程式設計
Ch15. Linked List

Chuan-Chi Lai 賴傳淇

Department of Communications Engineering
National Chung Cheng University

Spring Semester, 2024

Outline

鏈結串列簡介
Introduction to Linked List

鏈結串列簡介 (Introduction to Linked List)

概念

- 之前學過 struct，這次要學的就是如何把 struct 彼此串起來，就跟串貢丸一樣或串珠一樣。
- 如果大家還記得的話，之前我們教過 struct array 就可以做到類似的效果，那為甚麼還需要用到鏈結呢? 讓我們先看下方這個漫畫...

Photo credit:
https://learn.co/lessons/linked-lists-reading

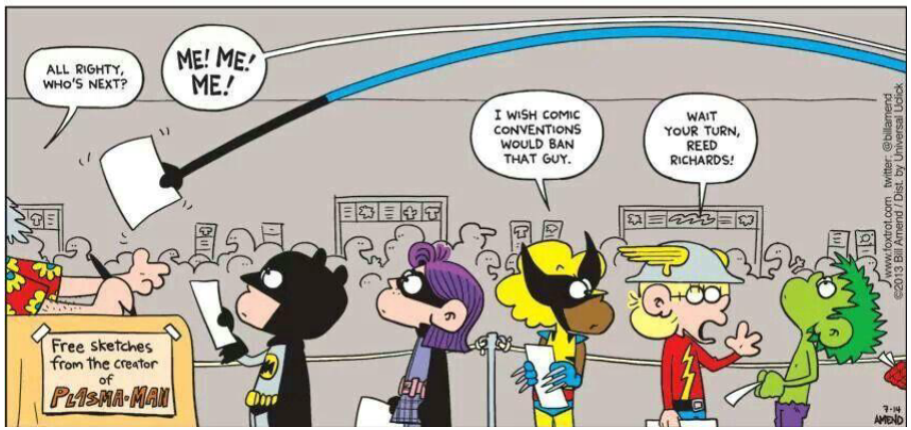
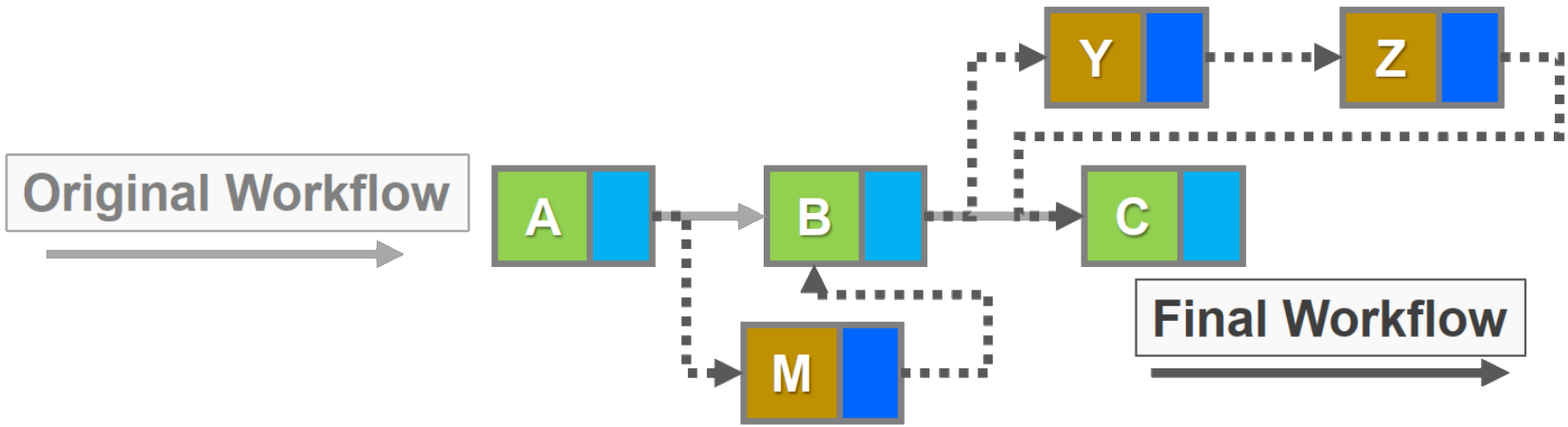


Photo credit: https://medium.com/@luc.highwalker/skip-the-nodes-dcb2fb542aa0

鏈結串列簡介 (Introduction to Linked List)

- 正常的情況就像是我們上面左邊看到的漫畫一樣，問題是現實生活中就會像是上面右邊的一樣。
- 如果以一個專案為例，我們永遠不會知道到底需要多少人才能某一件任務。有可會出現以下兩個情況 (但不只...)：
  1. 做到一半突然有不會的地方，需要請求支援。
  2. 有人中離 @@!



鏈結串列簡介 (Introduction to Linked List)

