C/C++ 進階班
資料結構

鏈結串列
(Linked List)

李耕銘

課程大綱

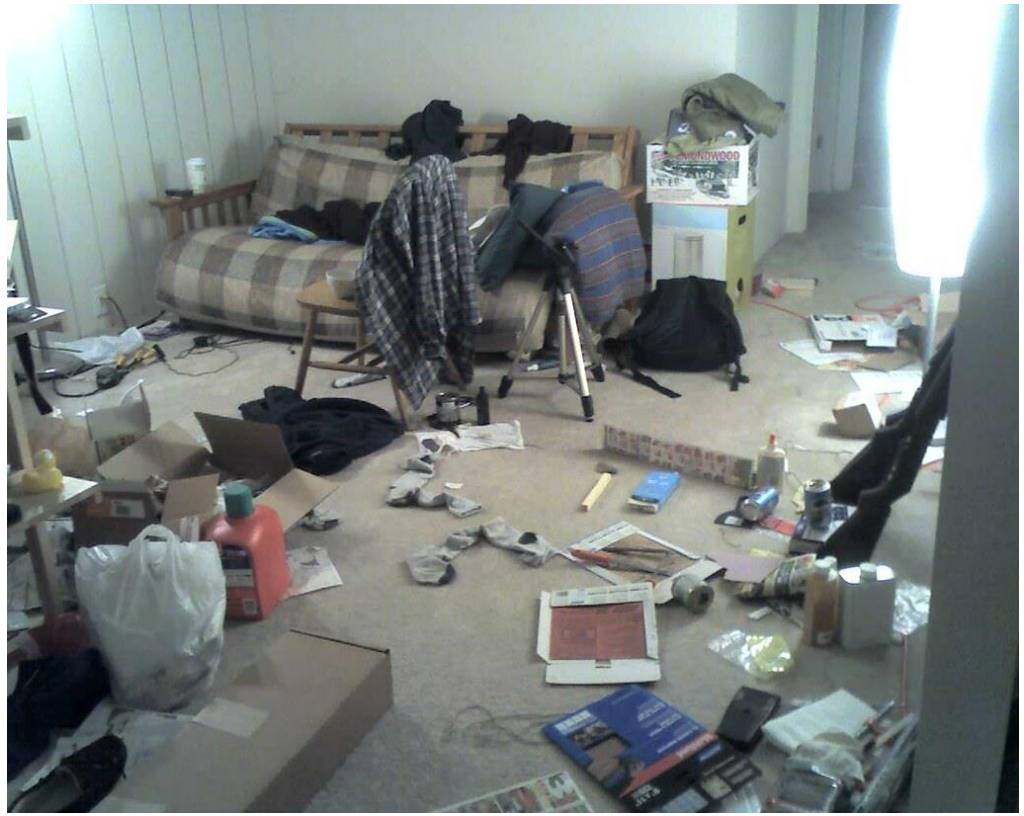
- 鏈結串列(Linked List)簡介
- 鏈結串列的基本操作
- 不同種 Linked List
- Linked List @ C++ STL

鏈結串列(Linked List)簡介

李耕銘

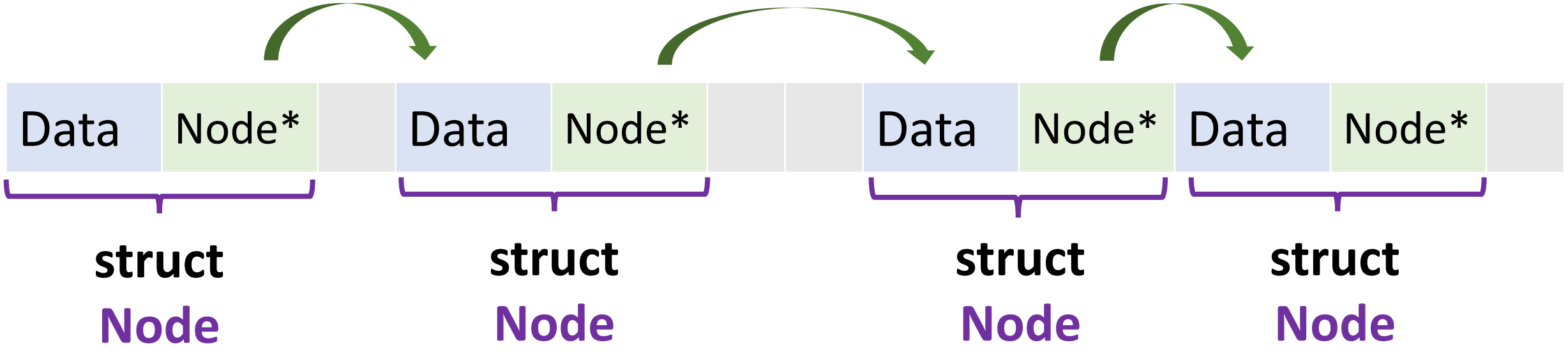
鏈結串列(Linked List)

- 陣列與 Vector 有連續的記憶體空間
  - 難以擴充、擴充耗時耗空間
  - 每次擴大大小，都可能伴隨資料遷移
  - 通常無法確定需要的大小阿 QQ
- 不要把資料連續放著，新增就方便了
  - 想想你的房間……
  - Linked List

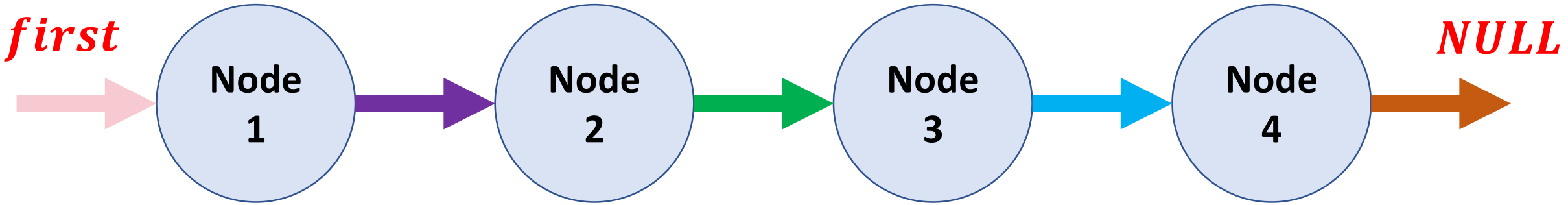


鏈結串列(Linked List)

- 鏈結串列(Linked List)
  - ➢ 每需要插入一筆新資料，放在記憶體中任意位置
    - ✓ 插入/刪除資料時，不需修改其他資料的位置
  - ➢ 利用指標紀錄資料的記憶體位置並串聯起來



鏈結串列(Linked List)



Data	A
Pointer	0x02
Address	0x01

Data	B
Pointer	0x03
Address	0x02

Data	C
Pointer	0x04
Address	0x03

Data	D
Pointer	0
Address	0x04

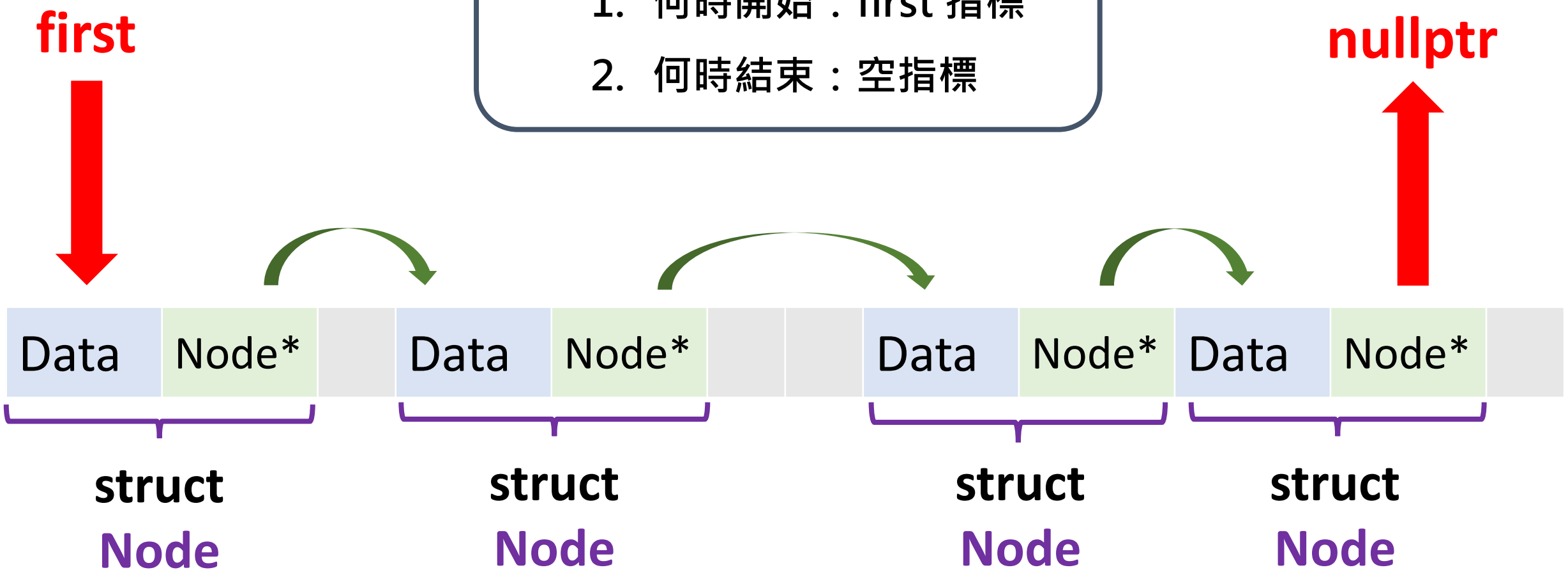
利用指標紀錄下一個節點/元素的位置

插入/刪除新的資料只需要O(1)

鏈結串列(Linked List)

但需要另外註記

1. 何時開始：first 指標
2. 何時結束：空指標

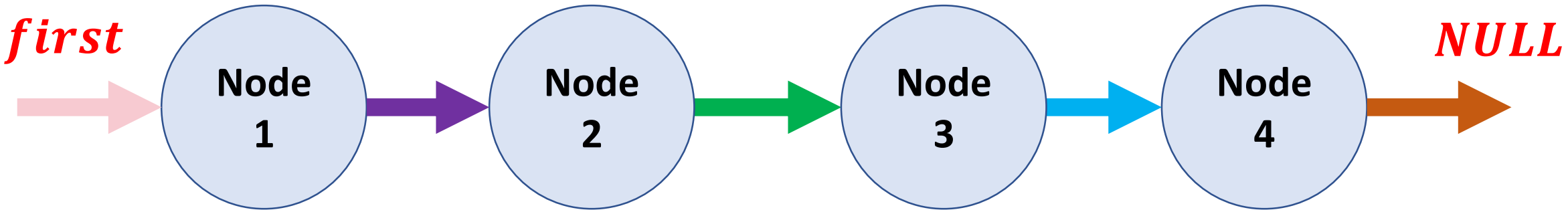


鏈結串列的效能分析

- Pros
  - 新增/刪除資料較陣列簡單
    - ✓ 只需要改寫 O(1) 個節點的資料內容
  - 若是新增開頭的資料，只要O(1)。
  - 資料數量是彈性的，不像陣列時常需要搬動資料
- Cons
  - 要刪除/新增特定位置的資料，需先執行O(N)的搜尋
  - 不支援索引值取值
    - ✓ 搜尋時必須從頭開始依序尋找，需要O(N)
  - 相較陣列，需要額外的空間來儲存指標與記憶體位置

