auto iter = Begin();iter!=End();iter++){  // 如果目前處理的節點資料與 value 相同  if (\*iter == value){  // 要刪除的是第一筆資料的話，使用 Pop\_Front()  if (iter == Begin()){  Pop\_Front();  // 一般情況  } else {  // 用 tmp 儲存刪除前的位置的「上一個節點」  // 以使 iter 被刪除後可以退回上一個節點  // 比如資料為 A B C，要刪除 B  // iter 刪除完需要退回 A，才能在下一輪往後檢查 C  auto tmp = iter;  tmp--;  // 利用 Erase 函式刪除 iter 所在的節點  Erase(iter);  // iter 退回 tmp 紀錄的上一個節點(A)  iter = tmp;  }  }  } |

|  |  |
| --- | --- |
| 測試 Remove | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | int main(){  Linked\_List<int> List;  List.Push\_Back(2);  List.Push\_Back(2);  List.Push\_Back(3);  List.Push\_Back(4);  List.Push\_Back(2);  List.Push\_Back(2);  List.Push\_Back(7);  List.Push\_Back(2);  for(auto iter=List.Begin();iter!=List.End();iter++){  cout << \*iter << " ";  }  cout << endl;  // 刪除所有 2  List.Remove(2);  for (auto iter=List.Begin();iter!=List.End();iter++){  cout << \*iter << " ";  }  } |
| 執行結果 | |
| 2 2 3 4 2 2 7 2  3 4 7 | |

(8) 修改外部函式 Find

剛剛寫 Find 時，因為還沒有 Begin 與 End 可以使用，需要直接取用 List.Head 和 List.Tail，所以把 Find 設定成 Linked\_List 下的 friend，現在可以修改成使用 Begin 和 End 函式後，拿掉 friend 的設定。

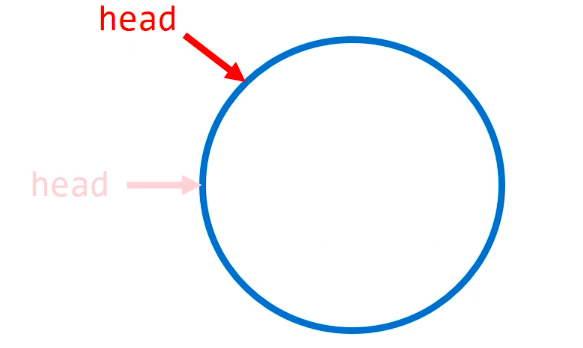
|  |  |
| --- | --- |
| Begin+End 版 Find | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | template<typename T>  typename Linked\_List<T>::Iterator Find(Linked\_List<T> List, T data){  typename Linked\_List<T>::Iterator null\_iter(nullptr);  if (List.Begin() == null\_iter){  return null\_iter;  }  auto current = List.Begin();  while (current != null\_iter){  if (\*current == data)  return current;  current++;  }  // Data Not Found in List  return current;  } |

## 3. 環狀鏈結串列 Circular Linked\_List

試著把鏈結串列改成 Circular Linked List（讓最後一筆資料指向第一筆資料），並加入兩函式

A. Remove：刪除特定資料

B. Rotate：旋轉內部資料，只需改變 head 指到的資料，往左或右轉皆可



範例：ABCDE -> （Rotate1） -> BCDEA （head 往後移一筆資料）

範例：ABCDE -> （Rotate2） -> CDEAB （head 往後移兩筆資料）

(1) 環狀鏈結串列的宣告

為了保留剛才寫好的雙向鏈結串列，可以將程式碼複製起來在另外一個檔案中修改。

|  |  |
| --- | --- |
| Circular\_Linked\_List.h | |
| 1  2  3  4  5  6  7  8  9  10 | #ifndef CIRCULAR\_LINKED\_LIST\_H\_INCLUDED  #defin CIRCULAR\_LINKED\_LIST\_H\_INCLUDED  #include <iostream>  #include "Function.h"  using namespace std;  … // 程式碼寫在這裡  #endif |

要測試時，記得在 main.cpp 當中引入：#include "Circular\_Linked\_List.h"。

|  |  |
| --- | --- |
| 修改 Linked\_List 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | template <typename T>  class Linked\_List {  private:  Node<T>\* Head;  // Node<T>\* Tail;  public:  class Iterator {  friend Linked\_List;  private: ...  public:  ...  /\*  Iterator operator+(int);  Iterator-(int);  \*/  T& operator\*();  };  /\*  ...  \*/  void Push\_Back(T);  void Rotate(int);  void Begin();  /\*  ...  \*/  } |

因為環狀串列沒有 Tail，要把「所有用到 Tail 的函式」拿掉或修改，如註解掉 Tail\_Iter、End 等等。

環狀鏈結串列中，所有需要遍歷的函式都不是檢查迭代器「是否到達空指標」，而是看是否又到達 Head。

因為修改所有函式為環狀串列版本會稍嫌冗長，除了「Rotate 函式」會用到的函式外，先全部註解起來：除 Rotate 外，只保留 Push\_Back。

(2) 修改 Push\_Back

|  |  |
| --- | --- |
| Circular 版 Push\_Back | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | template <typename T>  void Linked\_List<T>::Push\_Back(T value){  // 例外處理：空 list  if (Head == nullptr){  Head = new Node<T>(value, nullptr, nullptr);  // 環狀鏈結串列中，Head 的 Prev 和 Next 都指向 Head 本身  // 因為要指向 Head 本身，所以不能直接在上一行中指定  Head->Next = Head->Prev = Head;  }  // 一般情形  else {    // Head：A  // Head->Prev：Z  // Head->Prev->Next = Head = A  // … -> Z -> A -> …  // … -> Z -> Data -> A -> …  Node<T> \*new\_node = new Node<T>{value, Head->Prev, Head};  Head->Prev->Next = new\_node;  Head->Prev = new\_node;  }  } |

(3) 實作 Rotate 函式

Rotate 在一般的資料型態中很難實現，但是在環狀鏈結串列 Circular Linked List 中則很容易。

如果鏈結串列的長度是 5，但是傳入的引數是 28，此時應該如何處理呢？因為每旋轉 5 筆資料後會與原本的順序相同，所以實際需要旋轉的次數是傳入的引數 offset 對鏈結串列的長度 Len 取餘數，即「offset % Len」。