- 所以這個時候我們就要用到鏈結串列 (linked list)，通常在資料結構與演算法的課程會再仔細介紹他的精神與應用 (很多很多...)。

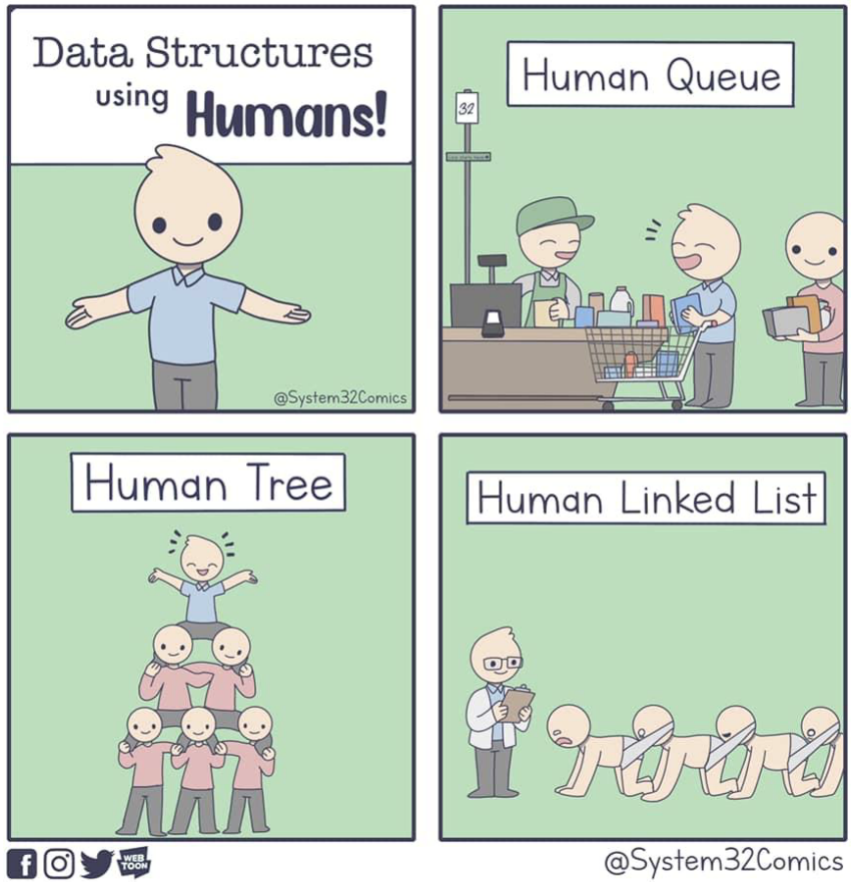


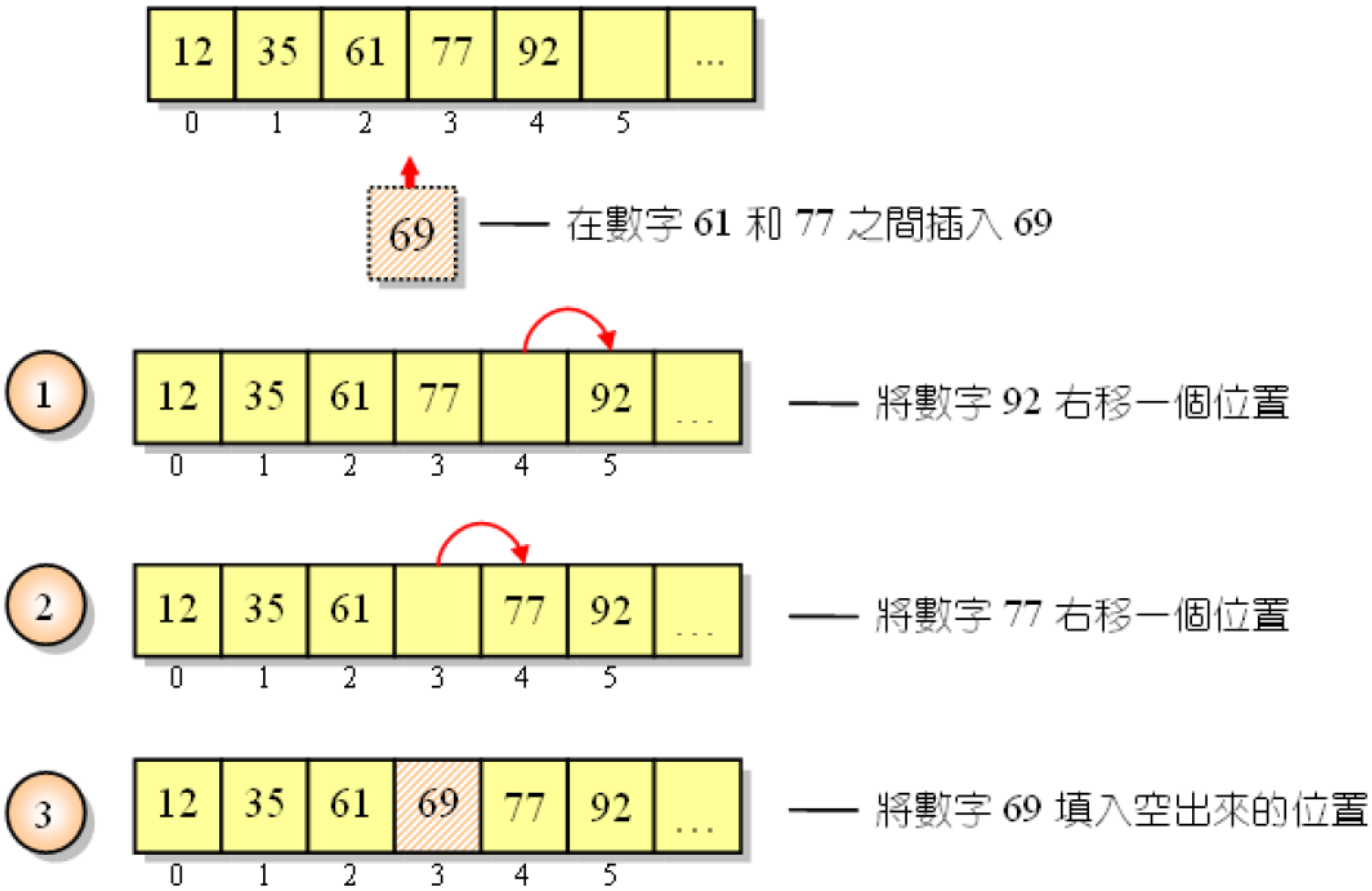
Photo credit: https://www.facebook.com/System32ComicsAdvanced/

串列 (List)

- 有次序的資料可組成一個串列 (list)
- 串列可分為循序串列與鏈結串列
  - 循序串列：存放串列的記憶體是循序的（即有先後次序）。
    - 優點：存取方便
    - 缺點：增加或刪除節點較麻煩，易造成記憶空間不足或浪費
  - 鏈結串列：以指標將存放串列的記憶體鏈結起來
    - 優點：記憶體配置較有彈性
    - 缺點：搜尋某個元素時，可能會較為耗時

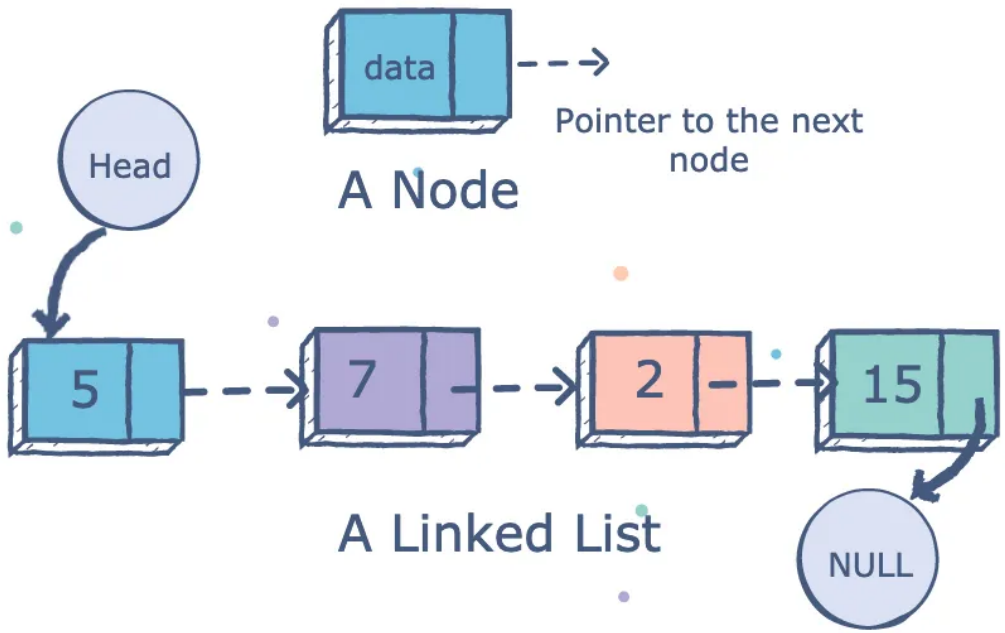
以陣列實作循序串列

- 以陣列實作循序串列，資料的移動會較為頻繁。



鏈結串列

- 鏈結串列是一種常見的資料結構，由多個節點連接而成。每個節點都包含一個儲存的資料和一個指向下一個節點的指標。
- 相比於之前所介紹的陣列，鏈結串列更加的靈活，因為它可以根據需要動態增加或刪除節點。



鏈結串列簡介 (Introduction to Linked List)

- 鏈結串列的主要優點如下:
  - 插入和刪除元素的操作速度很快，不需要像陣列那樣進行大量的元素移動。
  - 鏈結串列的各個節點之型態 (Data Type) 不一定要相同。
  - 鏈結串列的大小可以在運行時動態改變，而不需要事先指定大小。
  - 鏈結串列的節點可以存儲在記憶體的任意位置，不一定要佔用連續的記憶體空間，因此可以更有效地利用記憶體。

鏈結串列簡介 (Introduction to Linked List)