鏈結串列的效能分析

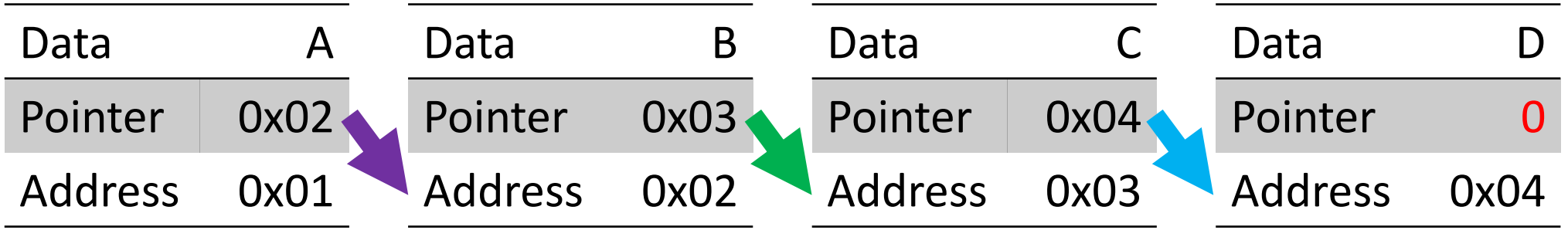
- 鏈結串列使用時機
  - 資料的總數量是浮動的，每次更動都不需要移動資料
  - 新增/刪除資料的操作頻繁
  - 很少查詢或走訪所有資料的需求



鏈結串列的架構

- Node 結構
  - Data：存放資料
  - Pointer：指向另一個 Node

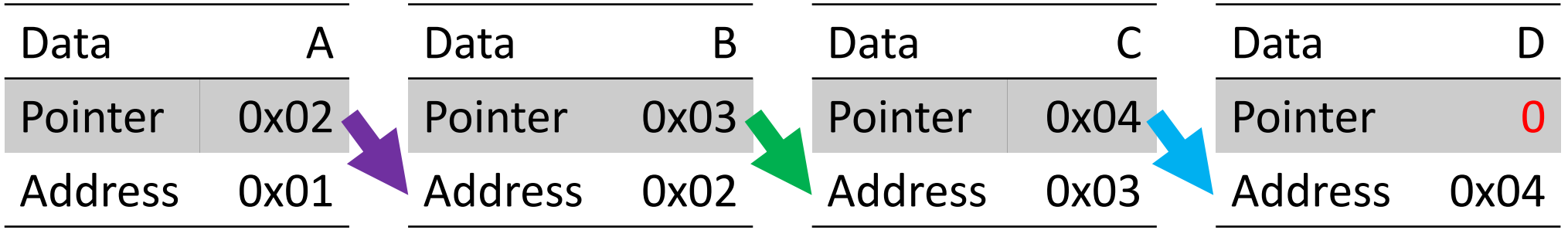
```
template <typename T>
struct Node {
        T Data;
        Node *Next;
};
```



鏈結串列的架構

- Linked List 類別
  - First：指向第一個 Node
  - 各種操作函式：
    - 走訪
    - 搜尋
    - 新增
    - 刪除
    - 反轉……

```
template <typename T>
class Linked_List{
    private:
        Node<T>* First;
    public:
        Linked_List();
        void Print_List();
        int Search_List(T);
};
```



C 語法

- 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
- free(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
- 指標 = 0;
  - ➢ 讓指標未指到任一個實體

C++語法

- 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
- 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
- 指標 = nullptr;
  - ➢ 讓指標未指到任一個實體，C++ 11後支援

C/C++語法

- malloc 與 new 的差異
  - new 會創造出物件實體，並呼叫建構式
  - malloc 僅配置記憶體空間不會呼叫建構式

Example Code

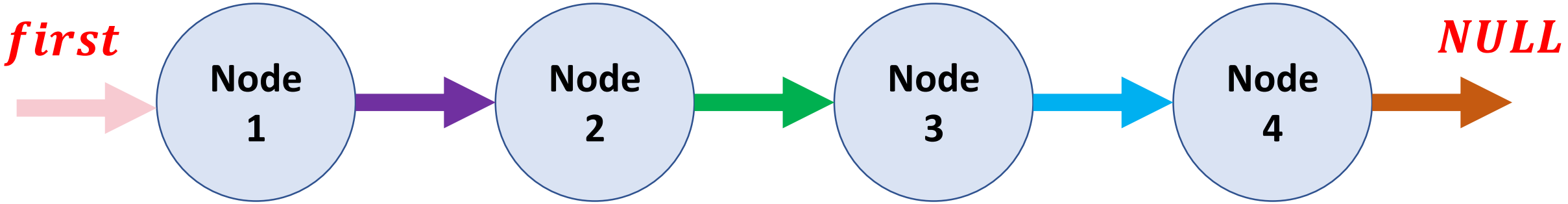
Mission

建立 Node 結構、Linked List 類別，並完成建構式

- Node結構：
  - Data
  - Pointer：指向另一個 Node

- Linked List 類別：
  - First：指向第一個 Node
  - 各種操作函式：
    - 走訪
    - 搜尋
    - 新增
    - 刪除
    - 反轉……

鏈結串列的走訪



Data	A
Pointer	0x02
Address	0x01

Data	B
Pointer	0x03
Address	0x02

Data	C
Pointer	0x04
Address	0x03

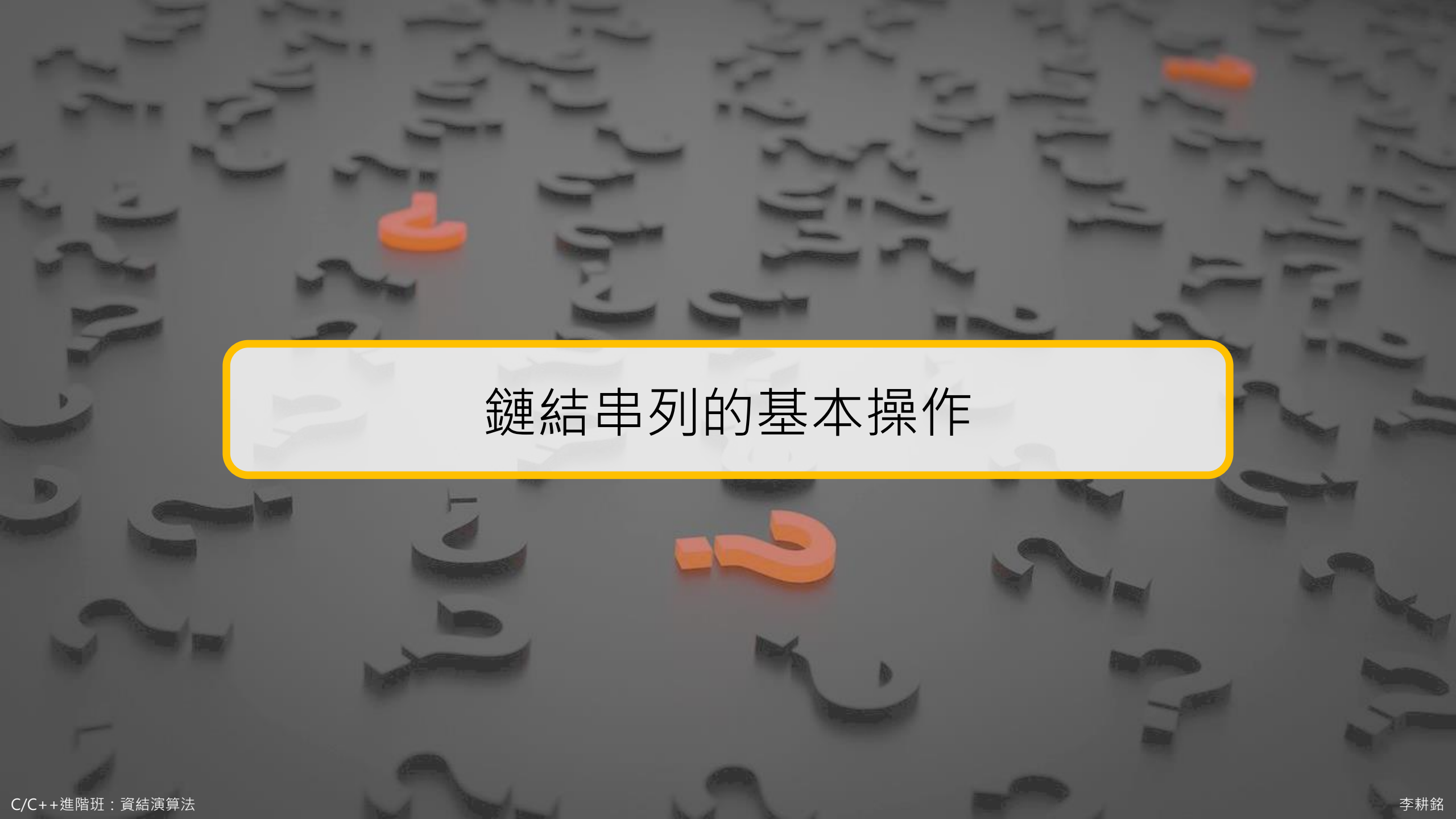
Data	D
Pointer	0
Address	0x04

Practice

Mission

- 完成下列 Linked List 內的函式：
  - Print_List()
    - ✓ 印出所有 Linked List 內的資料 (Data)
  - Search_List(T)
    - ✓ 在 Linked List 內找尋特定資料
    - ✓ 回傳第幾筆資料
    - ✓ 只需要回傳第一個找到的即可
  - 直接從剛剛的程式碼來改寫。

```
template <typename T>
class Linked_List{
    private:
        Node<T>* First;
    public:
        Linked_List();
        void Print_List();
        int Search_List(T);
};
```