另外，先前寫的類別中沒有 Len 成員，隨後會加上。

|  |  |
| --- | --- |
| Rotate | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | template <typename T>  void Linked\_list<T>::Rotate(int offset){  // 例外處理：空 list  if (Head == nullptr)  return ;  // 一般情形  // 計算實際需要把 Head 往後調整幾筆資料  offset = offset % Len;  // 根據計算出的移動次數，移動 Head 指向的位置  for(int i=0;i<offset;i++){  Head = Head->Next;  }  } |

(4) 加上私有成員 Len

要使用 Len 需要做以下三件事：

|  |  |
| --- | --- |
| A. 新增 Linked\_List 的私有成員 Len | |
| 1  2  3  4  5  6  7 | template <typename T>  class Linked\_List{  private:  Node<T> Head;  int Len;  public: ...  } |

|  |  |
| --- | --- |
| B. 在建構式把 Len 的初始值設為 0 | |
| 1  2  3  4  5 | template <typename T>  Linked\_List<T>::Linked\_List(){  Head = nullptr;  Len = 0;  } |

|  |  |
| --- | --- |
| C. 每次 Push\_Back 一筆資料後，調整 Len | |
| 1  2  3  4  5 | template <typename T>  void Linked\_List<T>::Push\_Back(T value){  ...  Len++;  } |

(5) 測試 Circular Linked List

|  |  |
| --- | --- |
| 測試 Circular Linked List | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | int main(){  Linked\_List<int> List;  List.Push\_Back(1);  List.Push\_Back(2);  List.Push\_Back(3);  List.Push\_Back(4);  List.Push\_Back(5);  List.Push\_Back(6);  List.Push\_Back(7);  List.Push\_Back(8);  // 標記是否為第一次走到 Head  bool first = true;  // 如果 iter 是第一次走到 Head 或者 Head 以外的其他節點  // 則繼續執行迴圈，否則跳出  for (auto iter=List.Begin() ; iter!=List.Begin() || first ; iter++){  // 印出當前節點的值  cout << \*iter << " ";  // 第一次執行時就會把 first 設成 false  if (first)  first = false;  }  cout << endl;  // 測試 Rotate，向後三筆資料  List.Rotate(3);  first = true;  for (auto iter=List.Begin() ; iter!=List.Begin() || first ; iter++){  cout << \*iter << " ";  if(first)  first = false;  }  cout << endl;  } |
| 執行結果 | |
| 1 2 3 4 5 6 7 8  4 5 6 7 8 1 2 3 | |

只要單純改變 Head 所指到的節點，就可以成功達成 Rotate 的效果，這是環狀鏈結串列最常見的應用之一。

# 第四節：C++ STL中的 list

這章的最後，要來看如何使用 STL 裡的鏈結串列。

首先，和使用向量時類似，要引入相對應的函式庫，這裡引入的是 <list>，之後就可以使用 C++ 提供的鏈結串列。

宣告方法是「list<資料型別>」，代表希望鏈結串列當中裝什麼類型的資料，後面再加上變數名稱。

宣告迭代器時，因為迭代器是寫在 list 類別之下，因此要加上「範圍運算子 ::」，代表現在宣告的是 list 類別下的迭代器，並且也要在後面取個變數名稱。

## 1. list 的使用方法

A. 引入標頭檔

#include <list>

B. 宣告

list<資料型別> 變數名稱;

C. 迭代器

list<資料型別>::iterator 變數名稱;

### (1) list 的優缺點

實務上，STL 裡的 list 比較少用。

A. 資料不連續，不支援索引值查詢

B. 搜尋資料要

C. 循序搜尋也比向量還要慢

向量可以直接往相鄰的記憶體空間移動，像把 RPG 裡的關主全部依序排成一列，容易尋找；鏈結串列則要先取出下一筆資料的位置，再到那個位置把資料取出來，比較花時間，就像關主四散在各地的話就比較難找到。

通常需要手刻 Linked List，比如待會會練習的題目就是，因此務必熟悉指標的使用與操作。

### (2) list 的常用操作

list 語法的使用相對沒有那麼重要，因為實務上很少題目可以用 STL 裡的 list 來解題，更多時候會需要自己寫一個基本版的 Linked List 使用；很多題目考的就是實際寫插入、旋轉等等的功能出來，所以要很了解指標的作用，包含哪個指標應該指向哪個節點。

首先，list 可以使用 begin 和 end，基本上所有容器都會支援這兩個函式。

接下來，push 都是新增資料，back 代表的是尾端，front 代表的是頭端；pop 則是刪除資料，pop\_back 代表的是刪除尾項，pop\_front 代表的是刪除頭項。

|  |  |  |  |
| --- | --- | --- | --- |
| list 的常用操作 | | | |
| list.begin()  list.end()  list.empty()  list.back()  list.front()  list.push\_back()  list.push\_front()  list.pop\_back()  list.pop\_front()  list.clear() | 回傳第一個迭代器  回傳尾端後一個位置的迭代器  確認 list 是否為空  回傳最後一個元素的值  回傳第一個元素  在尾端新增一個元素  在開頭新增一個元素  刪除最後一個元素  刪除第一個元素  清空所有元素 | list.insert()  list.remove()  list.reverse()  list.size()  list.sort()  list.swap()  list.unique()  list.merge()  list.splice()  advance() | 插入元素至特定位置  刪除特定元素  反轉整個 list  回傳 list 的長度  把 list 裡的資料排序  互換兩個 list 的值  刪除 list 裡重複的值  合併兩個 list  切割兩個 list  移動迭代器 |

移動迭代器 advance 是外部函式，所以其他函式都需要寫成 list.[函式名稱]，只有 advance 不需要。

屬於迭代器的操作則基本上是共通的，這些操作不限於 Linked List，也可以用在向量和後面會提到的 set 和 map 等資料結構上：

|  |  |
| --- | --- |
| 迭代器的操作 | |
| advance(it, n)  distance(it, it)  begin(container)  end(container)  prev(it)  next(it) | 迭代器 it 前後移動 n 步，n > 0 時往後，n < 0 時往前  回傳兩迭代器的距離  回傳容器中第一個元素的迭代器  回傳容器中最後一個元素下一個的迭代器（在容器外）  回傳迭代器 it 前面的迭代器  回傳迭代器 it 後面的迭代器 |