- 鏈結串列的主要缺點如下:
  - 無法隨機存取 (無法透過 index 直接存取)。
  - 循序存取效能差 (因為無法隨機存取，需要先讀取指標，從頭開始遍歷查找)。
  - 需要額外的指標空間，因此占用的記憶體空間比一般陣列更大。
  - 資料容易遺失 (指標若斷裂，資料就會遺失)。

鏈結串列簡介 (Introduction to Linked List)

- 鏈結串列可實現的抽象資料型別 (Abstract Data Type):
  - list(串列)：可在任何位置插入和移除資料項目。
  - stack(堆疊)：其資料項的插入和刪除操作只能夠在堆疊的頂部 (top) 進行。
  - queue(佇列)：其資料項的插入只能在佇列的尾端進行，刪除只能在佇列的頭端進行。

鏈結串列簡介 (Introduction to Linked List)

- 從下表中可以看出，鏈結串列相對於陣列來說在插入和刪除元素方面更加高效，但在存取元素方面效率則較低。
- 此外鏈結串列可以動態調整大小，而陣列則需要預先指定大小。

操作	陣列	鏈結串列
隨機存取	支援	不支援
存取特定元素	$O(1)$	$O(n)$
插入元素	$O(n)$	$O(1)$
刪除元素	$O(n)$	$O(1)$
調整大小	$O(n)$	$O(1)$

自我參考結構
Self-referential Structure

自我參考結構

- 鏈結串列或是二元樹等結構均使用自我參考結構組成。自我參考結構指一個結構含有 1 個或 1 個以上指向「與自身相同的結構」的指標 (pointer) 的成員 (member)。
- 下面是一個自我參考結構的例子：

```
1 struct node
2 {
3     int value;
4     struct node *nextPtr;
5 };
```

自我參考結構 (Self-referential Structure)

- 要注意，結構成員 (member) 不可以是與自身相同的結構類別，只可以是其指標，例如下定義會發生錯誤：

```c
struct node
{
    int value;
    struct node next; //Error
};
```

自我參考結構 (Self-referential Structure)

- 使用 typedef 定義自我參考結構須注意，不能在 typedef 敘述句中預先使用正在定義的符號。

```
1 typedef struct
2 {
3     int value;
4     Node *nextPtr; //Error, 在這行之前尚未定義識別字 Node
5 } Node;
```

自我參考結構 (Self-referential Structure)

- 這裡提供使用 typedef 定義自我參考結構的 2 種方法：

```c
typedef struct node { //加上 struct tag
    int value;
    struct node *nextPtr;
} Node;
```

```c
typedef struct node Node; //先定義好 Node 是 struct node
struct node { //接著定義 struct node
    int value;
    struct node *nextPtr;
};
```

鏈結串列的抽象資料型別
ADT of Linked List

鏈結串列的抽象資料型別

ADT *LinkedList* is

- objects: 由一組節點 (nodes) 所組成的有序串列 (ordered list)，各 Node 除了 Data 欄之外，另外有 ≥ 1 個 Link 欄 (或稱 Pointer)，用以指向其它 Node 之位址。
- functions: TRUE, FALSE $\in$ Boolean and where +, -, <, ==, and = are the usual integer operations.

  基本運算：

push_back() ::=	將節點添加到串列末尾
pop_back() ::=	刪除串列末尾元素
insert_at_index() ::=	插入元素至指定索引位置
delete_at_index() ::=	刪除指定索引位置的元素
find() ::=	查找元素

鏈結串列的抽象資料型別 (ADT of Linked List)

其它運算：

get_size() ::=	取回鏈結串列中的資料項目個數
is_empty() ::=	該鏈結串列是否為空 (沒有任何資料)
push_first() ::=	將節點添加到串列最前面
pop_first() ::=	刪除串列第一個元素
get_first() ::=	取得串列第一個元素
get_back() ::=	取得串列末尾元素
remove() ::=	搜尋特定的資料項目，並加以移除
replace() ::=	搜尋特定的資料項目，並更新其資料
clear() ::=	把整串串列刪除
reverse() ::=	反轉串列
...	

end *LinkedList*

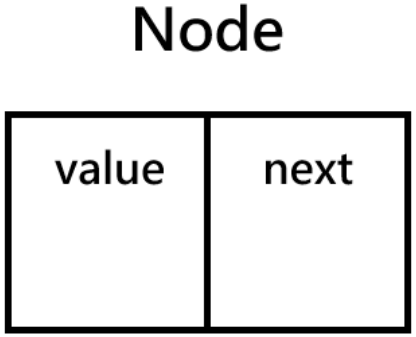
建立鏈結串列
Create a Linked List

建立鏈結串列 (Create a Linked List)

定義鏈結串列節點的結構

- 一個鏈結串列節點通常包含兩個成員 (member)：一個用於存儲節點的值，另一個用於指向下一個節點。

```
1 typedef struct Node {
2     int value;
3     struct Node* next;
4 } Node;
```



建立鏈結串列 (Create a Linked List)

建立空的鏈結串列

- 空的鏈結串列不包含任何節點，因此 head 只需維護一個指向第一個節點的指標，這個指標通常被設置為 NULL。
- 原因: 在鏈結串列中，我們存取下一個元素的方式是透過 next 指標，因此將鏈結串列的最後一個元素指向 NULL 的話，可以避免讀取到其他不屬於鏈結串列的記憶體。

建立鏈結串列 (Create a Linked List)

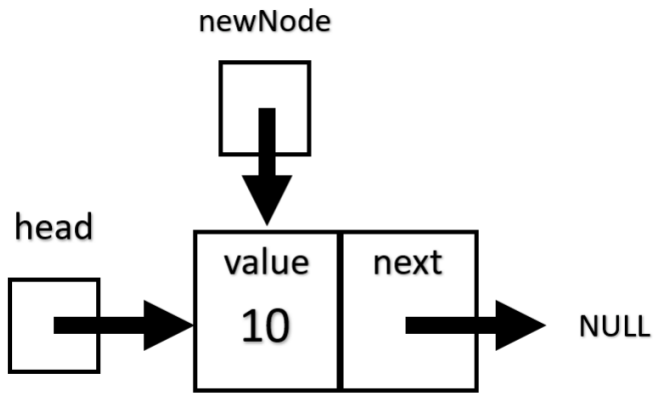
建立空的鏈結串列

```
1 typedef struct Node {
2     int value;
3     struct Node* next;
4 } Node;
5
6 int main() {
7     Node *head = (Node*)malloc(sizeof(Node));
8     head->next = NULL;
9 }
```

建立鏈結串列 (Create a Linked List)

- 接下來試著在空的鏈結串列後方加入一個元素吧。

```
1  int main() {
2      //建立head
3      Node *head = (Node*)malloc(sizeof(Node));
4      head->next = NULL;
5
6      //建立新節點
7      Node *new_node = (Node*)malloc(sizeof(Node));
8      new_node->next = NULL;
9      new_node->value = 10;
10
11     //連接起來
12     head->next = new_node;
13 }
```