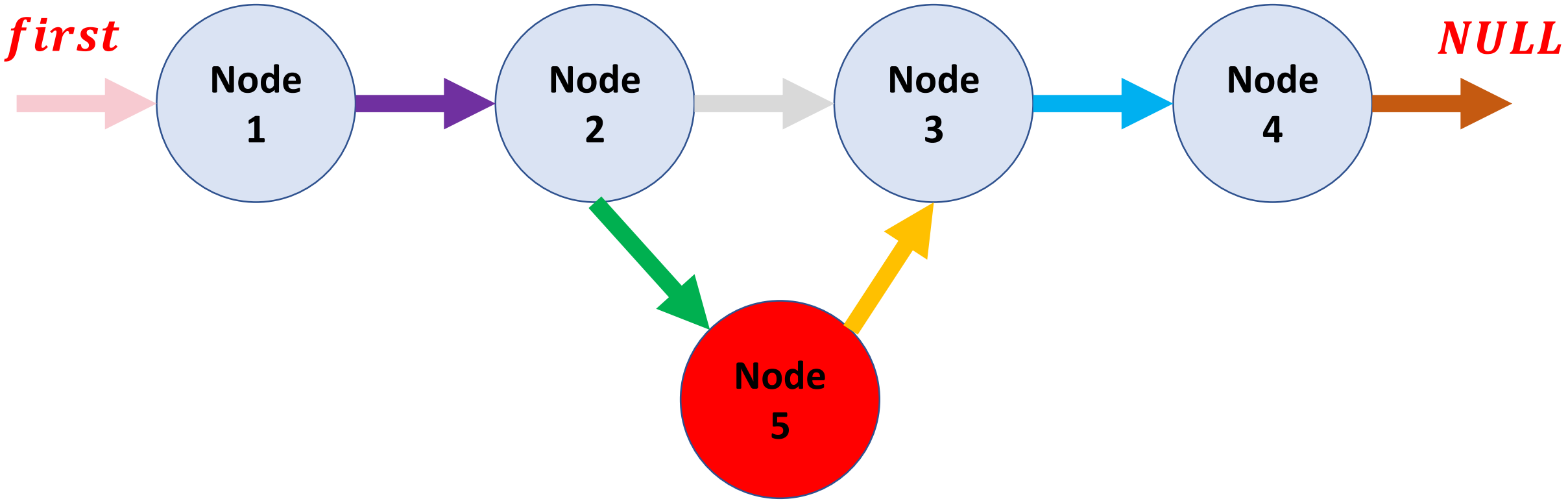


鏈結串列的基本操作

鏈結串列的基本操作

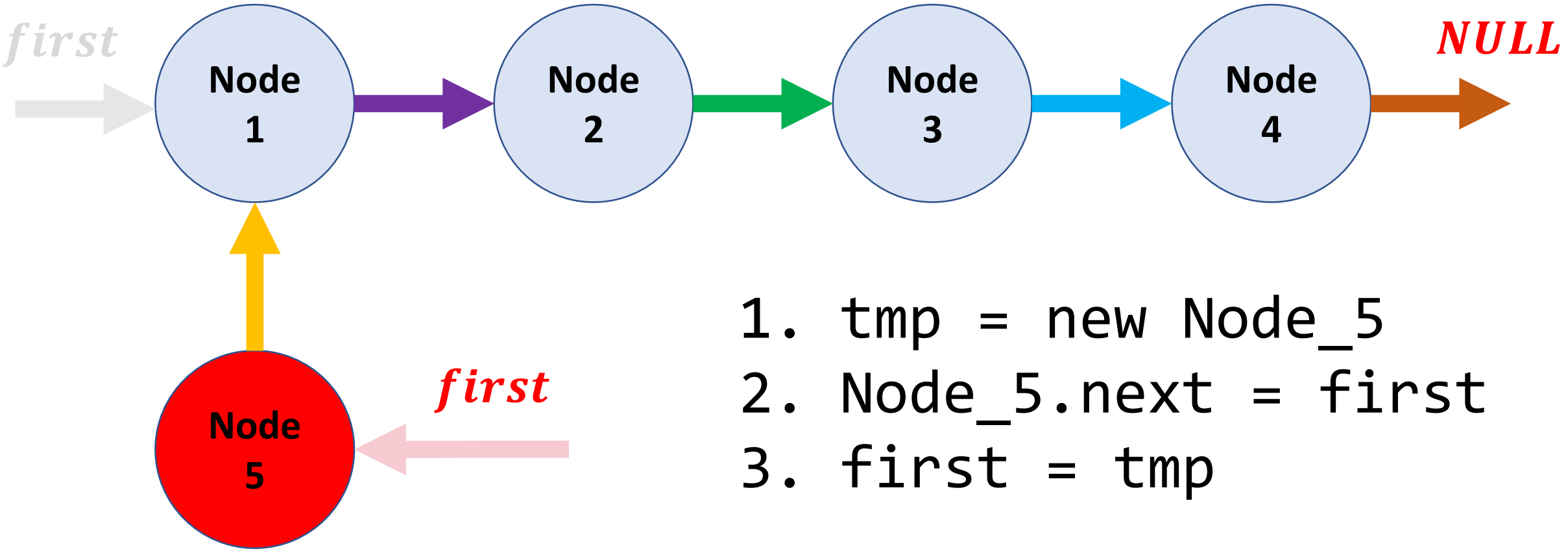
- 新增/插入
  - 在特定位置新增資料
  - 在首項新增資料
  - 在尾端新增資料
- 刪除
  - 在特定位置刪除資料
  - 在首項刪除資料
  - 在尾端刪除資料
- 反轉鏈結串列
- 清空所有資料

新增元素至特定位置




```
1.   tmp = new Node_5
2.   Node_5.next = Node_2.next
3.   Node_2.next = tmp
```

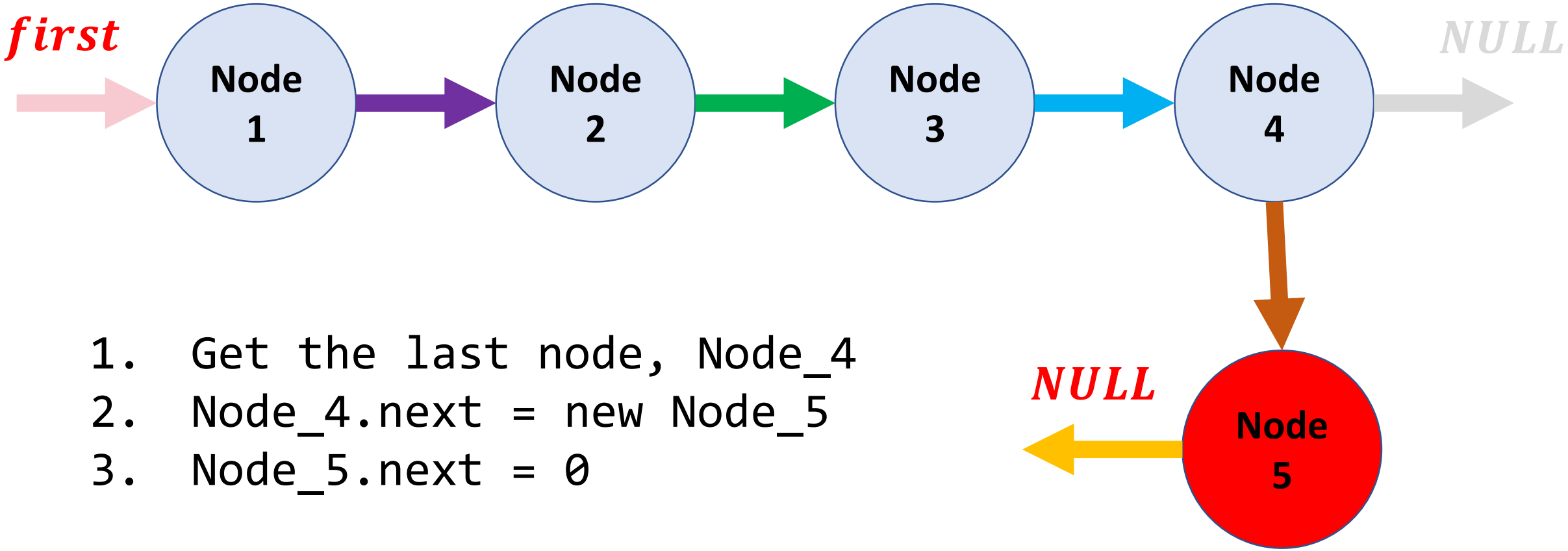
新增頭項




```
1. tmp = new Node_5
2. Node_5.next = first
3. first = tmp
```

不可用剛剛的新增函式！

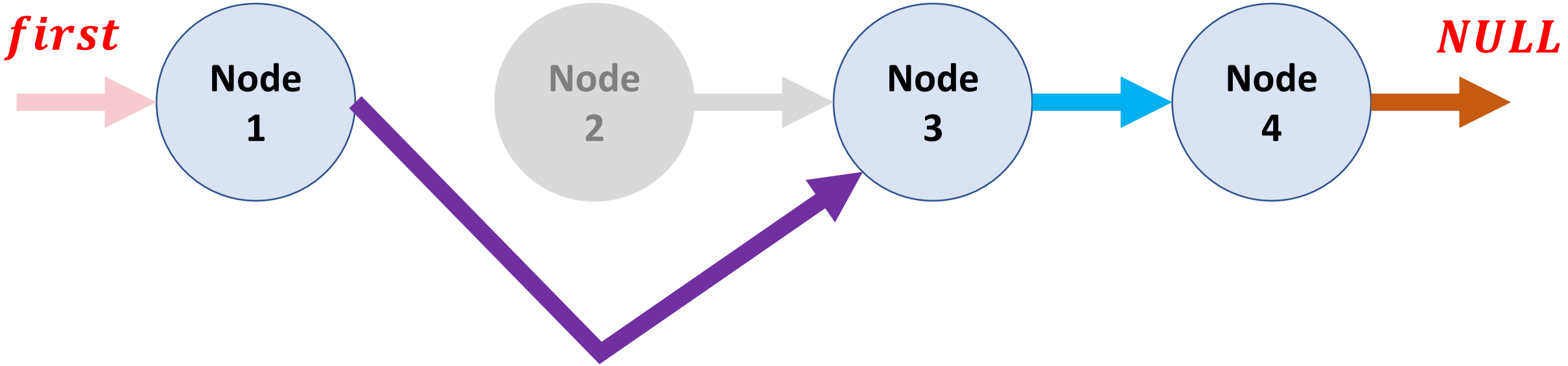
新增尾項




```
1.  Get the last node, Node_4
2.  Node_4.next = new Node_5
3.  Node_5.next = 0
```

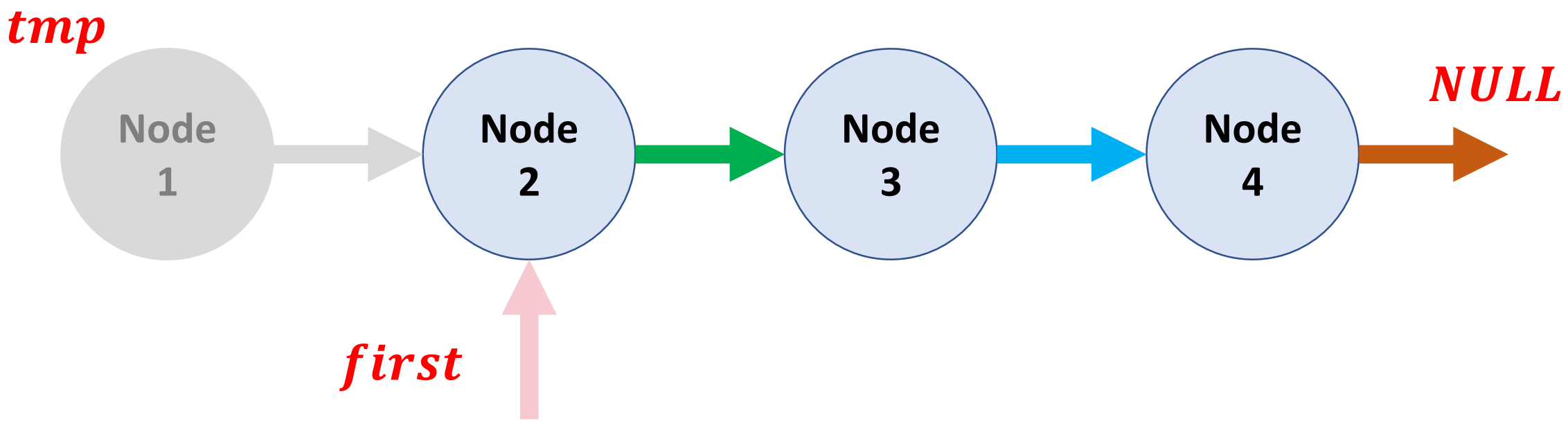
可用剛剛的新增函式！

刪除特定位置上的元素




```
1.  tmp = Node_2.next
2.  delete Node_1.next
3.  Node_1.next = tmp
```

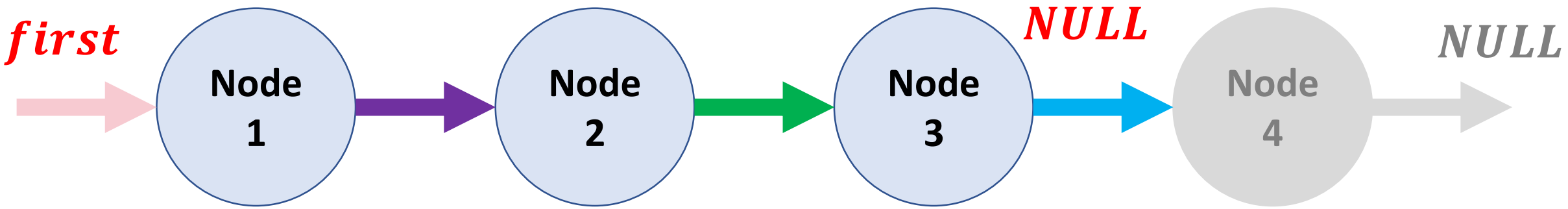
刪除頭項




```
1.  tmp = first
2.  first = first.next
3.  delete tmp
```

不可用剛剛的刪除函式！

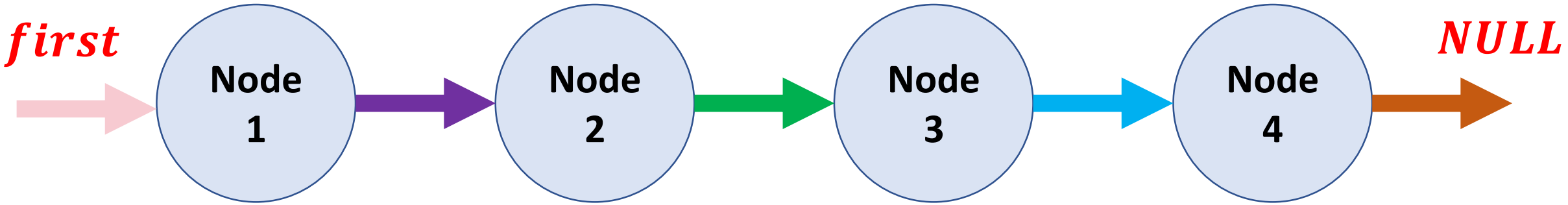
刪除尾項




```
1.  Get Node_3
2.  delete Node_3.next
3.  Node_3.next = 0
```