## 2. STL list 的使用

|  |  |
| --- | --- |
| 使用 STL 中的 list | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | #include <iostream>  #include <list>  using namespace std;  // print\_list：把 list 中的資料印出來（方便測試）  void print\_list(list<int> List){  for(auto iter=List.begin();iter!=List.end();iter++)  cout << \*iter << " ";  cout << endl;  }  int main(){  list<int> List;  List.assign(3,0); // 0 0 0  List.push\_back(1); // 0 0 0 1  List.push\_back(2); // 0 0 0 1 2  List.push\_back(3); // 0 0 0 1 2 3  print\_list(List);  cout << "Front: " << List.front() << endl; // 輸出 0  cout << "Back: " << List.back() << endl; // 輸出 3    List.pop\_front(); // 0 0 1 2 3  List.pop\_Back(); // 0 0 1 2    auto iter = List.begin(); // iter 指到開頭的 0  advance(iter, 2); // iter -> 1  List.insert(iter, 4); // 0 0 4 1 2, iter 仍然指向 1  advance(iter, 1); // iter -> 2  List.erase(iter); // 0 0 4 1  print\_list(List);    List.sort(); // 0 0 1 4  cout << "Sort:";  print\_List(List);    List.reverse(); // 4 1 0 0  cout << "Reverse:";  print\_list(List);  return 0;  } |

## 3. STL 中 list 的特性

A. list 底層是 double Linked List

a. 每個節點會有兩個 pointer ，一個指向前一筆、一個指向後一筆資料

b. 另外有兩個 pointer，Head 指向頭項，Tail 指向尾項

c. 因此頭尾端的新增刪除只需要

B. 可在任意位置新增或刪除

a. 不計搜尋的時間時為 ，不受資料比數影響

b. 效率佳

C. 搜尋需要線性時間

a. 執行「把所有 1 的值都刪掉」這類的操作時仍然需要進行搜尋

b. 會比 vector 還慢

D. 「forward\_list」這個資料結構則是 single Linked List

A. 不支援隨機訪問或索引值：不能使用下標運算子 []

B. 仍然需要額外空間存放記憶體位置

## 4. 輸入與輸出 list 的資料

宣告並建立 STL 中的 list

A. 儲存整數型態並依序輸入值，直到輸入值為 0 時結束

B. 利用迭代器印出鏈結串列中所有資料

|  |  |
| --- | --- |
| 輸入並印出 list 的資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <iostream>  #include <list>  using namespace std;  int main(){  // 宣告整數型態的鏈結串列 data  list<int> data;  int value;  // 輸入至少會做一次，可以使用 do while  do {  cout << "Please enter a value:" << endl;  cin >> value;  if(value)  data.push\_back(value);  }  while(value); // 輸入值 value 為 0 時等於 false，結束迴圈  // 迭代器 iter 從 begin 開始，往後移動到 end 時結束  for( auto iter=data.begin();iter!=data.end();iter++){  // 利用取值運算子 \* 印出資料  cout << \*iter << " ";  }  } |
| 執行結果 | |
| Please enter a value:  >> 5  Please enter a value:  >> 6  Please enter a value:  >> 7  Please enter a value:  >> 8  Please enter a value:  >> 9  Please enter a value:  >> 0  5 6 7 8 9 | |

## 5. LeetCode #24. 交換串列節點 Swap Nodes in Pairs

A. 題目

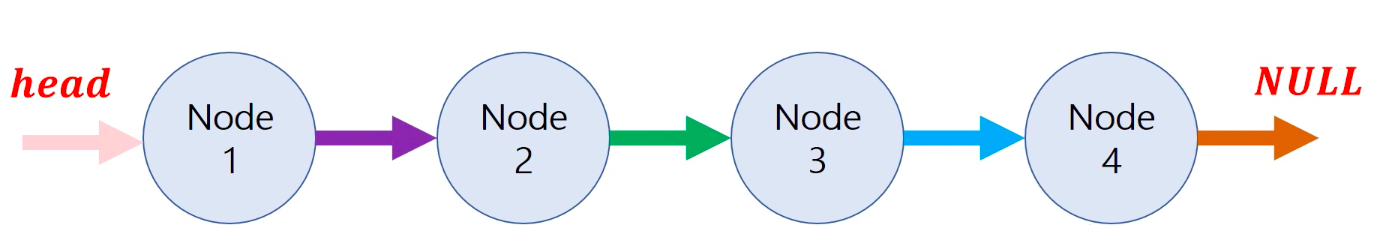
給定一個鏈結串列，將節點每兩個作為一組，每組各自前後交換後，回傳串列的 head。

本題要求不能透過改寫節點中的「值」來完成，必須改變節點的順序。

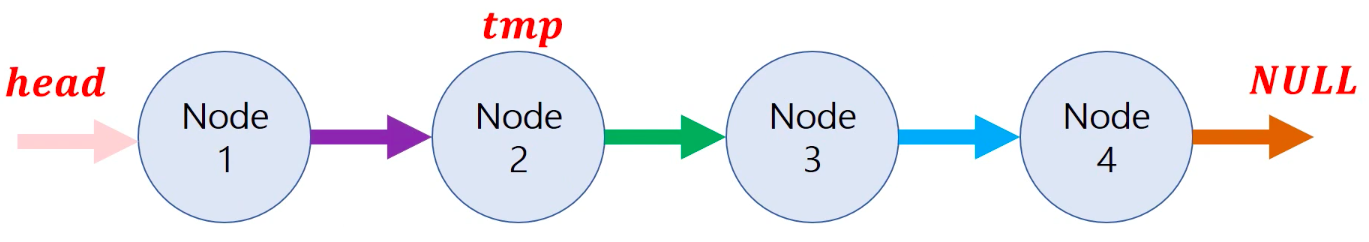
B. 出處：https://leetcode.com/problems/swap-nodes-in-pairs/

