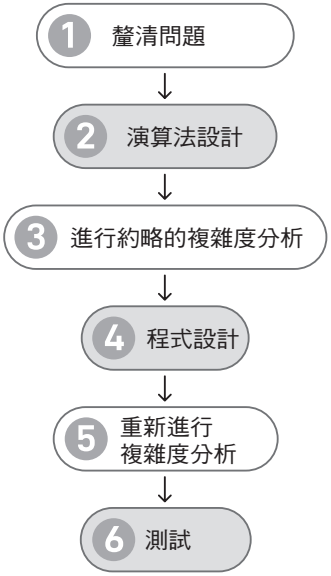


1. **釐清問題，並收集相關的限制條件。**
   - 用你自己的話來重新描述問題。這樣可以確保你與面試官確實達成了共識。
   - 提出一些可釐清疑問的問題，收集所有必要的限制條件與極端的情況。
2. **設計出你的演算法。**
   - 一開始先採用暴力解法或比較簡單的解法，是很有用的一種做法。簡單的解法總好過心裡沒有任何的解法。有時透過暴力解法或比較簡單的解法，就可以衍生出更優化的解法。如果解法實在很簡單而直接，只要用口頭描述也就足夠了，並不一定非要用程式碼寫出來。
   - 多考慮、多討論幾種可能的解法。你可以針對這幾個選項進行討論，再決定哪一個最適合，並對你的選擇做出解釋。
   - 如果演算法很複雜，請考慮使用虛擬程式碼。這樣就可以讓你的策略更加清楚，而且可以在編寫實際的程式碼時，用它來作為高階的指南。

3. **針對時間和空間複雜度，提供約略的估計。**
    - 針對你的做法進行初步的複雜度分析，以證明其有效性，並與面試官進行一些討論。
    - 如果是比較複雜的演算法，在這個階段只進行約略的估計，通常是可以接受的。一旦真正寫出程式碼之後，就可以再進行更徹底的分析了。
4. **寫出乾淨、易讀的程式碼。**
    - 盡量使用有意義的變數名稱。
    - 善用輔助函式來幫你建構出你的程式碼，並且避免寫出重複的程式碼。
    - 針對特定的物件，設計出相應的物件類別，這樣就可以提高可讀性（例如用一個 Interval 物件類別來表示區間，其中包含 start（開始）和 end（結束）等屬性）。
5. **重新分析複雜度。**
    - 清楚說明你的程式碼相應的時間與空間複雜度。
    - 針對目前與之前所進行的複雜性分析，特別說明一下兩者之間的差別。解釋一下為什麼會有這樣的差異。
6. **測試你的程式碼。**
    - 考慮使用合理數量的測試案例，來確保你程式碼的正確性。
    - 測試簡單的案例：先從簡單的測試案例開始，以確保程式碼的基本功能是正常的。
    - 測試極端的案例：接著再考慮一些可能會讓程式碼掛掉的異常輸入或比較極端的輸入。
    - 測試一些邊界值：確保程式碼可以正確處理輸入範圍的下限與上限（例如，如果輸入的整數範圍介於 0 到 2^{31} - 1 之間，那就要特別針對 0 和 2^{31} - 1 進行測試）。
    - 利用測試的機會來識別出程式碼裡的任何錯誤，並嘗試進行修正。

在面試過程中進行有效溝通

「溝通」可說是程式設計面試過程中非常關鍵的一個要素。你會不會在必要時詢問相關的需求，釐清實際的狀況，還是你只知道一頭鑽進程式碼？請務必記住，程式設計面試應該是一場對話，所以請不要猶豫，隨時都可以提問。下面就是關於有效溝通如何提高成功機會的一些提示：

- **想到什麼就大聲說出來（Think out loud）**。你可以與面試官一起檢視你的思考過程。他們通常比較希望可以瞭解你如何利用觀察結果，並從各個角度解決問題，而不是沒有任何討論就得出最佳的解法。
- **確保面試官有跟上你的腳步**。這點很關鍵，因為你如果走錯了路，這樣你才有機會去釐清他們對你的做法所產生的困惑，進而糾正其中的誤解，並修正你的做法其中任何的問題。
- **討論其他替代做法**。如果你在解決問題的過程中，心裡出現好幾種不同的做法（例如要在不同的資料結構之間進行選擇），請讓面試官知道你所面臨的不同選擇。你們可以討論一下各種方法的優缺點，然後再針對你最終的選擇做出解釋。
- **擺脫困境**。如果你發現自己陷入了困境，請解釋一下你目前的思考來到了什麼樣的階段、你考慮過哪些選擇、為什麼你認為這條路走不通。這樣的做法可以讓面試官更瞭解你的情況，並提供一些可能的指導建議。
- **思考的時間**。有時候，花一點時間靜靜思考問題，也是很有用的做法。如果你需要花一點時間思考，請務必先告知面試官。這樣就可以表明你還是在持續處理與解決問題，而不是陷入了困境，或是不知道該如何表達。