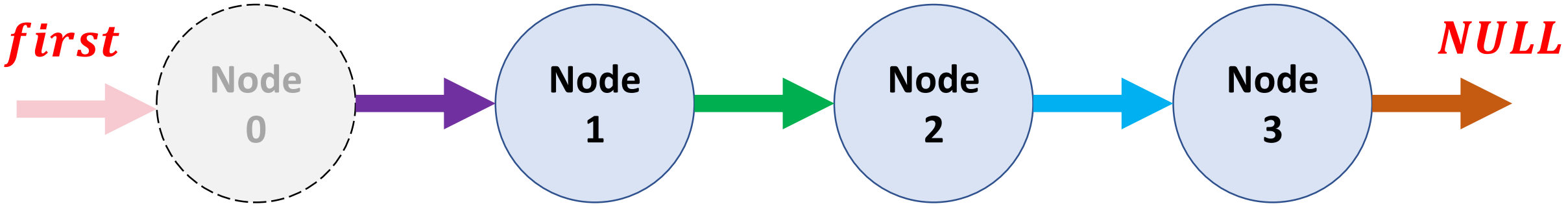
可用剛剛的刪除函式！

鏈結串列的基本操作



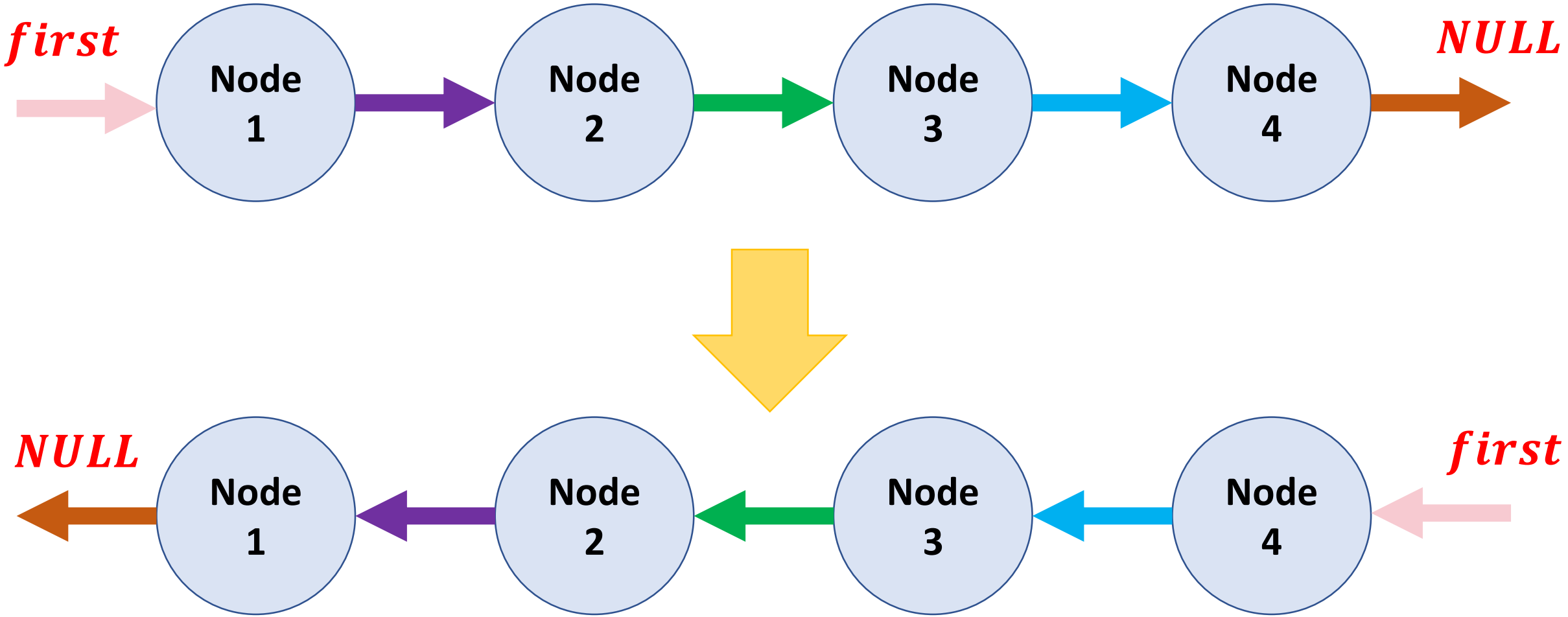
- 尾項新增/刪除
  - 可以套用通用的新增/刪除函式
- 頭項新增/刪除
  - 不可套用通用的新增/刪除函式
- Dummy head
  - 在頭項新增一個空的 node
  - 頭端的新增/刪除可以套用通用的函式

鏈結串列的基本操作

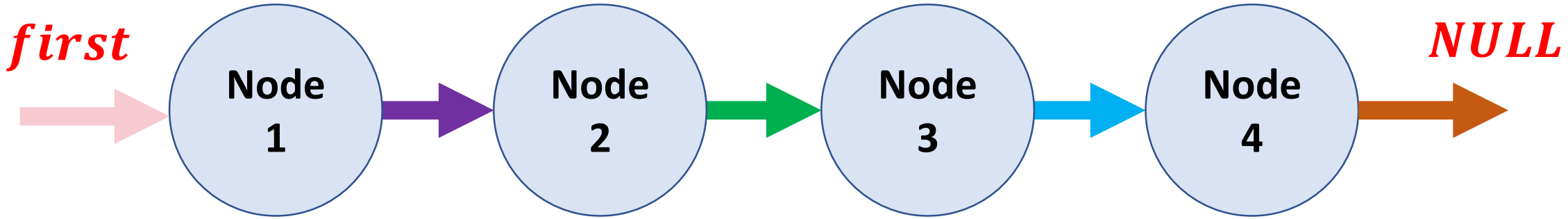


- 尾項新增/刪除
  - 可以套用通用的新增/刪除函式
- 頭項新增/刪除
  - 不可套用通用的新增/刪除函式
- Dummy head (node 0)
  - 在頭項新增一個空的 node
  - 頭端的新增/刪除可以套用通用的函式