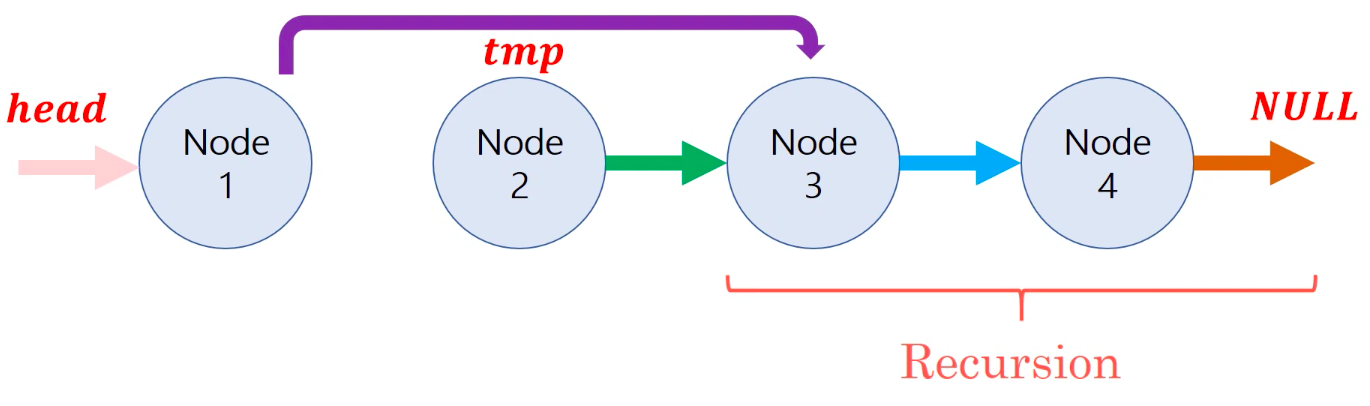
C. 說明

這題要把鏈結串列的節點兩兩一對做交換，比方說一開始鏈結串列是 1 -> 2 -> 3 -> 4，交換後會成為 2 -> 1 -> 4 -> 3。因為「互換」這個動作一直進行，每次都是互換兩個，因此可以用遞迴的形式來完成。

D. 解題邏輯

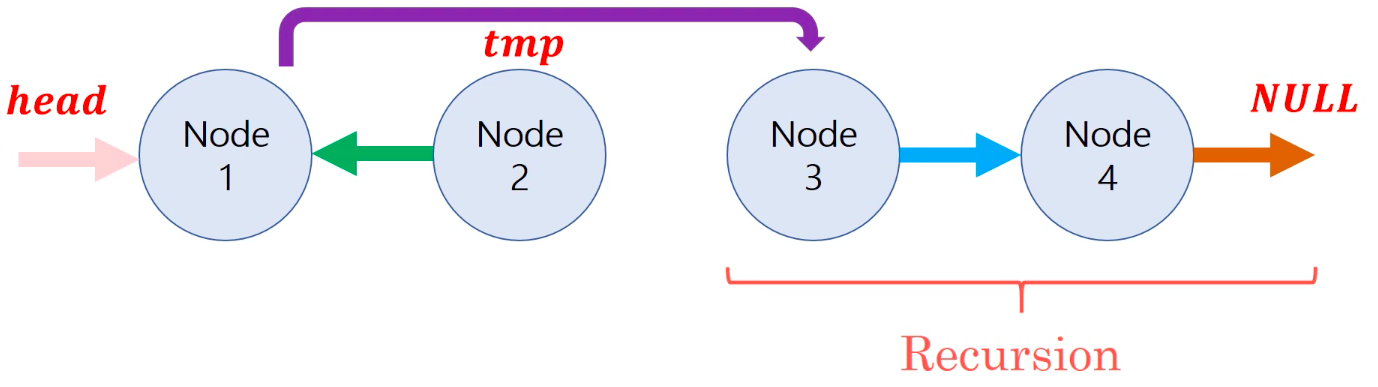
目標是先互換 1 和 2 這兩個節點，其中 Node1 要指向 Node3，Node 2 要指向 Node 1，但是如果 Node 1 直接指向 Node 3，Node 2 的位置就不見了，因此我們需要把 Node 2（Head 的 next）存起來。



隨後才可以讓 Node 1 指向 Node 2 後面、從 Node 3 開始的串列。

從 Node 3 開始的子串列也需要進行上述兩兩互換的過程，因此可以再呼叫一次同一個 swapPairs 函式，處理完成後將這個子串列的（處理完會以 Node4 開頭）位置回傳，接到 Node1 後方。

Node 1 順利指到後面串列的開頭後，還要讓剛才儲存起來的 tmp（即Node 2）的 next 指向 Node 1。



這題基本上是指標題的延伸，重點是要清楚「互換後每個指標要指向誰」。

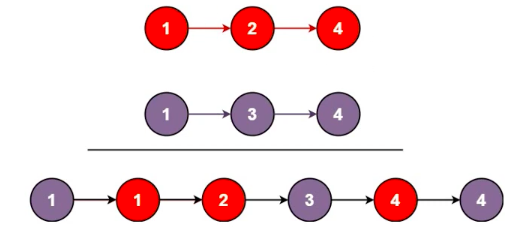
|  |  |
| --- | --- |
| 交換串列節點 Swap Nodes in Pairs | |
| A. Node\* tmp = head->next // tmp = Node 2  // Node 1、2 除外的子串列也是用同樣的函式遞迴處理  // tmp->next 是 Node 3  B. head->next = swapPairs(tmp->next) // swapPairs(Node 3)  C. tmp->next = head // Node 2 -> Node 1  D. return tmp // return Node 2 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Solution{  public:  // 輸入的引數是鏈結串列的第一個節點的指標  ListNode\* swapPairs(ListNode\* head){  // 邊界條件：  // 節點總數是奇數時，會剩下最後一個節點  // 偶數時則會剛好用完指向空指標  // 遇到這兩種情形時，直接回傳這個指標（或為 nullptr）  if (head->next == nullptr || head == nullptr)  return head;  // Step A  ListNode\* tmp = head->next;  // Step B  head->next = swapPairs(tmp->next);  // Step C  tmp->next = head;  // Step D  return tmp;  }  } |

## 6. LeetCode #21. 合併兩個排序好的串列 Merge Two Sorted Lists

A. 題目

合併兩個已經排序好的鏈結串列，並回傳為單一串列，這兩個串列的合併應透過改變指標指向進行。

B. 出處：https://leetcode.com/problems/merge-two-sorted-lists/

C. 說明

本題要合併兩個已經排序好的 linked list，比方有兩個 list [1 2 4] 和 [1 2 4]，要把它合併成單一一個排序好的 list [1 2 2 3 4 4]。