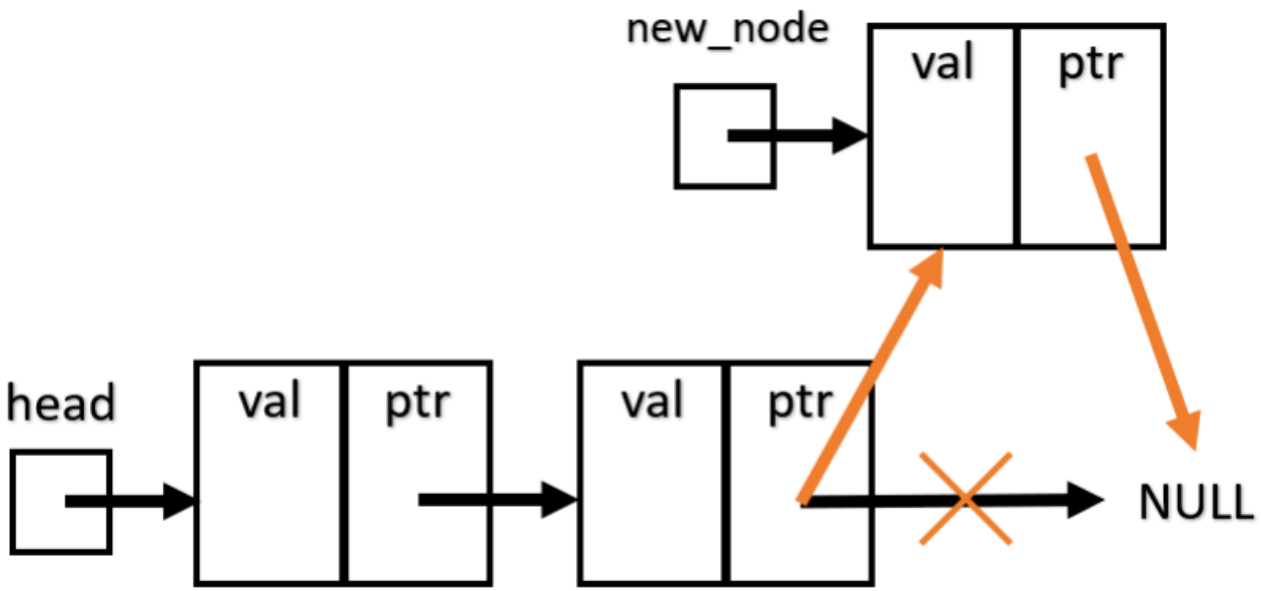


- 大致上理解鏈結串列的概念後，下一章將會介紹如何操作鏈結串列。

鏈結串列操作
Linked List Operations

將節點添加到串列末尾

- 方法: 透過迭代尋找鏈結串列的尾端，接著分配一個新的節點，再將原本尾端節點的 next 指向新節點。



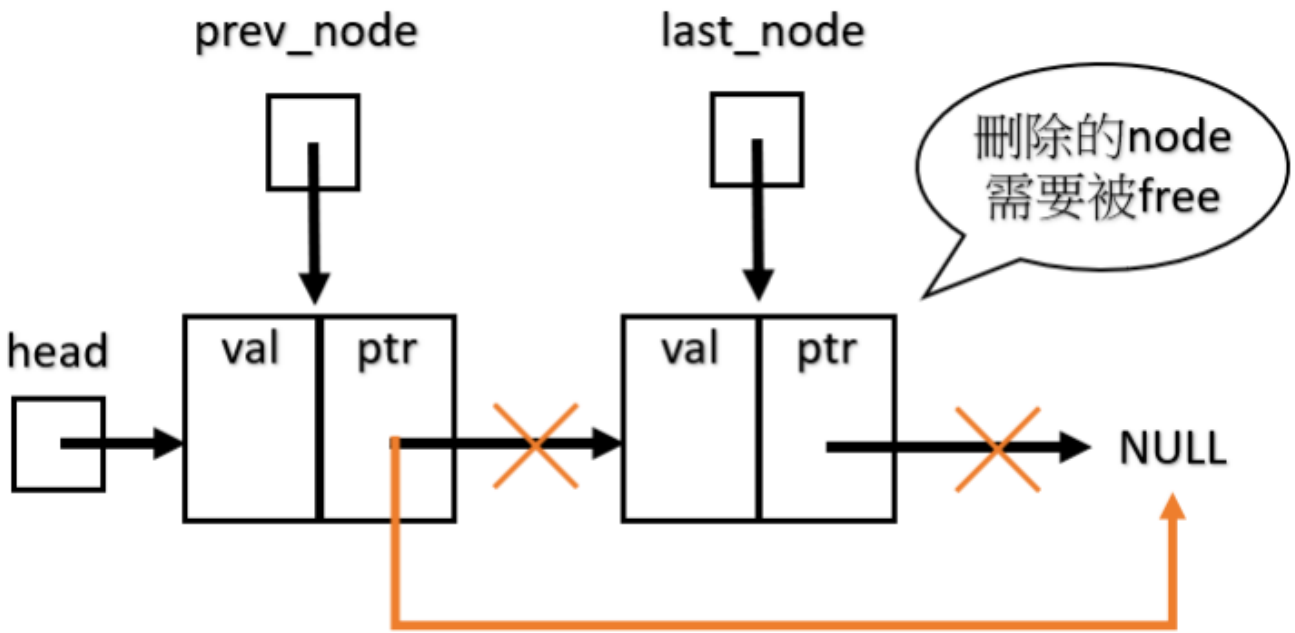
鏈結串列操作 (Linked List Operations)

- push_back() 程式碼：

```
1  void push_back(Node* head, int newValue) {
2      //找到當前串列最後一個節點
3      while (head->next != NULL) {
4          head = head->next;
5      }
6
7      //建立新節點，並將其接到最後一個節點後面
8      Node *new_node = (Node*)malloc(sizeof(Node));
9      new_node->next = NULL; //新節點為最後一個節點，所以其指向為NULL
10     new_node->value = newValue; //將新值存入新節點
11     head->next = new_node; //將新節點接到最後一個節點後面
12 }
```

刪除串列末尾元素

- 方法: 透過迭代找到最後一個元素，然後將其前一個元素的指標設為 NULL，最後釋放最後一個元素的記憶體空間。



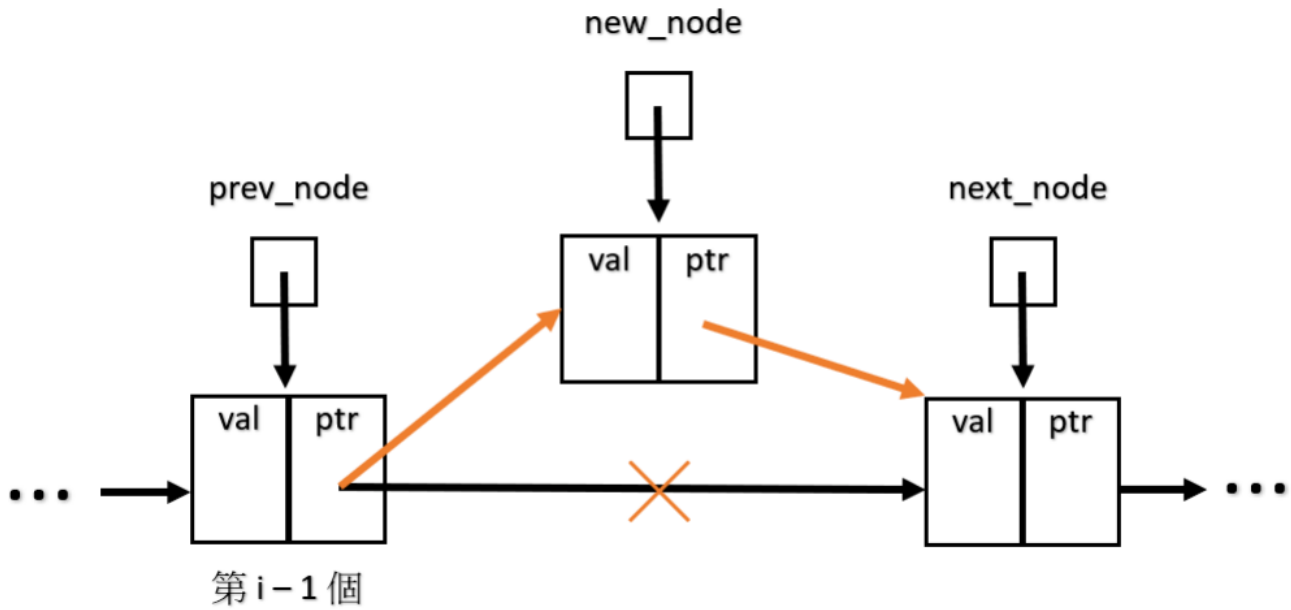
鏈結串列操作 (Linked List Operations)