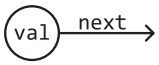
除了這裡所提供的一些面試小提示之外，你在接下來的章節中還會看到更多的內容。綜合以上所述，接下來我們就要直接深入探討如何掌握程式設計的各種模式，讓你的面試表現更上一層樓！

3

鏈結列表（Linked List）

鏈結列表簡介

鏈結列表是由一系列節點所組成的一種資料結構，其中每個節點都鏈結到下一個節點。鏈結列表裡的節點都有兩個主要的組成元素：它所保存的資料（val）以及指向序列中下一個節點的一個參照（next）：

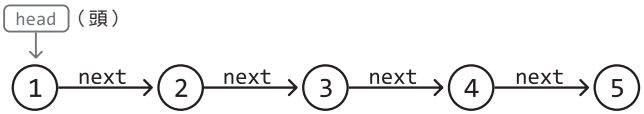


我們會用 ListNode 這個物件類別來定義一個節點，程式碼如下：

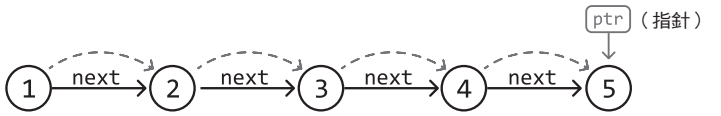
```
class ListNode:
   def __init__(self, val: int, next: ListNode):
       self.val = val
       self.next = next
```

單向鏈結列表（singly linked list）

鏈結列表最簡單的形式，就是單向鏈結列表，其中每一個節點都指向鏈結列表的下一個節點，而最後一個節點則指向 null，表示來到了鏈結列表的結尾。鏈結列表的開頭就叫做「頭」（head），它通常就是我們一開始可以直接存取到的唯一節點：

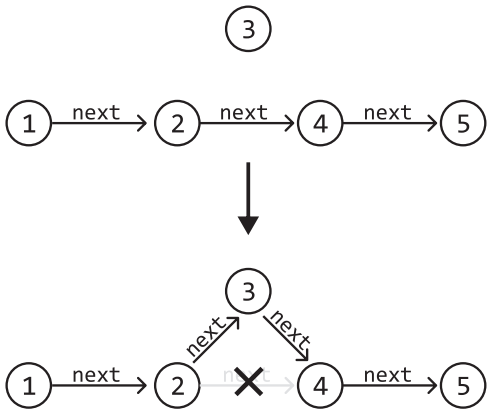


如果想要存取到鏈結列表裡其他的節點，我們就要從「頭」開始進行遍歷。



單向鏈結列表可用來保存一整組的資料。它最主要的優點之一就是可以**動態調整大小**，因為它可以很靈活地增大或縮小相應的大小，而不會像陣列只能是固定的大小。此外，單向鏈結列表在**需要頻繁進行插入和刪除**的情境下也有很出色的表現，因為它在執行這些操作時比陣列更有效率；陣列在執行插入或刪除的操作時，還必須移動元素的位置。

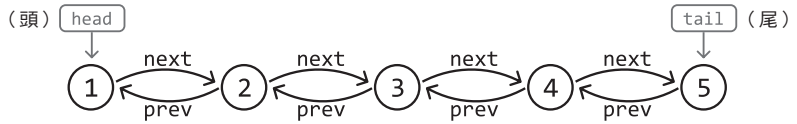
範例——把節點 3 插入到節點 2 和節點 4 之間



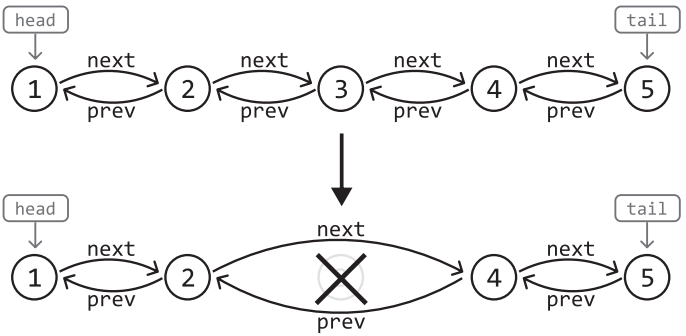
這些操作的高效率，是以**無法執行隨機存取**作為其代價，因為鏈結列表的節點並不能像陣列裡的元素那樣透過索引來進行存取。在許多使用情境下，這樣的權衡取捨或許是可接受的，因為有時候可動態調整大小的優勢和插入 / 刪除的高效率，或許比隨機存取的效能優勢更加重要。

雙向鏈結列表

雙向鏈結列表其實是鏈結列表的延伸版本；每個節點都有兩個參照（reference）：其中一個指向下一個節點（next），另一個指向上一個節點（prev）。在大多數的實作程式碼中，雙向鏈結列表都可以直接存取「頭」（head）節點和「尾」（tail）節點。