D. 解題邏輯

本題思路不難，因為兩個 linked list 都是左邊最小，所以先比較兩個串列最左邊的節點，把較小者優先插到新的鏈結串列裡面，後面以相同作法繼續進行。

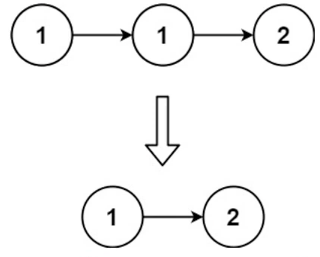
|  |  |
| --- | --- |
| 合併兩個排序好的串列 Merge Two Sorted Lists | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | class Solution{  public:  // 傳入的是兩個 list 的開頭節點 l1 和 l2  ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2){    // 先設定一個節點 assign 給 head  // 這個節點只是為了方便初始化  ListNode\* head = new ListNode(0);  // l1 和 l2 分別在兩個傳入的 list 上移動  // 每次把其中一個節點加入新串列後  // 就把對應的 l1 或 l2 往後移動一筆資料  // 當 l1 和 l2 都有資料（非 nullptr）時  while (l1 && l2){  // 如果 l1 所在節點比 l2 所在節點的資料小  if (l1->val < l2->val){  // 選擇把 l1 節點加入新的串列中  current->next = l1;  // l1 在自己所在的 list 上往後移動一筆資料  l1 = l1->next;  }  // l2 所在節點比 l1 所在節點小（或相等）時  else {  // 把 l2 節點加入新的串列中  current->next = l2;  // l2 在自己所在的 list 上往後移動一筆資料  l2 = l2.next;  }  // 最新加入的節點是 current->next  // 接下來要在 current->next 後面再加上下一筆資料  // 所以 current 要後移  current = current->next;  }  // 如果 l2 所在的是空 list，或者 l2 所在的 list 提前跑完了  // 新的 list 就往後接上 l1 所在 list 剩餘的部分  if (l1)  current->next = l1;  // 如果 l1 所在的是空 list，或者 l1 先跑完，l2 還有剩下  // 新的 list 接上 l2 所在 list 剩餘的部分  else if (l2)  current->next = l2;  // 第一個節點 head 是用來初始化的節點  // 不是我們想回傳的 list 的開頭  // 回傳的開頭應該是 head 的下一個節點  return head->next;  }  }; |

## 7. LeetCode #83. 移除已排序的 list 中重複的項目Remove Duplicates from Sorted List

A. 題目

給定一個指向已排序的 list 的指標 head，將該指到的 list 中所有資料重複者刪除，使得每個元素的值都只出現一次，回傳指向該排序完且處理完成的 list 的指標 head。

B. https://leetcode.com/problems/remove-duplicates-from-sorted-list/

C. 說明

給定一個已經排序好的 list，要將裡面重複的資料刪除。

D. 解題邏輯

因為已經排序完成，所以只需要比較目前的節點和下一個有沒有一樣，如果不一樣的話，不進行處理繼續往下做，如果一樣，則把下一個節點刪掉。

|  |  |
| --- | --- |
| 移除已排序的 list 中重複的項目 Remove Duplicates from Sorted List | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | class Solution{  public:  ListNode\* deleteDuplicates(ListNode\* head){  // 例外處理：空 list 或只有一筆資料，直接回傳  if (head == nullptr || head->next == nullptr)  return head;  // 一般情形，current 指標從 head 開始往後處理  ListNode\* current = head;  // 要比較目前處理的 current 指標和下一個節點是不是相同  // current 和 current->next 都不能是空指標  // while(current) 的寫法意義等同 while(current!=nullptr)  while (current && current->next){  // 目前 current 指到節點的值與下一個節點的值相同時  // 把下一個節點刪掉  // A A B 若要把第二個 A 刪除，則第一個 A 要指向 B  // 1st A: current, 2nd A: current->next, B: current->next->next  if (current->val == current->next->val)  // 直接把第二個 A 刪掉，就沒辦法找到 B 的位置  // 因此先用 tmp 把 B 的位置儲存起來  ListNode\* tmp = current->next->next;  // 刪除第二個 A  delete current->next;  // 把第一個 A 的 next 指向 B  current->next = tmp;  }  // 目前節點的值與下一個節點的值不同時  // 直接往後處理（current 往後移動一筆資料）  else {  current = current->next;  }  // 最開頭的節點不會被刪掉，因此開頭仍然是 head  return head;  }  } |

## 8. LeetCode #707. 設計鏈結串列 Design Linked List

A. 題目

設計一個手動實現的鏈結串列 linked list，該串列為單向或雙向皆可。

單向鏈結串列中的節點中應該有兩個資料成員：val 和 next，val 是該節點代表的資料，next 則是一個指標，指向下一個節點。

如果撰寫的是雙向鏈結串列，應多加一個資料成員 prev 來指向串列中的前一個節點。節點由 0 開始編號。

B. 出處：https://leetcode.com/problems/design-linked-list/

C. 說明

本題要求手動設計一個 linked list，網站上會提供這個 list 的介面，裡面有

a. 建構式

b. get：取出特定索引值的資料

c. addAtHead：在開頭新增資料，即Push\_Front

d. addAtTail：在尾端新增資料，即 Push\_Back

e. addAtIndex：在特定索引值新增資料

f. deleteAtIndex：刪除特定索引值資料

這些函式的實現方式和前面的講解是一樣的，因函式較多，以下分段列出。

|  |  |
| --- | --- |
| 設計鏈結串列 Design Linked List | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class MyLinkedList{  // 節點 Node 的結構  struct ListNode{  int val;  ListNode\* next;  };  // 兩個 ListNode 指標分別指向開頭與結尾  ListNode\* head;  ListNode\* tail;  // 目前鏈結串列的長度  int len;  public:  … // 函式內容直接寫在這裡  } |

|  |  |
| --- | --- |
| 建構式 | |
| 1  2  3  4  5  6  7  8 | MyLinkedList(){  // 初始化 head, tail 與 len  head = nullptr;  tail = nullptr;  len = 0;  } |