- pop_back() 程式碼：

```
1  void pop_back(Node* head) {
2      //檢查是否為空的鏈結串列(沒有元素可以刪除)
3      if (head->next == NULL) {
4          printf("The list is empty.\n");
5          return;
6      }
7      Node *prev_node = head;
8      Node *last_node = prev_node->next;
9      //尋找最後一個節點及其前一個節點
10     while (last_node->next != NULL) {
11         prev_node = last_node;
12         last_node = last_node->next;
13     }
14     //刪除最後一個節點
15     prev_node->next = NULL;
16     free(last_node);
17 }
```

插入元素至指定索引位置

- 方法: 透過迭代找到第 i - 1 個元素，先創造一個新的節點，然後將新節點指向原本第 i 個元素 (若不存在就指向 NULL)，最後再將 prev_node 指向新節點。



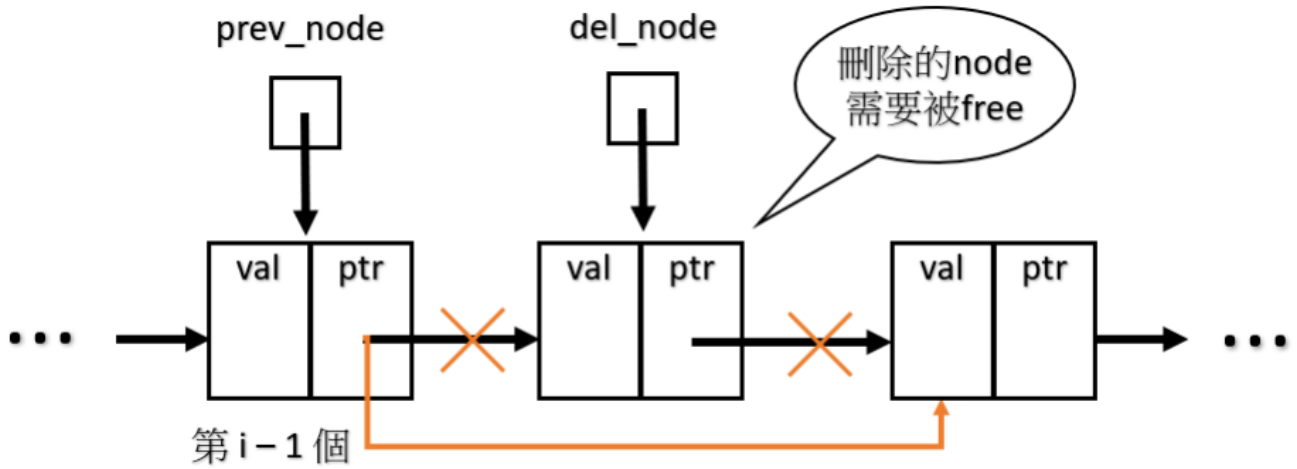
鏈結串列操作 (Linked List Operations)

- insert_at_index() 程式碼：

```
1  void insert_at_index(Node* head, int targetIndex, int newValue) {
2      Node* prev_node = head;
3      int i;
4      //遍歷串列，找到目標索引的前一個節點
5      for (i = 0; i < targetIndex && prev_node->next != NULL; i++) {
6          prev_node = prev_node->next;
7      }
8      //如果找到目標索引的前一個節點
9      if (i == targetIndex) {
10         Node *new_node = (Node*)malloc(sizeof(Node)); //創建新節點
11         new_node->value = newValue; //給新節點賦值
12         new_node->next = prev_node->next; //新節點的指標指向目標索引後一個節點
13         prev_node->next = new_node; //目標索引前一個節的指標指向新節點
14     }
15     else { //如果沒有找到目標索引的前一個節點，插入失敗
16         printf("Insert failed.\n");
17     }
18 }
```

刪除指定索引位置的元素

- 方法: 透過迭代找到欲刪除的前一個節點 (prev_node)，確認要刪除的節點 (del_node) 存在，然後將 prev_node 指向刪除節點的下一個節點，最後釋放刪除節點 (del_node) 的記憶體。



鏈結串列操作 (Linked List Operations)

- delete_at_index() 程式碼：

```
1  void delete_at_index(Node* head, int targetIndex) {
2      Node *prev_node = head;
3      //遍歷串列，找到目標索引的前一個節點
4      for (i = 0; i < targetIndex && prev_node->next != NULL; i++) {
5          prev_node = prev_node->next;
6      }
7      //如果找到目標索引的前一個節點
8      if (i == targetIndex) {
9          Node *del_node = prev_node->next; //找到要刪除的節點
10         prev_node->next = del_node->next; //修改前一個節點的指標
11         free(del_node); //釋放要刪除的節點
12     }
13     else { //如果沒有找到目標索引節點，刪除失敗
14         printf("Delete failed.\n");
15     }
16 }
```

查找元素

- 方法: 透過迭代遍歷整個鏈結串列，假如發現元素就回傳 true，否則回傳 false。

```
1 bool find(Node* head, int targetValue) {
2     while (head->next != NULL) { //假如後面還有節點，繼續搜尋
3         head = head->next;  //指向下一個節點
4         if (head->value == targetValue) { //如果節點的值為目標值
5             return true;    //回傳true
6         }
7     }
8     return false;   //如果沒有找到，回傳false
9 }
```

插入排序 (Insertion Sort)

- 插入排序是一種簡單直觀的排序演算法。將序列分為已排序與未排序兩部份，對於未排序資料，在已排序序列中依序掃描，找到相應位置並插入。如數列 [3, 7, 9, 6, 2] 其排序過程如下：

step	sorted	unsorted
0	[ ]	[3, 7, 9, 6, 2]
1	[3]	[7, 9, 6, 2]
2	[3, 7]	[9, 6, 2]
3	[3, 7, 9]	[6, 2]
4	[3, 6, 7, 9]	[2]
4	[2, 3, 6, 7, 9]	[ ]

練習一 - 插入排序

- 請使用鏈結串列實作插入排序。
  - 輸入：第一個數字為 n，接下來有長度為 n 的數列。
  - 輸出：將數列由小到大輸出。
  - 範例輸入：5 3 7 9 6 2
  - 範例輸出：2 3 6 7 9