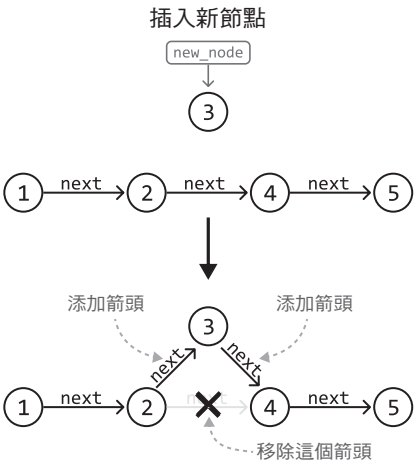


雙向鏈結列表的一大優點，就是它可以進行**雙向遍歷**。此外，如果要刪除掉雙向鏈結列表裡的某個節點，做法上通常會更直接，因為我們同時擁有指向下一個節點和指向上一個節點的參照：



指針操作

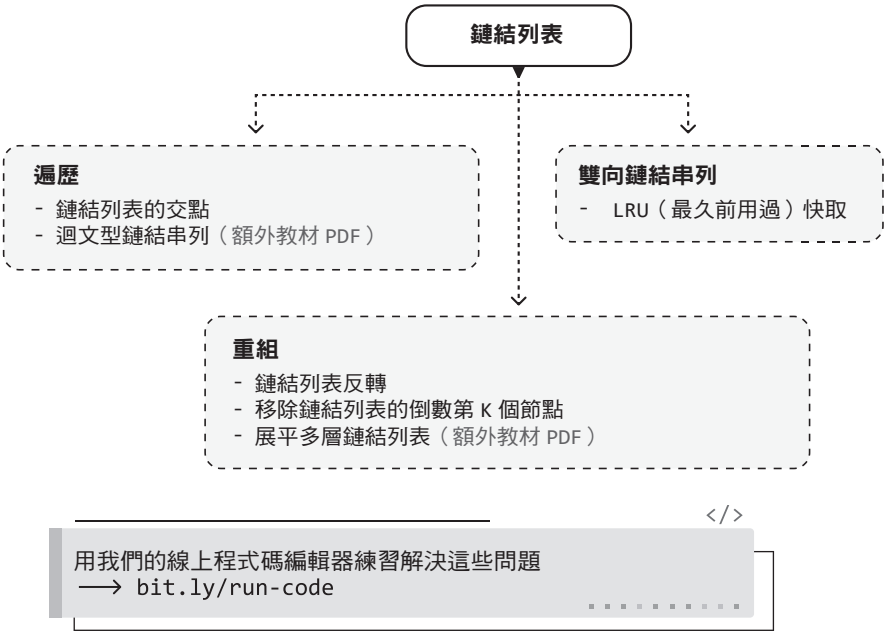
許多鏈結列表相關的面試問題，都會對鏈結列表進行遍歷或重組。理解並熟練指針相關操作，對於解決這些問題來說至關重要。其中一個很有用的技巧，就是把指針視覺化成從一個節點指向另一個節點的箭頭，然後再觀察這些箭頭應該如何移動，才能夠反映出結構上的改變。舉例來說，我們可以用下面的視覺化方式來呈現節點的插入操作：



真實的範例

音樂播放清單：音樂播放器這類的應用通常都是用鏈結列表（尤其是雙向鏈結列表）來實作出播放清單的功能，其中每一個歌曲節點都會鏈結到下一首和上一首歌曲。這樣的結構就可以用很有效率的方式添加、刪除歌曲或重新進行排序，因為只要更新節點之間的指針即可，而不需要去移動記憶體裡的歌曲資料。

本章綱要



如果想取得**這份額外提供的 PDF 教材**，請使用以下鏈結訂閱程式設計模式面試指南的最新資訊：

bit.ly/coding-patterns-pdf

實作程式碼——遞迴的解法

```
def linked_list_reversal_recursive(head:  ListNode) -> ListNode:
    # 基本情況。
    if (not head) or (not head.next):
        return head
    # 以遞迴的方式，去反轉那個以下一個節點為起頭的子列表。
    new_head = linked_list_reversal_recursive(head.next)
    # 把反轉後的鏈結列表與 head 節點相連起來，
    # 完成整個鏈結列表的反轉。
    head.next.next = head
    head.next = None
    return new_head
```

複雜度分析

時間複雜度：遞迴做法的時間複雜度為 $O(n)$，因為它會以遞迴的方式把整個鏈結列表遍歷過一次，而且每個節點都只會被造訪過一次。

空間複雜度：空間複雜度為 $O(n)$，因為遞迴調用堆疊需要佔用一些空間，而這裡應該會有 n 次的遞迴調用，所以堆疊所佔用的空間最多有可能會增長到 n 的程度。

面試小提示

提示：用視覺化方式來呈現指針操作

一般來說，在處理鏈結列表的操作時，光是要搞清楚到底應該做什麼，可能就是一件很棘手的事。用箭頭的形式畫出在節點之間的指針，通常還蠻有幫助的。只要觀察這些箭頭應該重新指向哪裡，以呈現出鏈結列表結構的變化，這樣就可以讓我們推斷出一些必要的指針操作邏輯。這樣的做法也可以讓我們在進行改變時，協助我們識別出需要把參照指向哪幾個節點。