|  |  |
| --- | --- |
| 取出特定索引值資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | int get(int index){  // 例外處理：index 超出邊界  if(index<0 || index>=len)  return -1;  // 節點 current 從開頭 head 開始往後處理  ListNode\* current = head;  // 移動到第 index 筆資料（一共需要移動 index-1 次）  for(int i=0;i<index;i++){  current = current->next;  }  // 回傳第 index 個節點（現在的 current）的值  return current->val;  } |

|  |  |
| --- | --- |
| 頭端新增資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | void addAtHead(int val){  // 建立一個新的節點並 assign 給 head  // 資料為 val，next 是原本的 head  head = new ListNode(val, head);  // 調整長度 len  len++;  // 例外處理  // 如果原先是空 list，加上新節點後只有一個節點  // 因此 tail 也要指向這個節點  if(tail == nullptr)  tail = head;  }  } |

|  |  |
| --- | --- |
| 尾端新增資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | void addAtTail(int val){  // 一般情形，即 if (tail!=nullptr)  if (tail){  // 新增一個節點在 tail 後面  // 資料為 val，next 為 nullptr  tail->next = new ListNode(val, nullptr);  // 把 tail 移到新的最後一個節點  tail = tail->next;  }  // 例外處理：空 list，head 和 tail 都指到新增的節點  else{  head = tail = new ListNode{val, nullptr};  }  // 調整長度 len  len++;  } |

|  |  |
| --- | --- |
| 在特定索引值新增資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | void addAtIndex(int index, int val){  // 例外處理：index 超出邊界  if (index<0 || index>len)  return ;  // 例外處理：index 在尾端  if(index == len){  // 呼叫尾端插入函式  addAtTail(val);  // 例外處理：index 在頭端  } else if (index == 0){  // 呼叫頭端插入函式  addAtHead(val);  // 一般情形  } else {  // A B C 變為 A B D C，要把 D 插入到 C（index 2） 之前  // current 要移動到 B，從 A 開始共移動一次  // 所以 current 總共往後移動 index-1 次  // 注意迴圈中 i 是從 1 開始而非從 0 開始  ListNode\* current = head;  for(int i=1;i<index;i++){  current = current->next;  }  // current: B  // current->next: C  // 把 C 的位置儲存在 tmp 中  ListNode\* tmp = current->next;  // 讓 B 指到新的節點  // 內容是 D，next 是 tmp，也就是 C  current->next = new ListNode(val, tmp);  // 調整長度 len  len++;  }  } |

|  |  |
| --- | --- |
| 刪除特定索引值的資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | void deleteAtIndex(int index){  // 例外處理：index 超出邊界  if(index<0 || index>=len)  return;  // 例外處理：刪除頭項  if (index == 0){  // 先記錄 head 的位置在 tmp 中  // 再把 head 往後一格到第二筆資料  // 等同刪除第一筆資料，最後釋放 tmp  ListNode\* tmp = head;  head = head->next;  delete tmp;  // 調整長度 len  len--;      // 例外處理  // 如果原先只有一個節點，刪除後變為空 list  // 此時 tail 也要指向 nullptr  if (head == nullptr){  tail = nullptr;  }  return ;  }  // 一般情形  // A B C D, delete C, current 要移動到 B，C 的 index 是 2  // 我們要移動到 B 需要從 A 開始移動一次  // 因此共移動 index-1 次  // 注意下面 i 從 1 而非 0 開始  for (int i=1;i<index;i++){  current = current->next;  }  // 例外處理  // 當刪除的是尾端資料時，要把 tail 往前移動一格到 current  if (index == len-1){  delete tail;  current->next = nullptr;  tail = current;  }  // 刪除的不是尾端資料時  else {  // current:B  // current->next:C  // current->next->next:D  // 把 C 的位置存在 tmp 以便待會刪除  ListNode\* tmp = current->next;  // B 的 next 指向 D  current->next = current->next->next;  // 刪除 C 節點  delete tmp;  // 調整長度 len  len--;  }  } |

## 9. 用鏈結串列儲存字串

A. 題目

已知 ,

試印出 的前 1000 個字元。

B. 解題邏輯

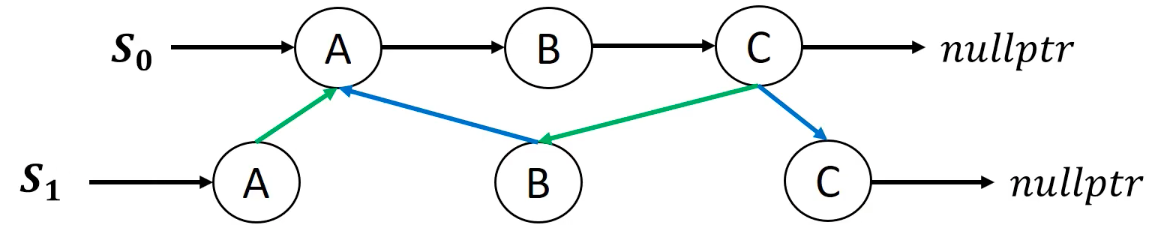
直覺上，可能會想要直接把這個字串生成出來，但是根據下面的算式， = 3 + 2 x 3 = 9, = 3 + 2 x 9 = 21，很快就會超過容量限制，所以不能直接把字串生成出來再印出前 1000 個字元， 個字元記憶體是放不下的。

（等比級數公式：）

屬於

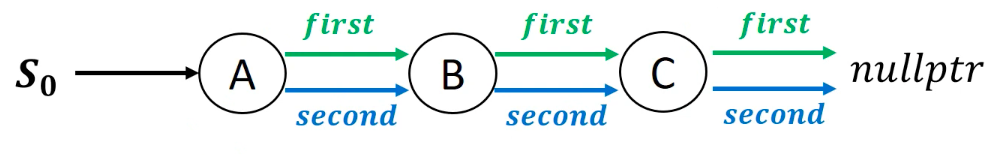
再觀察一下，會發現前面一個字串是不變的，因此不需要去改變它，只要用指標去指就好。