反轉


first
Node 1
Node 2
Node 3
Node 4
NULL
NULL
Node 1
Node 2
Node 3
Node 4
first

反轉

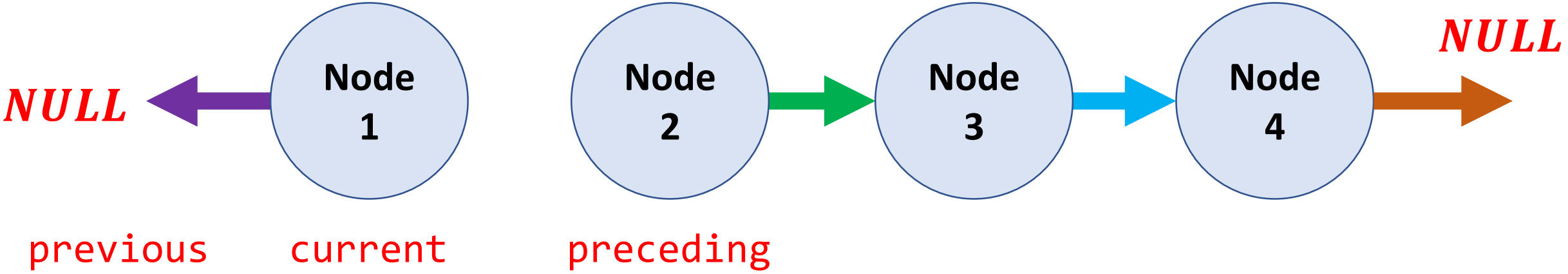


previous current preceding

```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

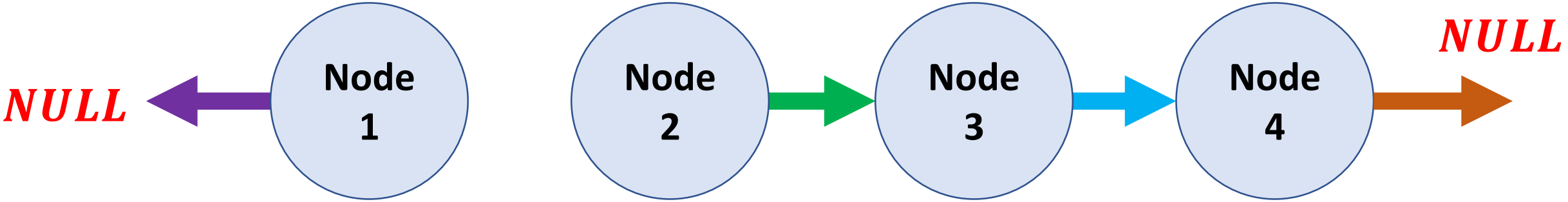
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

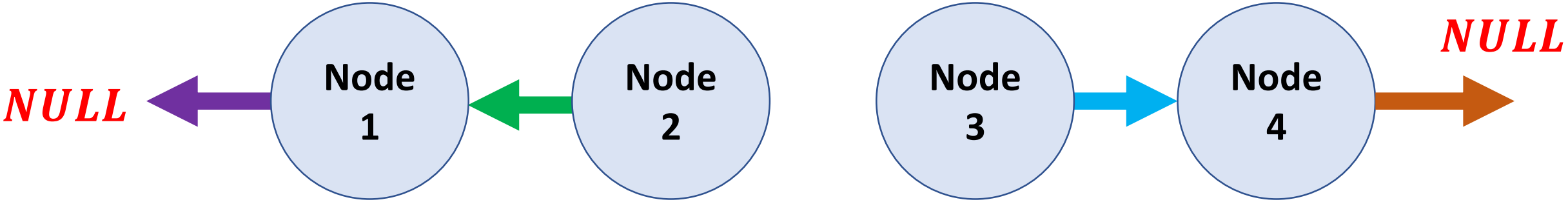
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

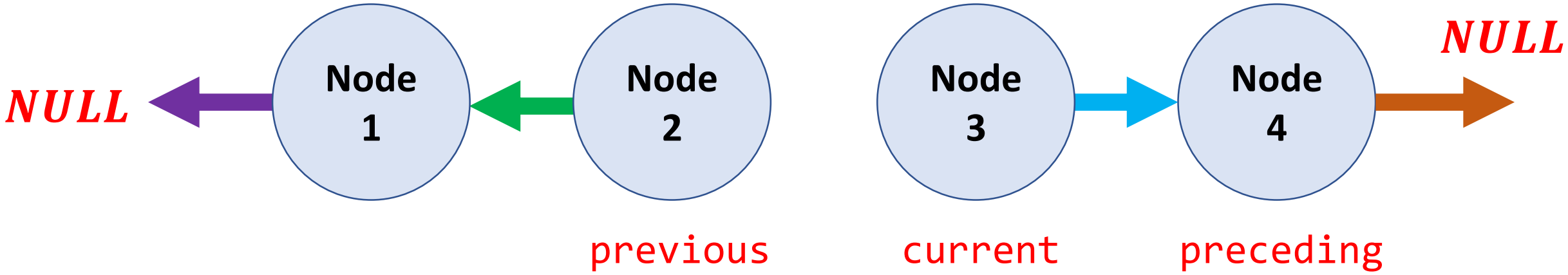
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

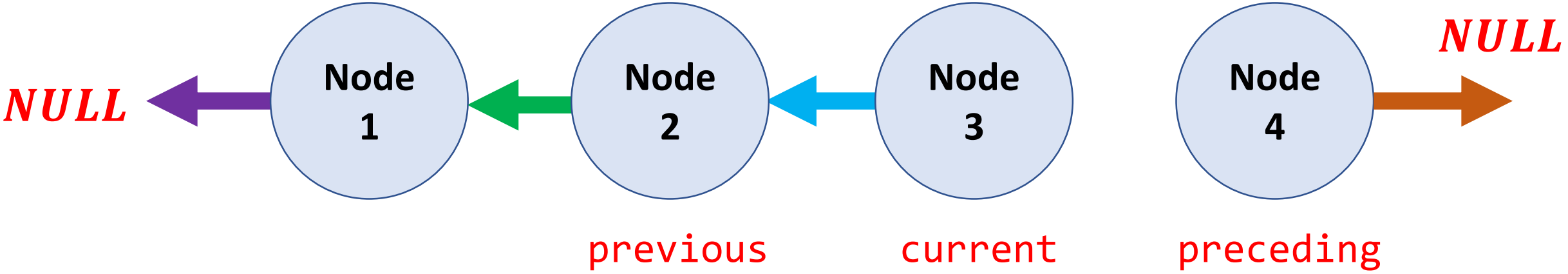
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

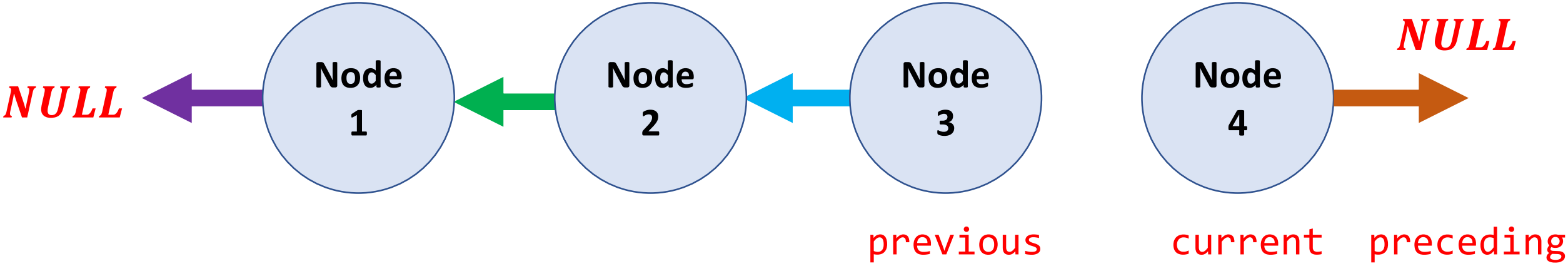
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

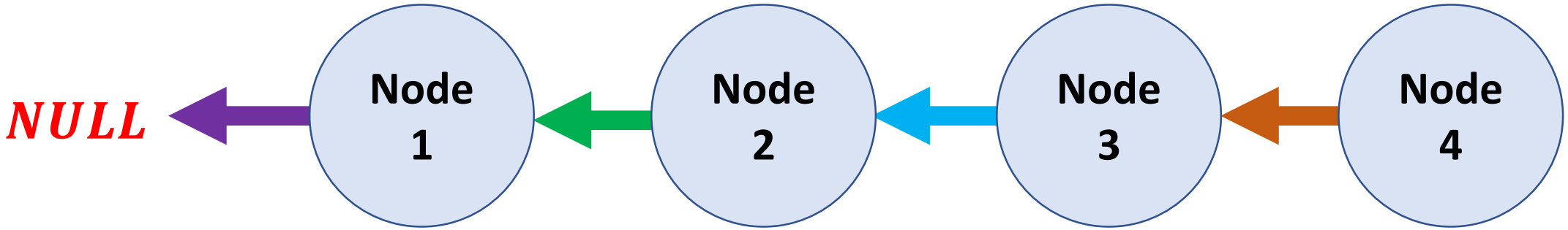
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

反轉

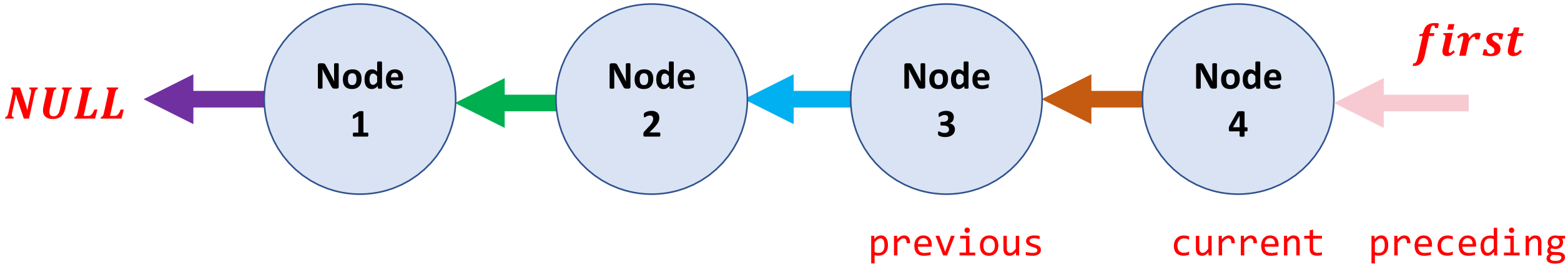


previous current preceding

```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

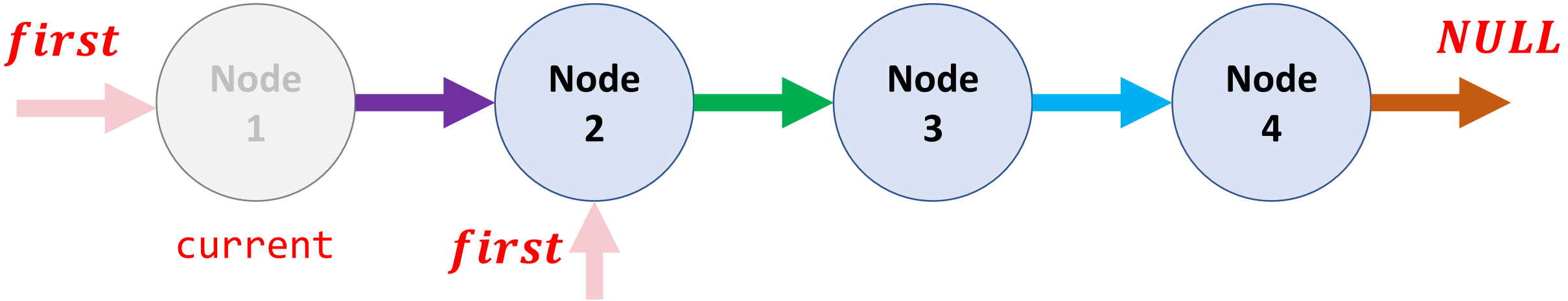
反轉




```
Initialization：
1.   previous = nullptr
2.   current = first
3.   preceding = current -> next
```

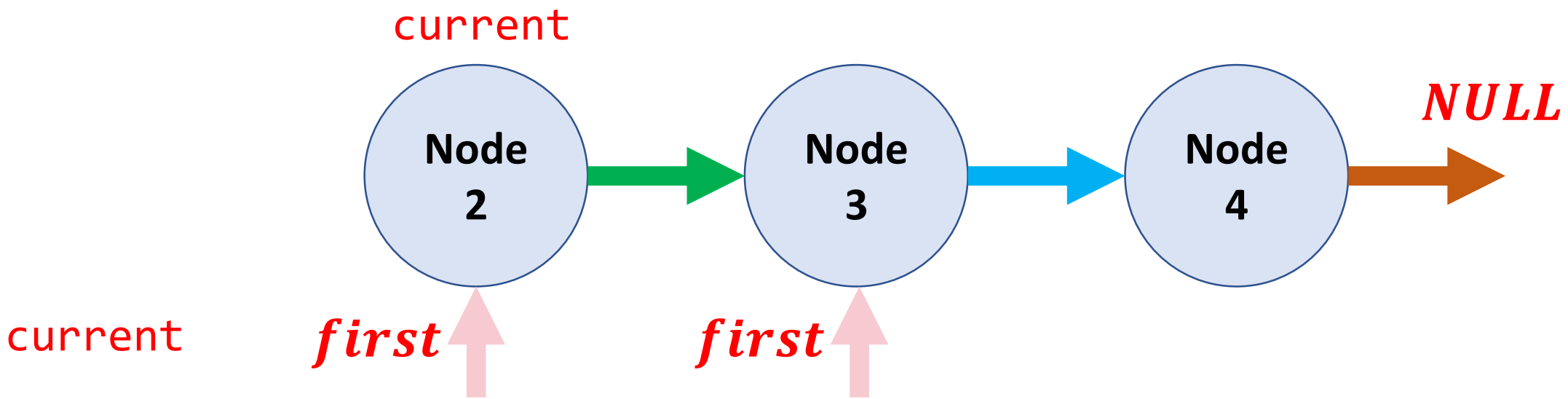
```
1.   current -> next = previous
2.   previous = current
3.   current = predceding
4.   predceding = predceding -> next
5.   repeat step 1~4
     (until predceding == nullptr)
6.   current -> next = previous
7.   first = current
```

清除



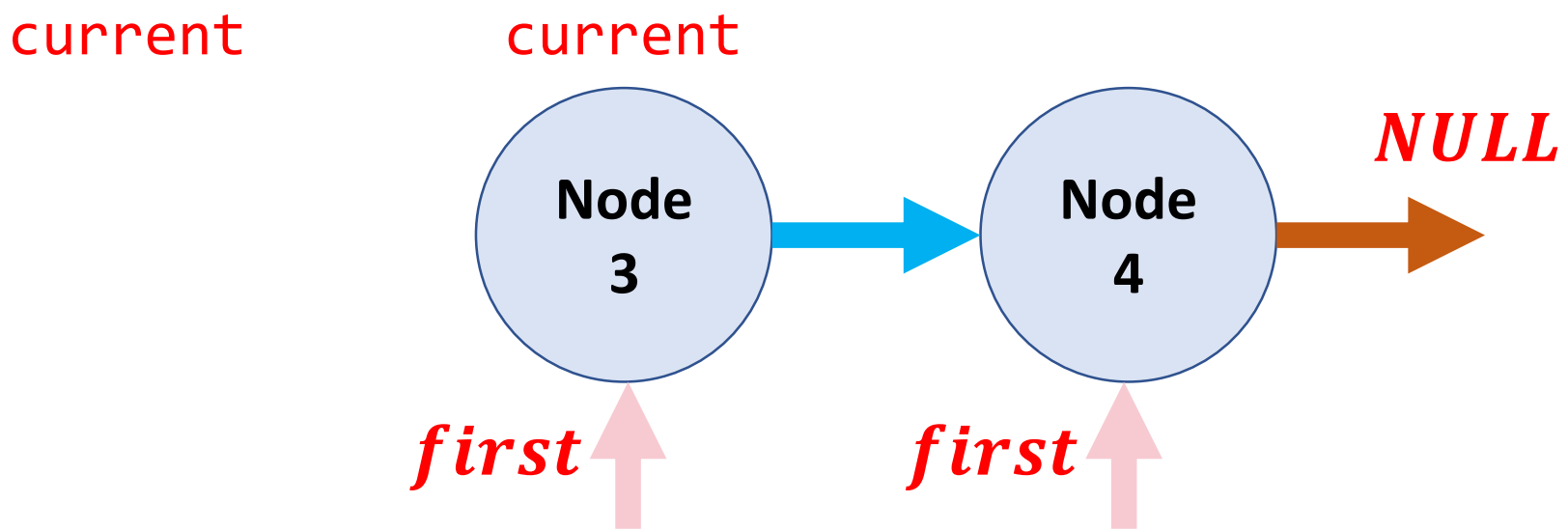

```
1.  current = first
2.  first = first->next
3.  delete current
4.  repeat step 1~3
        (until first == nullptr)
```

清除




```
1.  current = first
2.  first = first->next
3.  delete current
4.  repeat step 1~3
        (until first == nullptr)
```

清除




```
1.  current = first
2.  first = first->next
3.  delete current
4.  repeat step 1~3
    (until first == nullptr)
```