練習二 - 反轉鏈結串列

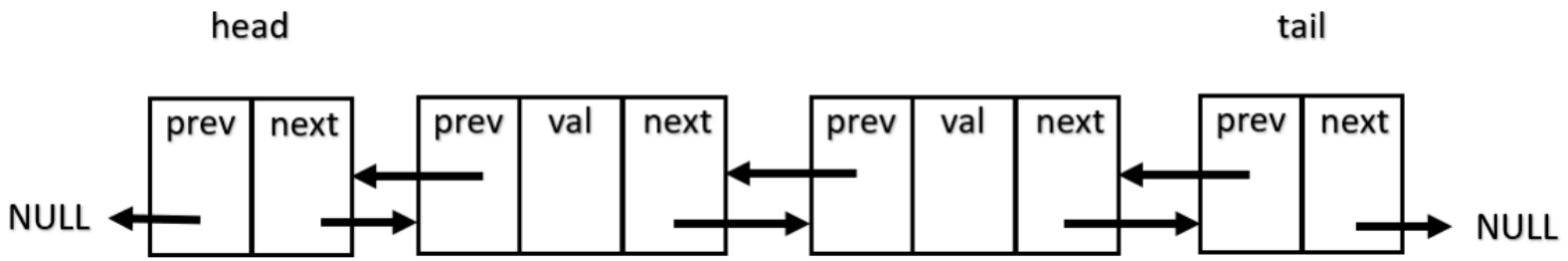
反轉鏈結串列 (Reverse A Linked List)

- 輸入一個數列，建立一個單向鏈結串列將其儲存，試著使用時間複雜度 $O(n)$ 的演算法將該串列反轉。
- 試著寫出此演算法的 pseudo code。

雙向鏈結串列
Doubly Linked List

雙向鏈結串列 (Doubly Linked List)

- 雙向鏈結串列是一種特殊的鏈結串列，每個節點都包含指向前一個節點和下一個節點的指標。它可以方便地實現反向遍歷和刪除操作，相比於單向鏈結串列，雙向鏈結串列的優勢在於：
  - 可以反向遍歷鏈結串列，使得在一些應用中更加方便。
  - 可以方便地從鏈結串列中刪除節點，而不需要遍歷整個鏈結串列來查找前一個節點。



雙向鏈結串列的結構

- 每個節點包含三個部分：資料、指向前一個節點的指標和指向下一個節點的指標。
- 節點的結構通常定義為：

```
1 typedef struct node {
2     int value;
3     struct node *prev;
4     struct node *next;
5 } Node;
```

雙向鏈結串列 (Doubly Linked List)

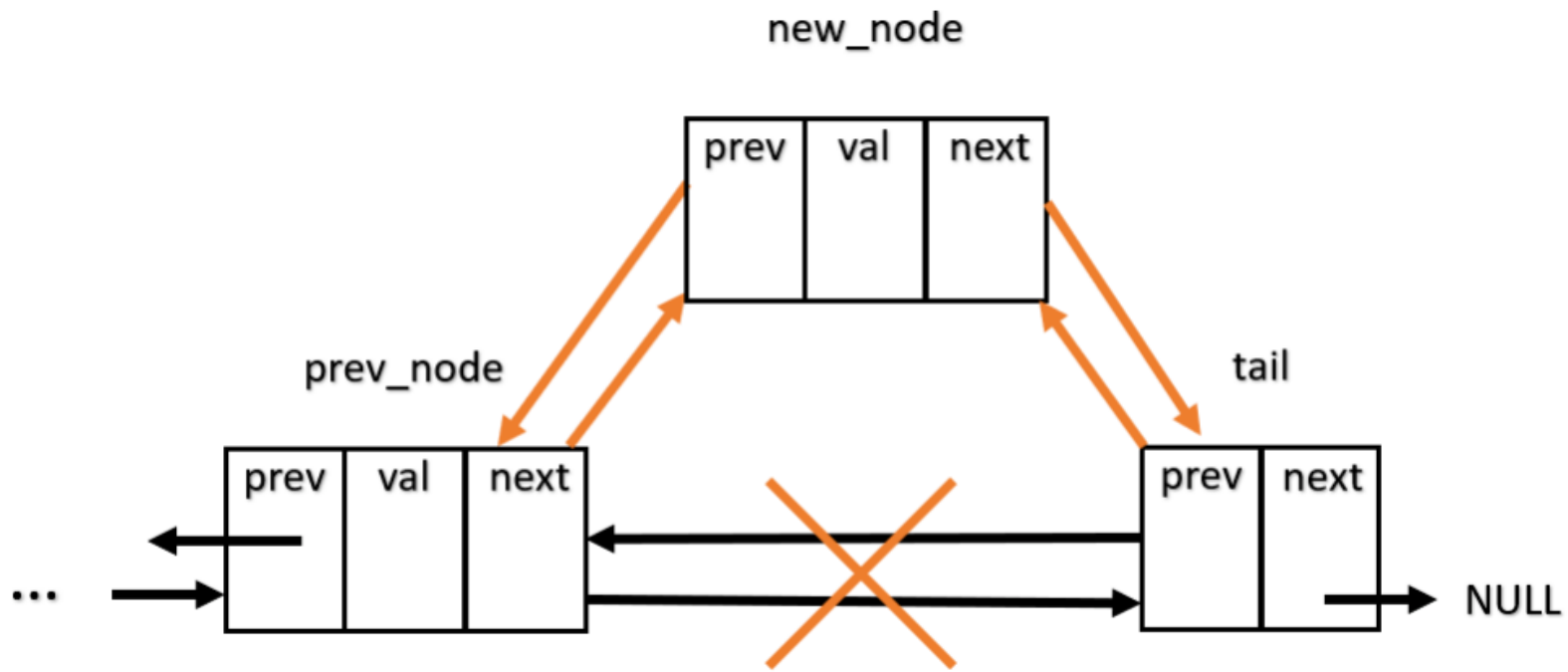
- 由於雙向鏈結串列向前向後都擁有一個指標，因此在做宣告的時候除了 head 以外，還需要維護 tail。

```
1  int main() {
2      // 分別為頭節點和尾節點分配記憶體
3      Node *head = (Node*)malloc(sizeof(Node));
4      Node *tail = (Node*)malloc(sizeof(Node));
5
6      head->prev = NULL;
7      head->next = tail;
8      tail->prev = head;
9      tail->next = NULL;
10
11 }
```

雙向鏈結串列 (Doubly Linked List)