這可以用兩個字串 A B C 實現：讓 的 A 指到 中的 A B C，等到跑完後再指回 的 B，再跑完一次 後最後再指到 的 C。

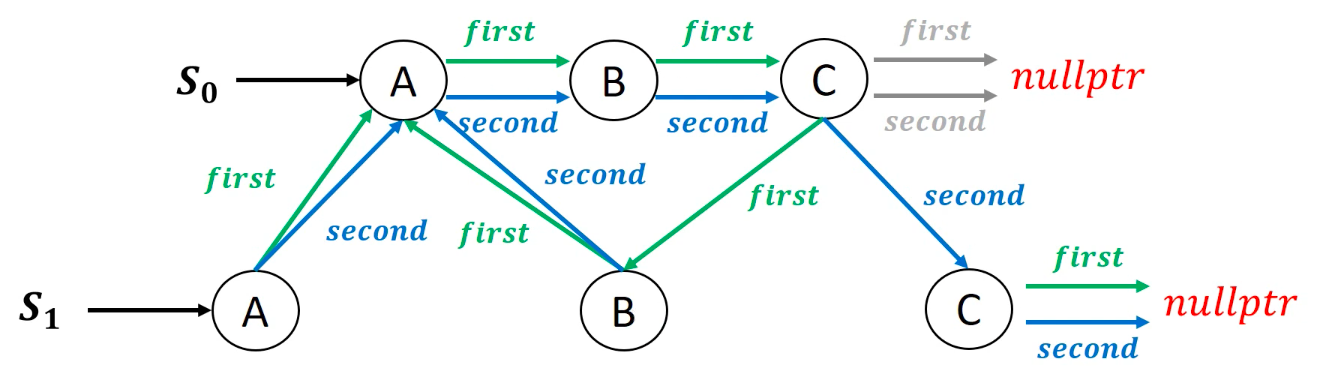


這樣 就不需要真的用 9 個字元來表示，使用兩個 'ABC' 加上一些指標就可以做到。這樣一來，每個 都只比 多了三個字元，總共只要花費取決於 n 的記憶體，儲存上就不會出問題。

屬於

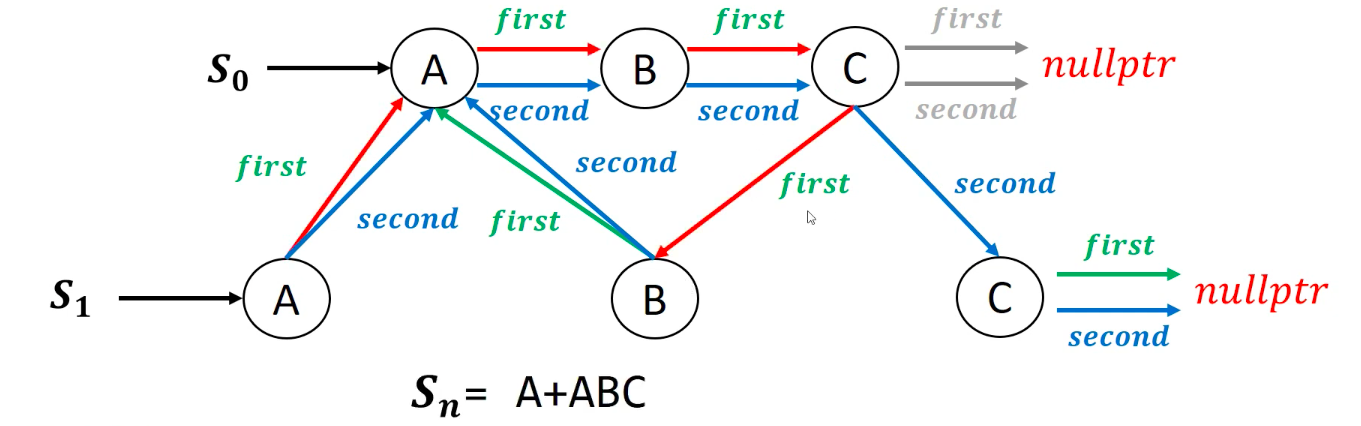
然而，重點是要實現「兩次到達同一個節點時」「往後走到不同節點」的功能，這可以透過讓每個節點都有 first 和 second 兩個指標來達成。

對於 來說，A 的兩個指標都指向 B，B 的兩個指標都指向 C，但是 C 的兩個指標中 first 要指向 的 B，second 則是指向 的 C。

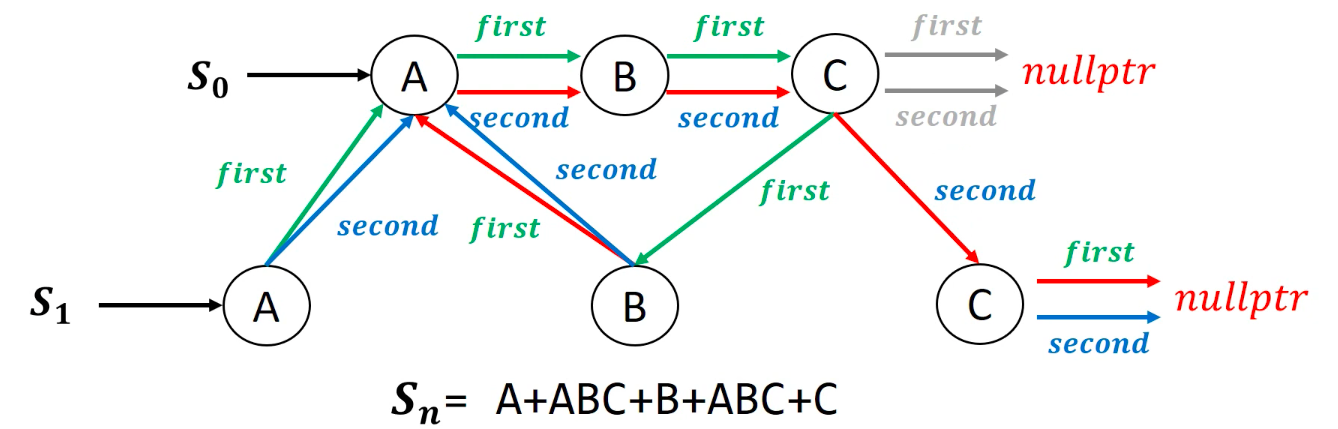


對於 來說，A 的兩個指標都指向 開頭的 A，B 的兩個指標也都指向 的 A，C 的兩個指標則暫時指向 nullptr（建立 之後要修改指向），這樣從 S1 的 A 出發，就會經過 A -> (ABC) -> B -> (ABC) -> C 的過程，最後到達空指標。