Example Code

Mission

- 在鏈結串列中新增以下函式：
  - ✓ 新增第一個元素 (Push_Front)
  - ✓ 刪除第一個元素 (Pop_Front)
  - ✓ 新增尾端元素 (Push_Back)
  - ✓ 刪除尾端元素 (Pop_Back)

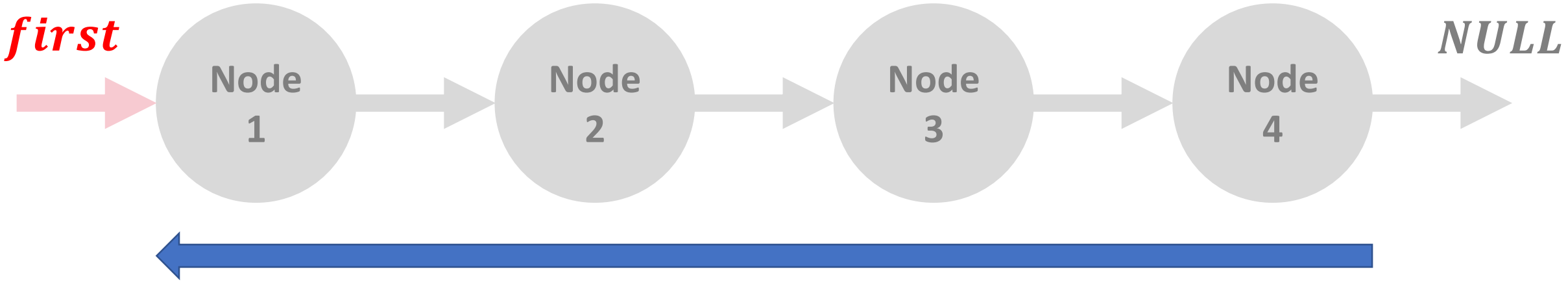
Practice

Mission

- 新增以下函式：
  - ✓ 清空所有資料 (Clear)
  - ✓ 反轉所有資料的方向 (Reverse)
- 視情況處理例外！

不同種 Linked List

不同種 Linked List

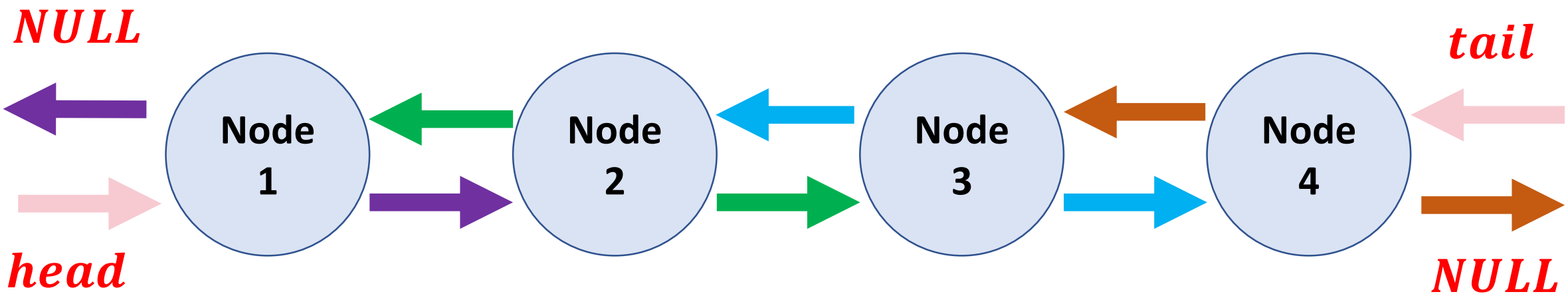


一出發就無法回頭了 :(

不同種 Linked List

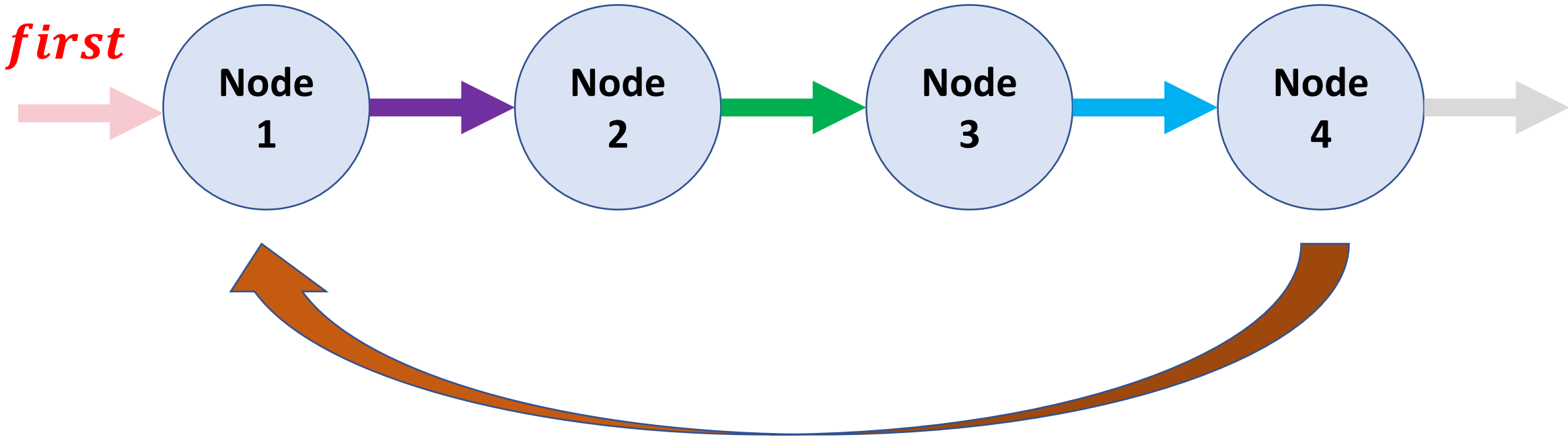
- 我們的Linked List一出發就無法回頭 :(
  - Single Linked List
  - 新增/刪除最後一個元素需要O(N)
    - ✓ 雖然可以開一個 last 指標解決啦...
- 其餘兩種 Linked List
  - Double Linked List
  - Circular Linked List

Double Linked List



會有兩個空指標！

Circular Linked List



Q：如何知道已經走完一圈了？

A：當目前 Node 的記憶體位置等於 first

Linked List 與 Array 的比較

比較	鏈結串列	陣列
記憶體需求	無需連續的記憶體	需連續的記憶體
記憶體大小	需額外空間儲存指標	無須額外空間
元素的資料型態	可不同	相同
插入/刪除複雜度	O(1)	O(n)
空間配置	可隨時調整	固定的大小
存取方式	只能依序存取	隨機(索引值)及依序存取
存取速度	慢	快

Example Code

Mission

- 把單向鏈結串列修改成雙向鏈結串列
  - node 內：prev 與 next
  - linked list 類別內：head 與 tail
- 在鏈結串列類別中新增迭代器，並支援
  - ~~比較運算子 (>、<、>=、<=、==、!=)~~
  - ++、--
  - +、-、*
  - 可直接使用 vector 裡的程式碼
- 新增外部函式 find(container, target)
  - 回傳找到的第一個元素迭代器
  - 若無，回傳空指標

Example Code

Mission

- 新增以下兩函式
  - ➢ 在特定位置新增資料
  - ➢ 在特定位置刪除資料
    - ✓ 利用迭代器指定要新增/刪除的位置！

Practice

Mission

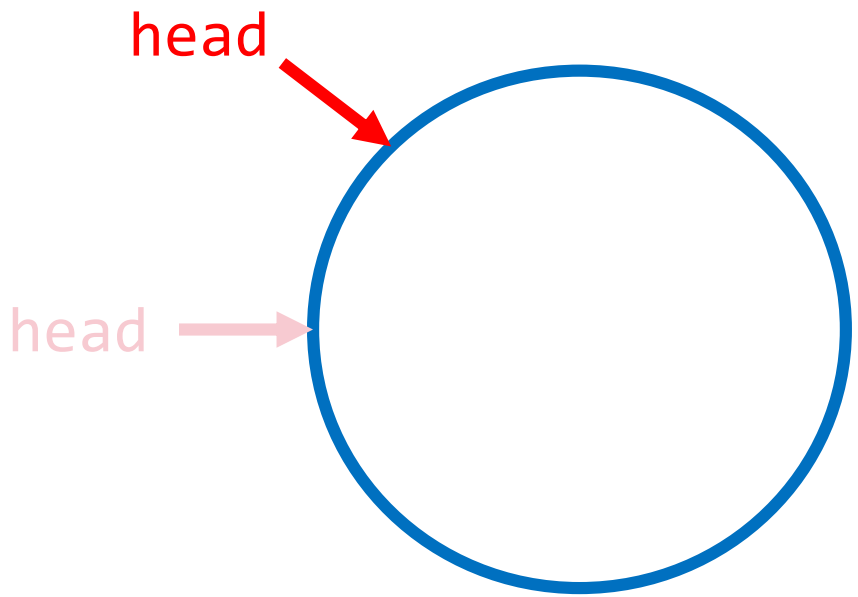
- 鏈結串列內的元素記憶體位置不連續，無法使用索引值
  - 迭代器直接記錄記憶體位置，可使用迭代器替代！
- 加入以下外部函式操作迭代器，並修改之前的函式
  - Advance(it, n)：迭代器 it 前後移動 n 步
  - Distance(it1, it2)：計算兩迭代器間的距離
  - Begin()：第一個元素的迭代器
  - End()：最後一個元素下一個的迭代器，即空指標
  - Prev(it)：迭代器 it 前面的迭代器
  - Next(it)：迭代器 it 後面的迭代器

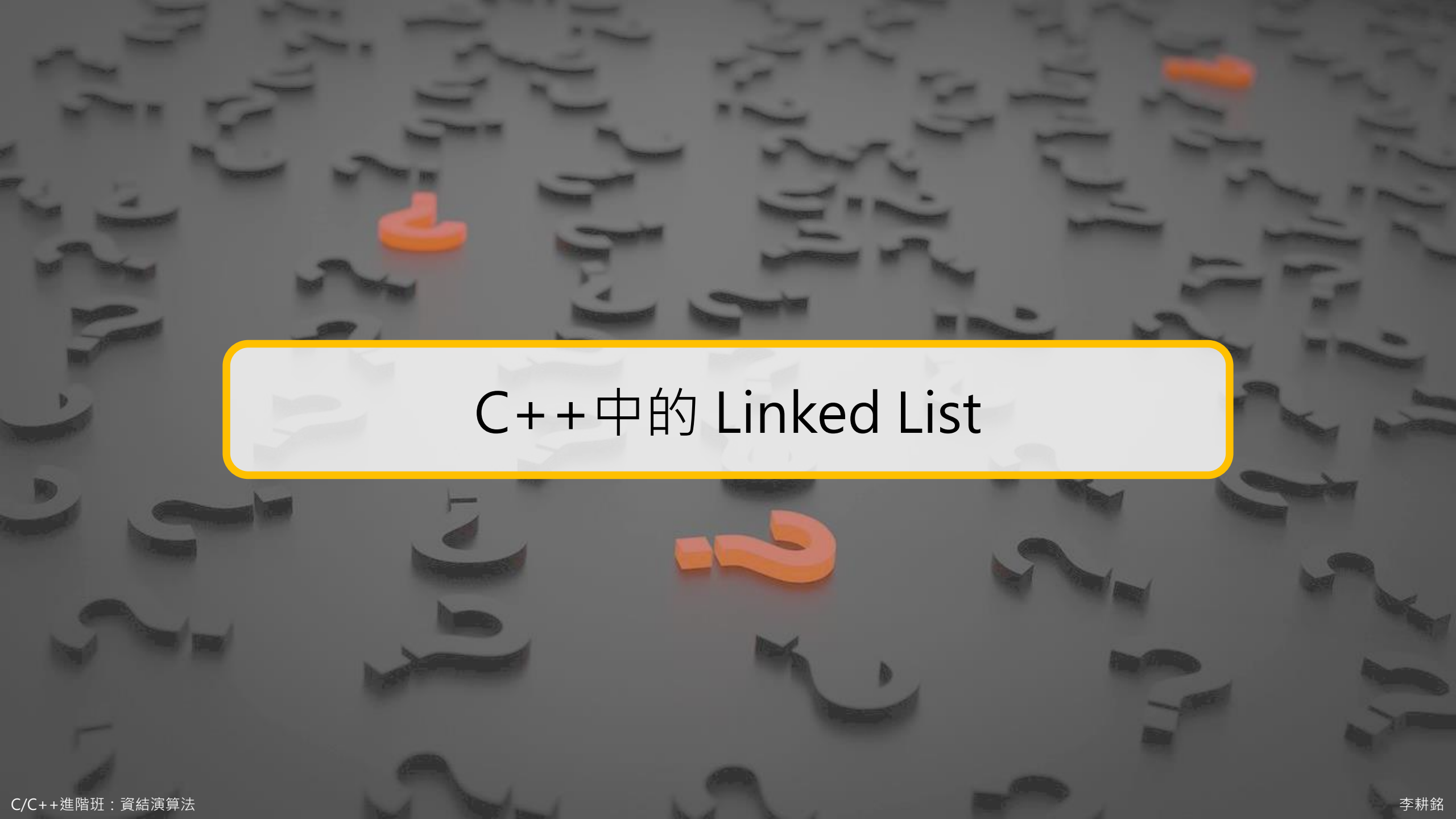
Practice

Mission

試著把鏈結串列改成 Circular Linked List ，並加入兩函式

- Remove：刪除特定資料
- Rotate：旋轉內部資料
  - ✓ 範例：ABCDE→Rotate 1→BCDEA
  - ✓ 範例：ABCDE→Rotate 2→CDEAB