將節點添加到末尾

- 方法: 由於我們維護了一個 tail 指標，因此我們可以透過 tail 往前直接找到最後一個節點的位置，並將新節點加在此節點之後。



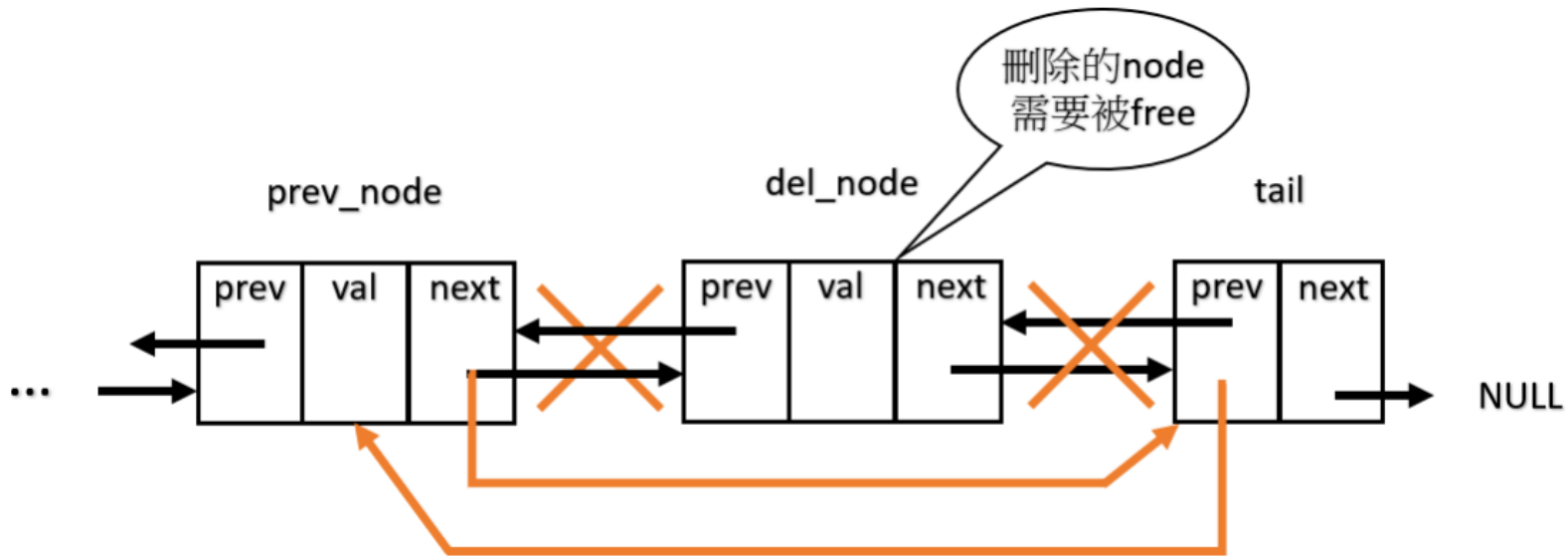
雙向鏈結串列 (Doubly Linked List)

- 將節點添加到末尾程式：

```
1  void push_back(Node *tail, int newValue) {
2      Node *prev_node = tail->prev; // 找到目前最後節點
3
4      // 分配新節點
5      Node *new_node = (Node*)malloc(sizeof(Node));
6      new_node->value = newValue;
7
8      // 連接起來
9      prev_node->next = new_node;
10     new_node->prev = prev_node;
11     new_node->next = tail;
12     tail->prev = new_node;
13     return;
14 }
```

刪除末尾元素

- 方法: 由於我們維護了一個 tail 指標，因此我們可以透過 tail 往前直接找到欲刪除節點的位置，並將其前面的節點與 tail 指標連接起來。



雙向鏈結串列 (Doubly Linked List)

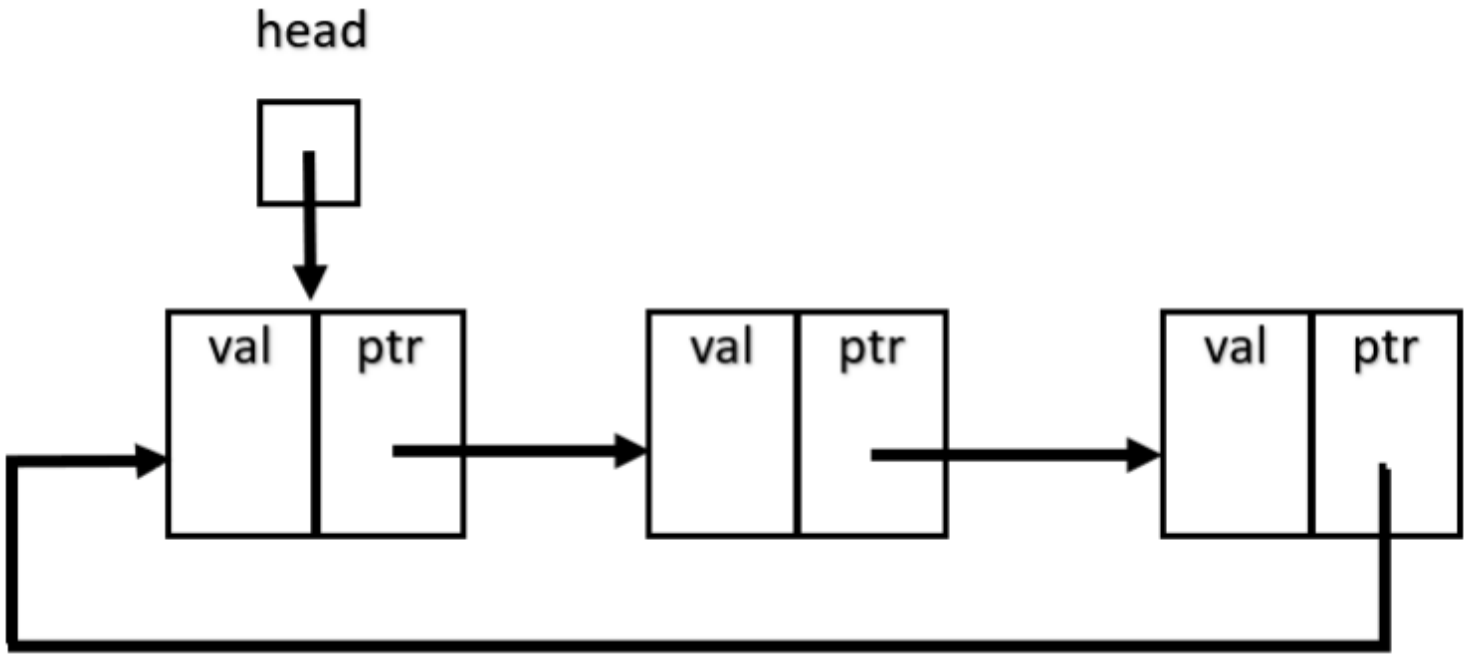
- 刪除末尾元素程式：

```
1  void pop_back(Node *head, Node *tail) {
2      if(tail->prev == head) {  // 如果鏈結串列為空的
3          printf("List is empty.\n");
4          return;
5      }
6
7      Node  *del_node = tail->prev;
8      Node  *prev_node = del_node->prev;
9
10     prev_node->next = tail;
11     tail->prev = prev_node;
12     free(del_node);
13     return;
14 }
```

環狀鏈結串列
Circular Linked List

環狀鏈結串列

- 環狀鏈結串列是一種基於鏈結串列的資料結構，其特點在於最後一個節點會指向第一個節點，形成一個環形的結構。
- 與前面所介紹的單向鏈結串列和雙向鏈結串列不同的是，環狀鏈結串列沒有結尾，因此可以無限地往後遍歷。



環狀鏈結串列 (Circular Linked List)

- 當我們在使用環狀鏈結串列時，由於它沒有結尾的特性，因此做迭代的時候要小心避面進入無限迴圈。
- 為了避免這種情況，我們需要設置一個條件來確定何時停止遍歷鏈結串列。通常當我們遍歷整個鏈結串列並返回到起點時，就可以停止遍歷。