first 指標會產生的字串：



second 指標會產生的字串：



C. Pseudo Code

節點 node

a. char: data

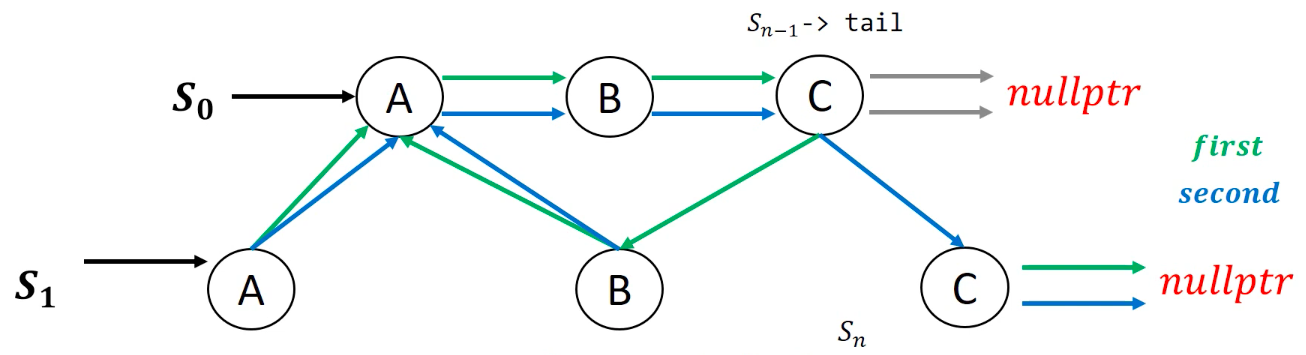
b. node\*: first

c. node\*: second

d. bool: first

每個節點會有 4 筆資料，第一筆資料 data 是一個字元，可能是 ABC，再來是兩個指標 first 和 second 代表第一次和第二次走分別要走到的位置，最後還有一個 boolean first 代表目前是第一次還是第二次走。

設定 first 與 second：



a. new A, B, C

b. A->first, A->second = ->head

c. B->first, B->second = ->head

d. C->first, C->second = nullptr

e. ->tail->first = &B // 的 C 的 first 指向 的 B

f. ->tail->second = &C // 的 C 的 second 指向 的 C

g. ->head = &A

h. ->tail = &C

按照剛剛的想法可以寫出上面的 pseudo code，使用一個迴圈開出 到 之後，就可以利用一個迭代器去把前 1,000 個指到的節點的值都印出來。

這是一個很經典的鏈結串列應用，因為發現「資料當中有很多重複」，既然重複了，不用在記憶體中塞滿一樣的東西，用鏈結串列把它們依序串連起來即可。

|  |  |
| --- | --- |
| 用鏈結串列儲存字串 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <iostream>  using namespace std;  # 建立節點 Node 結構  struct Node{  // 一個節點的資料是一個字元，如 'A' 或 'C'  char data;  // first 和 second 記錄第一次和第二次要走到的下一個節點位置  Node\* first;  Node\* second;  // 以一個 boolean 值在 first 與 second 間切換  bool read\_first;  };  int main(){  // 使用一個陣列儲存 51 個字串的開頭位置  Node\* str\_start[51];  // 開出 3 個節點放到第一個字串中  // 把開頭的 A 用 str\_start[0] 記錄  Node\* C = new Node{'C', nullptr, nullptr, true};  Node\* B = new Node{'B', C, C, true};  Node\* A = new Node{'A', B, B, true};  str\_start[0] = A;  return 0;  } |

可以 main 中再加上幾行測試這三個節點：

|  |  |
| --- | --- |
| 測試 | |
| 1  2  3  4  5  6  7  8  9 | Node\* current = str\_start[0];  while(current->first){  cout << current->data << " ";  current = current->first;  }  // 上面的迴圈不會輸出最後一筆資料，要另行輸出  cout << current->data << " "; |
| 執行結果 | |
| A B C | |

|  |  |
| --- | --- |
| 完整版 main | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | int main(){  // 使用兩個陣列儲存 51 個字串的開頭和結尾位置  Node\* str\_head[51];  Node\* str\_tail[51];  // 開出 3 個節點放到第一個字串中  // 把開頭的 A 用 str\_start[0] 記錄  Node\* C = new Node{'C', nullptr, nullptr, true};  Node\* B = new Node{'B', C, C, true};  Node\* A = new Node{'A', B, B, true};  str\_head[0] = A;  str\_tail[0] = C;  // 用迴圈生成 str\_start[1] 到 str\_start[50]  for (int i=1;i<51;i++){  // 建立新的字串的三個節點  C = new Node{'C', nullptr, nullptr, true};  B = new Node{'B', str\_head[i-1], str\_head[i-1], true};  A = new Node{'A', str\_head[i-1], str\_head[i-1], true};  str\_head[i] = A;  str\_tail[i] = C;  // 修改上一個字串的指標指向  str\_tail[i-1]->first = B;  str\_tail[i-1]->second = C;  }  // 從第 51 個字串開頭的 A str\_head[50] 開始往後走  Node\* current = str\_head[50];  // 印出首先走到的 1000 個節點  for (int i=0;i<1000;i++){  // 印出 current 節點的資料  cout << current->data << "->";  // 每走完兩次（走過了一次 first、一次 second）  // 下次再走到還要再往 first 走  // 所以 read\_first 會在 true 和 false 之間交替變換  // 用 go\_first 記錄反轉前「往 first 還是 second 走」的訊息  bool go\_first = current->read\_first;  current->read\_first = !(current->read\_first);  // 根據 go\_first 決定要往 first 還是 second 指到的節點走  // read\_first 已經被反轉了，所以改用 go\_first 決定移動路徑  if (go\_first){  current = current->first;  } else {  current = current->second;  }  }  } |

測試時，可以先修改下面兩行

Node\* current = str\_head[50]; // 從 開頭的 'A' 出發

for(int i=0;i<1000;i++){...} // 印出前 1,000 個走到的字元

分別改為

Node\* current = str\_head[1]; // 從 開頭的 'A' 出發

for(int i=0;i<9;i++){...} // 印出前 9 個走到的字元

此時的執行結果會和題目說明中相同，為 A -> A -> B -> C -> B -> A -> B -> C -> C。

利用這種方法，資料數目的增加速度就不再是 ，而是被改進為 。