C++中的 Linked List

List @ C++ STL

標頭檔

```
#include <list>
```

宣告

```
list<資料型別> 變數名稱;
```

迭代器

```
list<資料型別>:: iterator 變數名稱;
```

List @ C++ STL

- 實際上 STL 的 List 較少用
  - ➢ 不支援索引值查詢輸一半
  - ➢ 搜尋資料要 O(n) 再輸一半
    - ✓ 循序搜尋甚至比 vector 還慢
- 通常是需要手刻 Linked List 來解
  - ➢ 就像下面那幾題
  - ➢ 指標的使用與操作務必熟悉

List @ C++ STL

語法	功能	語法	功能
list.begin()	回傳第一個迭代器	list.insert()	插入元素至特定位置
list.end()	回傳尾端後一個的迭代器	list.remove()	刪除特定元素
list.empty()	確認 list 是否為空	list.reverse()	反轉整個 list
list.back()	回傳最後一個元素的值	list.size()	回傳 list 的長度
list.front()	回傳第一個元素	list.sort()	把 list 排序
list.push_back()	在尾端新增一個元素	list.swap()	互換兩個 list 的值
list.push_front()	在開頭新增一個元素	list.unique()	刪除 list 裡重複的值
list.pop_back()	刪除最後一個元素	list.merge()	合併兩個 list
list.pop_front()	刪除第一個元素	list.splice()	切割兩個 list
list.clear()	清空所有元素	advance()	移動迭代器

List @ C++ STL

語法	功能
advance(it, n)	迭代器 it 前後移動 n 步
distance(it, it)	回傳兩迭代器的距離
begin(container)	回傳容器中第一個元素的迭代器
end(container)	回傳容器中最後一個元素下一個的迭代器
prev(it)	回傳迭代器 it 前面的迭代器
next(it)	回傳迭代器 it 後面的迭代器

就是我們剛剛刻的！

List @ C++ STL

```cpp
#include <iostream>
#include <list>

using namespace std;

void print_list(list<int> List){
    for(auto iter=List.begin();iter!=List.end();iter++)
        cout << *iter << " ";
    cout << endl;
}
```

```cpp
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;
    cout << "Back: " << List.back() << endl;
    List.pop_front(); // 0 0 1 2 3
    List.pop_back(); // 0 0 1 2

    auto iter = List.begin();
    advance(iter, 2); // iter -> 1
    List.insert(iter, 4); // 0 0 4 1 2, iter -> 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;
}
```

List @ C++ STL

- List 底層是 double Linked List
- 可在任意位置新增或刪除，O(1)
  - 不計搜尋的時間
  - 新增或刪除的效率較好
- 搜尋需要線性時間，O(N)
- forward_list 是 single Linked List
- 不支援隨機訪問或索引值
- 需要額外空間存放記憶體位置

Example Code

Mission

- 宣告並建立 STL 中的鏈結串列，使其：
  - 儲存整數型態並依序輸入值，直至 0 結束
  - 最後利用迭代器印出鏈結串列中所有資料

Example Code

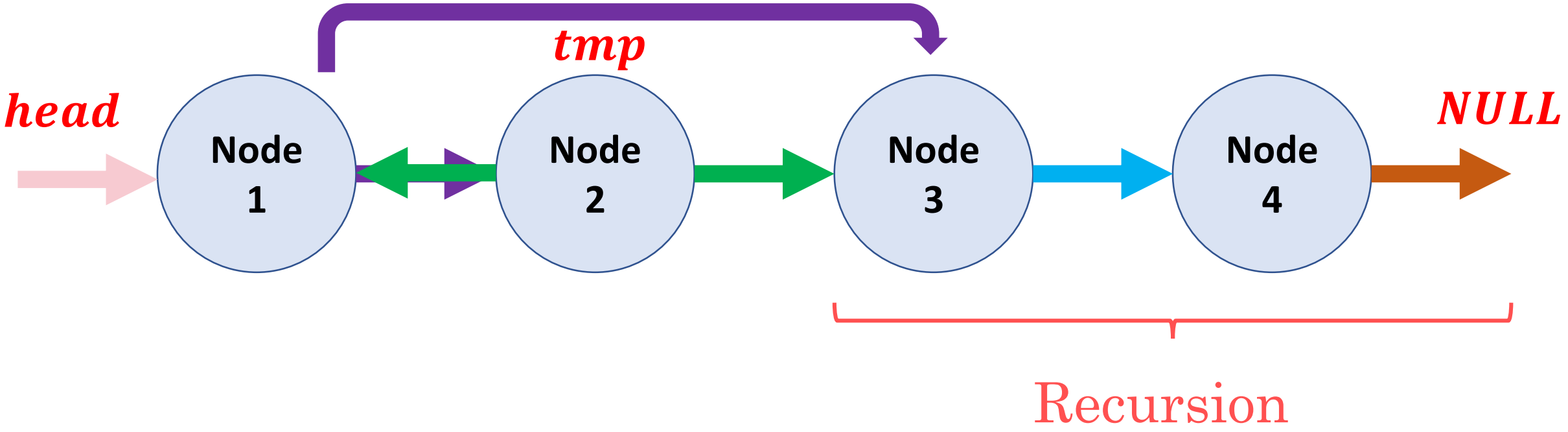
Mission

LeetCode #24. Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

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

Example Code



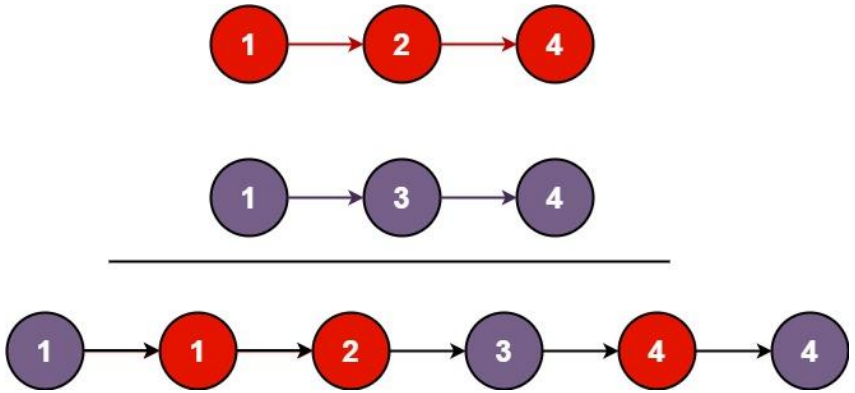

```
1.  Node* tmp = head->next
2.  head->next = swapPairs(tmp->next)
3.  tmp->next = head
4.  return tmp
```

Practice

Mission

LeetCode #21. Merge Two Sorted Lists

Merge two sorted linked lists and return it as a sorted list. The list should be made by splicing together the nodes of the first two lists.



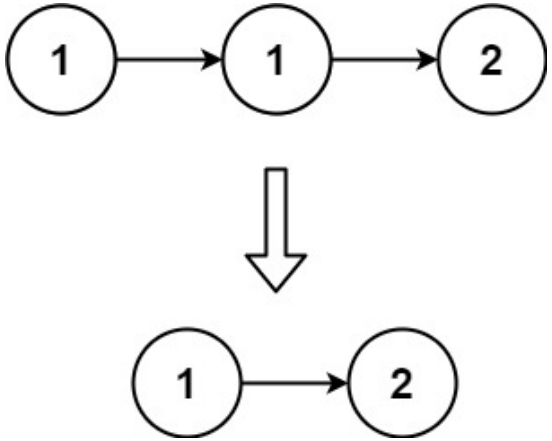
Ref：https://leetcode.com/problems/merge-two-sorted-lists/

Practice

Mission

LeetCode #83. Remove Duplicates from Sorted List

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.



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

Practice

Mission

LeetCode #707. Design Linked List

Design your implementation of the linked list. You can choose to use a singly or doubly linked list.

A node in a singly linked list should have two attributes: val and next. val is the value of the current node, and next is a pointer/reference to the next node.

If you want to use the doubly linked list, you will need one more attribute prev to indicate the previous node in the linked list. Assume all nodes in the linked list are 0-indexed.

Ref：https://leetcode.com/problems/design-linked-list/

Practice

Mission

已知 $S_0 = 'ABC', S_n = 'A' +' S_{n-1} +' 'B' +' S_{n-1} +' 'C'$

$S_1 = A + ABC + B + ABC + C$

試印出 S_{50} 的前 1000 個字元。

Practice

Mission

已知 $S_0 = 'ABC', S_n = 'A' + S_{n-1} +' B' + S_{n-1} +' C'$，

試印出 S_{50} 的前 1000 個字元。

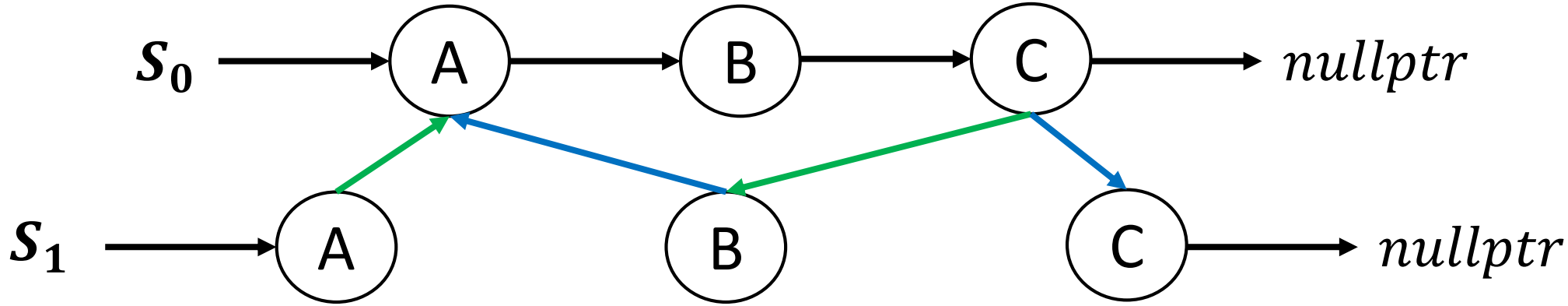
$$
\begin{aligned}
&Length(S_n) \\
&= 3 + 2 \times Length(S_{n-1}) \\
&= 3 + 2 \times (3 + 2 \times Length(S_{n-2})) \\
&= 3 + 2 \times 3 + 2^2 \times 3 + \cdots + 2^n \times Length(S_0) \\
&= 3 + 2 \times 3 + 2^2 \times 3 + \cdots + 2^n \times 3 \\
&= \frac{a_0(r^n - 1)}{r - 1} = \frac{3(2^{n+1} - 1)}{2 - 1} = 6 \times 2^n - 3 \in 2^n
\end{aligned}
$$