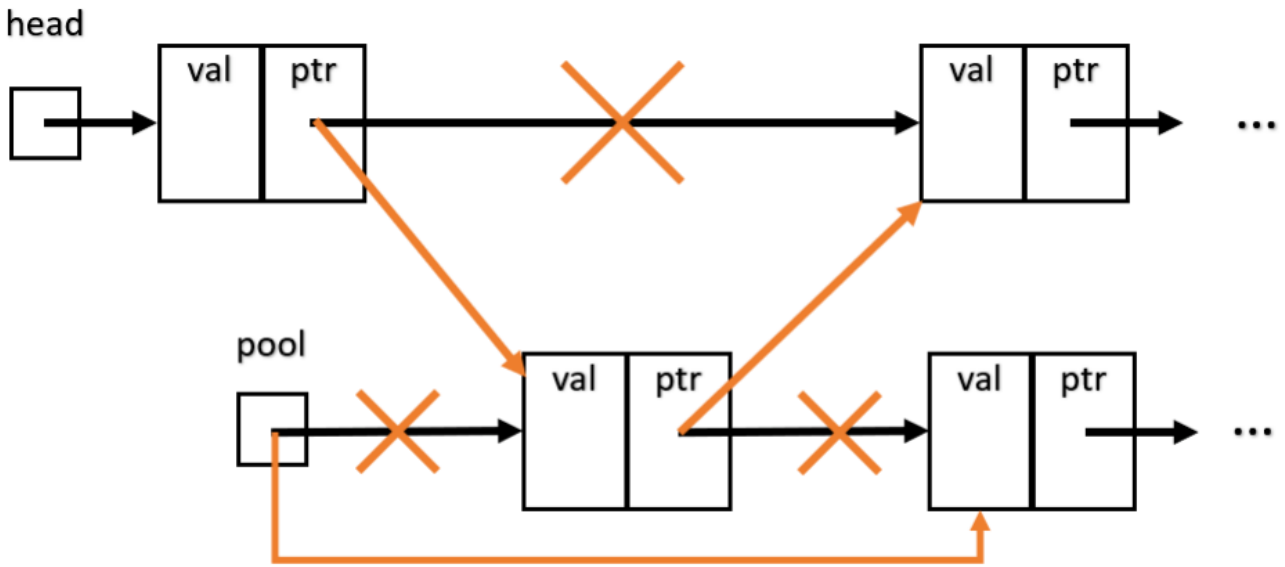
儲存池
Storage Pool

儲存池

- 鏈結串列儲存池的基本思路是先分配一定數量的節點，然後將這些節點連接起來，形成一個鏈結串列。
- 接著，將這個鏈結串列中的節點放入一個池中。當需要新節點時，直接從池中取出節點即可，而不需要進行記憶體的動態分配和釋放操作。
- 當節點不再使用時，也不釋放節點，而是將其放回池中，以供下一次使用。

儲存池 (Storage Pool)

- 以下是示意圖，當我們需要一些元素的時候，就可以直接從 pool 中取出我們所需的量，而不需要重新分配記憶體。
- 而刪除元素亦是如此，將元素丟入 pool 即可。
- 因此 pool 為可用節點 (Free Node) 之集合。
- 以 single link list 來管理可用節點，稱為 AV-list (Available list)。



儲存池 (Storage Pool)

- 鏈結串列儲存池的作用有以下幾個方面：
  - 提高記憶體分配效率：使用鏈結串列儲存池可以避免頻繁地進行動態記憶體分配和釋放，從而提高分配效率。
  - 避免記憶體碎片：由於鏈結串列儲存池會事先分配一定數量的元素，元素的大小相同且連續存儲，因此可以避免產生記憶體碎片。
  - 下一頁會說明記憶體碎片。

儲存池 (Storage Pool)

記憶體碎片

- 記憶體碎片是指電腦中存在一些未被使用的小塊記憶體空間，它們的大小通常比需要分配的記憶體空間小，無法滿足大記憶體需求。
- 這可能導致系統浪費記憶體，即使有足夠可用的記憶體，由於記憶體分佈不連續，也無法滿足大記憶體需求。
- 記憶體碎片通常出現在動態分配記憶體的情況下，例如使用動態分配記憶體時，由於記憶體分配和釋放的不規律性，就會出現碎片。
- 記憶體碎片會降低系統性能，因為系統需要頻繁搜索並重新分配可用空間，增加了運行時間和開銷。

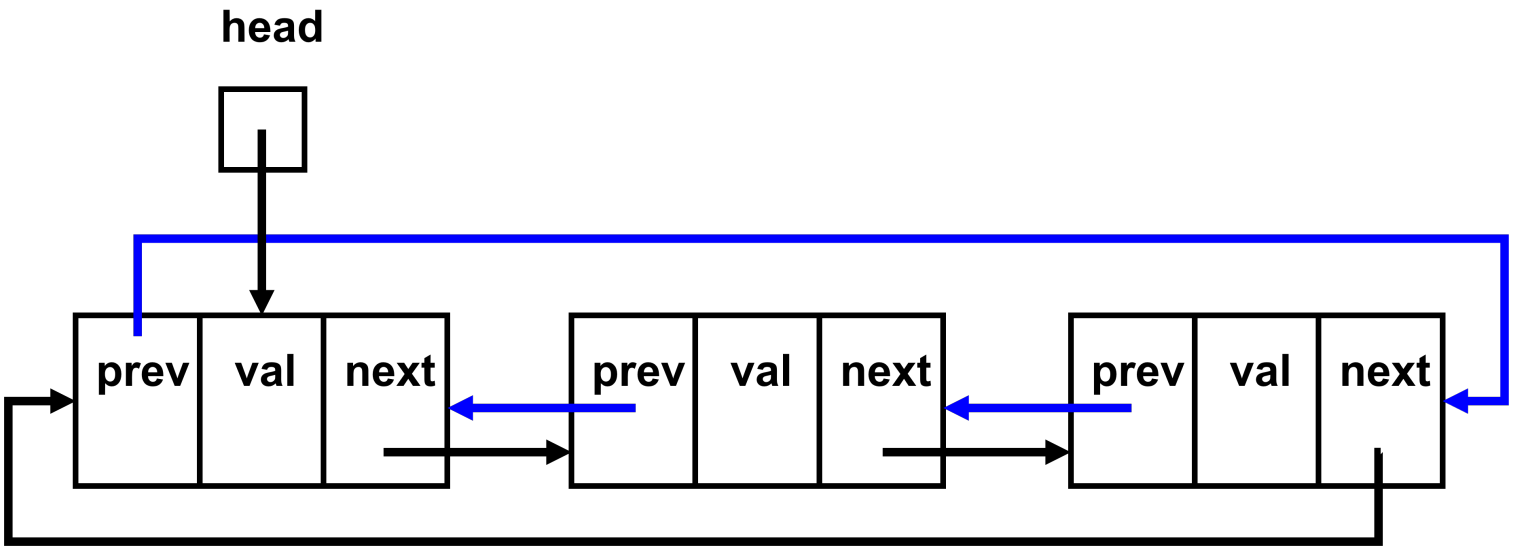
回收環狀鏈結串列

- 輸入一個數列，建立一個環狀鏈結串列將其儲存，試著使用時間複雜度 $O(1)$ 的演算法將該串列回收至儲存池/AV-list。
- 試著寫出此演算法的 pseudo code。

環狀雙向鏈結串列
Circular Doubly Linked List

環狀雙向鏈結串列

- 環狀雙向鏈結串列是雙向鏈結串列和循環鏈結串列的結合，其特點在於串列的每個節點是可以前後互相連通的，並且當遍歷整個串列後，最後會回到最源頭的那個節點。



實作環狀雙向鏈結串列

- 輸入一個數列，建立一個環狀雙向鏈結串列將其儲存。
- 試著寫出環狀雙向鏈結串列各種操作之演算法的 pseudo code，並以 c 語言程式碼實作練習。
  - 各種操作請參照此簡報的 p.20 和 p.21。

Q & A