Practice

Mission

已知 $S_0 = 'ABC', S_n = 'A' + S_{n-1} +' B' + S_{n-1} +' C'$，

試印出 S_{50} 的前 1000 個字元。



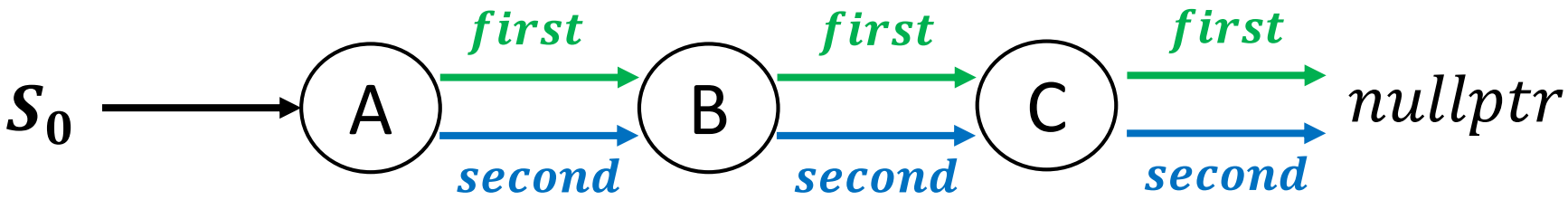
$$
\begin{aligned}
&Length(S_n) \\
&= 3 + Length(S_{n-1}) \\
&= 3 + 3 + 3 + \cdots + Length(S_0) \\
&= 3n + Length(S_0) \\
&= 3n + 3 \in n
\end{aligned}
$$

Practice

Mission

已知 $S_0 = 'ABC', S_n = 'A' + S_{n-1} +' B' + S_{n-1} +' C'$，

試印出 S_{50} 的前 1000 個字元。

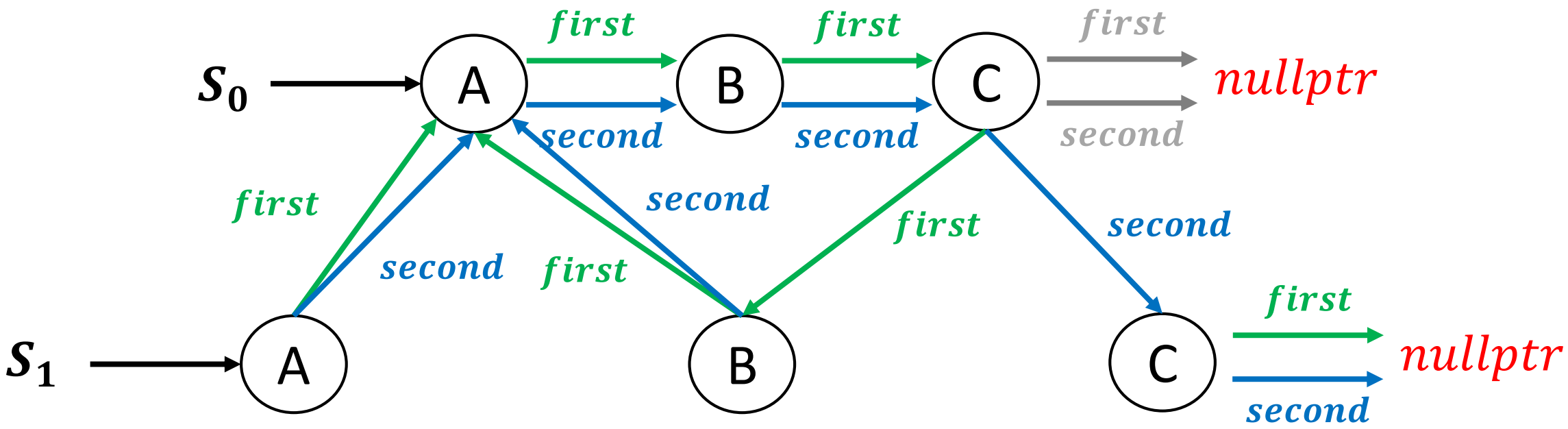


Practice

Mission

已知 $S_0 = 'ABC', S_n = 'A' + S_{n-1} +' B' + S_{n-1} +' C'$，

試印出 S_{50} 的前 1000 個字元。

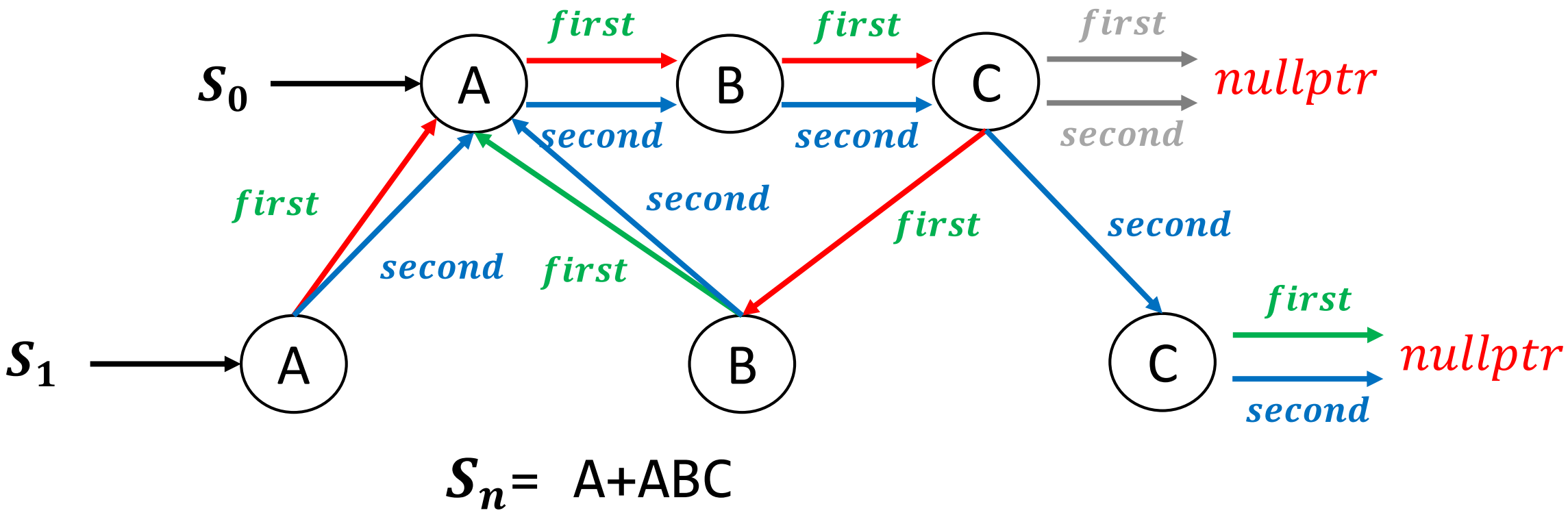


Practice

Mission

已知 $S_0 = 'ABC', S_n = 'A' + S_{n-1} +' B' + S_{n-1} +' C'$，

試印出 S_{50} 的前 1000 個字元。

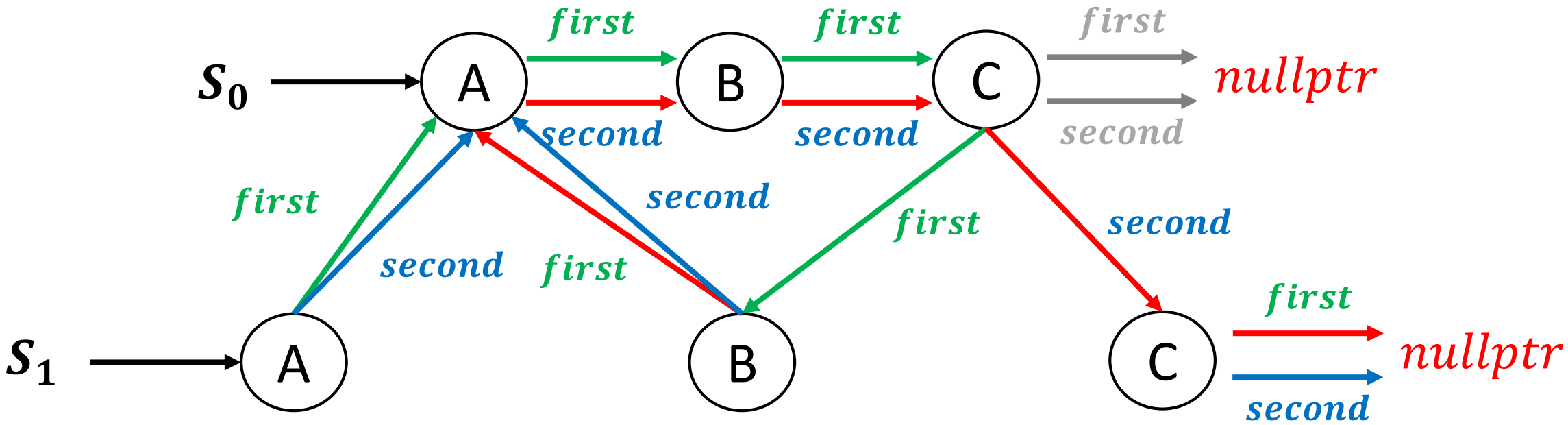


Practice

Mission

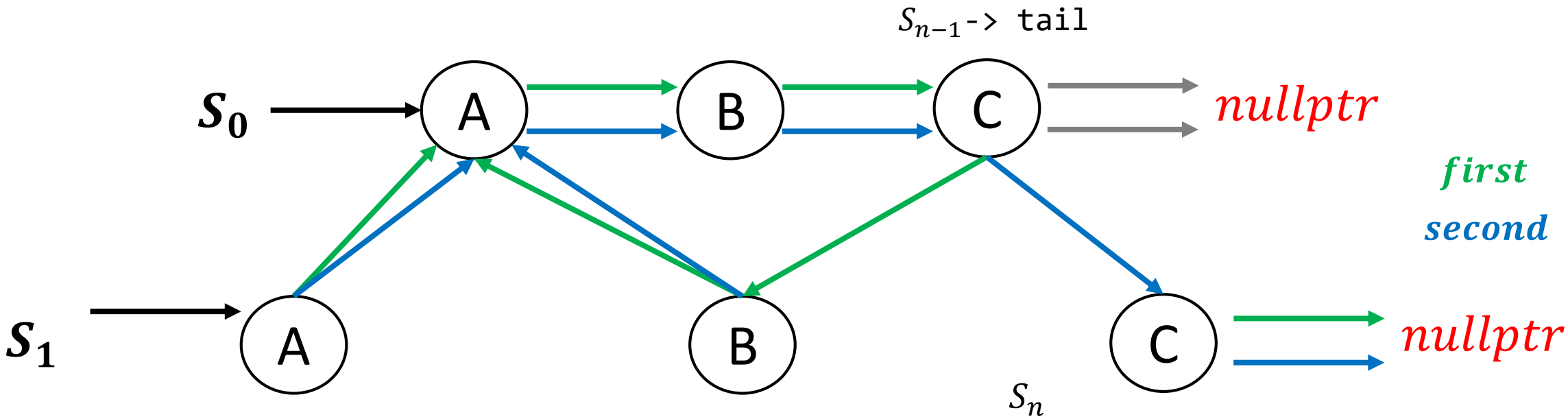
已知 $S_0 = 'ABC', S_n = 'A' + S_{n-1} +' B' + S_{n-1} +' C'$，

試印出 S_{50} 的前 1000 個字元。



S_n= A+ABC+B+ABC+C

Practice



node

1. char：data
2. node*：first
3. node*：second
4. bool：read_first

1. new A, B, C
2. A->first, A->second = $S_{n-1} \rightarrow head$
3. B->first, B->second = $S_{n-1} \rightarrow head$
4. C->first, C->second = $nullptr$
5. S_{n-1}-> tail -> first = &B
6. S_{n-1}-> tail -> second = &C
7. S_n->head = &A
8. S_n->tail = &C