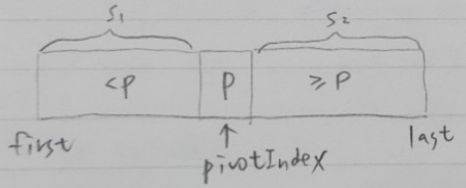
CH1 遞迴 Recursion

中 迴圈 iteration

- 遞迴
  ① 把問題變小，變小後問題還是同一個。(把大問題變小問題)
  ⇒ 一份程式碼即可 ⇒ 精簡
  ② 不一定速度很快
  ③ 可讓寫懂程式的時間縮短。
  ④ Linear Recursion, Binary recursion
  ⑤ 經典問題:
  Factorial 階乘, Greatest Common Divisor 最大公因數
  Search in Array 搜尋, Fibonacci series 費式數列
  Combinatorial numbers 組合數, Towers of Hanoi 河內塔

鏡子裡有鏡子，且越來越小

- Find the k^{th} smallest Item in a Array
  ① 在 Array 中找一個 pivot item (本區新井)
  ② 進行分區，把值 ≥ pivot item 放右，< pivot item 放左 ⇒ 分成 2 區
  ③ 看要找的位置在哪(只找其中一邊)，並重覆以上動作(遞迴)




```
ksmall(k, anArray, first, last); // kth smallest item in anArray[first..last]
```

Ex: 設 k=4, pivot item = anArray[first]

<anArray>: (30), 7, 25, 39, 19, 48, 2, 16, 12

↑ pivot item

位子>4

⇒ 7, 25, 19, 2, 16, 12, (30), 39, ~~48~~

ANS 往這找　不用看

過程中，找到了30在排序中正確的位子

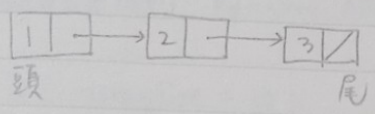
- Tail Recursion

```
void writeBackward (string s, int size) {
  if (size > 0) {      // write the last character
    cout << s.substr(size-1, 1);
    writeBackward (s, size-1);
  } // if()
} // writeBackward
```

Tail Recursion,
聰明的編譯器會轉成迴圈
⇒ 效率↑

```
void WB (string s, int size) {
  while (size > 0) {
    cout << s.substr(size-1, 1);
    --size;
  } // while()
} // WB
```

CH3 鏈結串列 Linked List



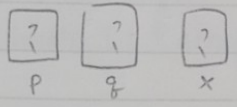
• Pointer 指標

Ex: int *p ; // ① Initially undefined, but not NULL
(初始值未定義，但不是NULL)
② static allocation 靜態配置

△基本使用

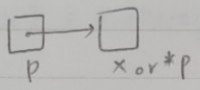
① 宣告

int *p, *q ;
int x ;



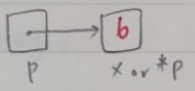
② 指向一個靜態配置的memory (statically allocated)

p = &x ;



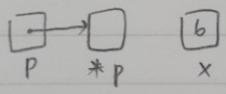
③ assign a value

*p = 6 ;



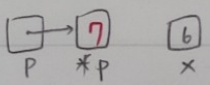
④ 動態配置記憶體 allocating memory dynamically

p = new int ;



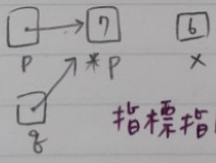
⑤ assign a value

*p = 7 ;



⑥ copy a pointer

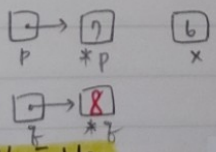
q = p ;



指標指向同一塊記憶體

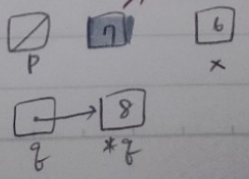
⑦ 動態配置記憶體 & assign a value

q = new int ;
*q = 8 ;



⑧ assign NULL to a pointer variable

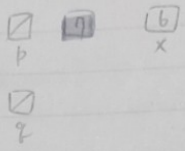
p = NULL ;



memory leak ⇒ 佔記憶體，直到程式執行完，記憶體才會被釋放。

⑨ deallocating memory 完全清除

```
delete q;
q = NULL;
```



☆ The delete operator returns dynamically allocated memory to the system for reuse, and leaves the variable's contents undefined.

所以delete完後，要再設成NULL，才算完全清空!!

• 動態配置陣列 Dynamic Allocation of Arrays

```
int arraySize = 50;
double *anArray = new double [arraySize];
```

① 陣列名稱 = 指標

⇒ anArray[2] ≡ *(anArray+2)

② 配置更大的空間

```
double *oldArray = anArray;          舊的 50 個
anArray = new double [3 * arraySize];   新的 150
```

資料搬移：// 一個一個搬到新的

```
for (int index=0 ; index < arraySize; ++index)
    anArray [index] = oldArray [index];
```

刪除

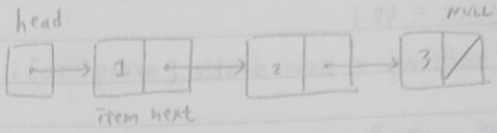
```
delete [] oldArray;     // deallocation array
```

一群羊

• Pointer-Based Linked Lists

① The head pointer points to the first node in a linked list

```
struct Node {
    int item;
    Node *next;
};
```



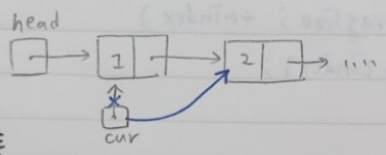
② If head is NULL, the linked list is empty.

```
Node *p;        // pointer to node
p = new Node;   // allocate node
```

① 走訪

```
for ( Node *cur = head; cur != NULL; cur = cur->next )
```

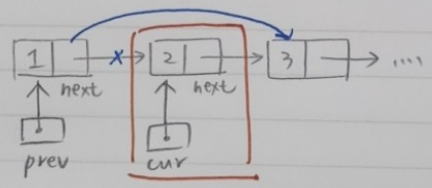
current



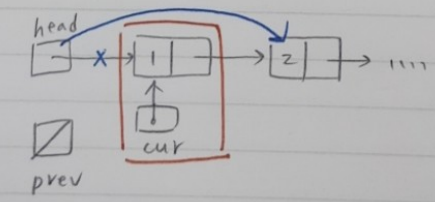
② 刪除

(1) `prev->next = cur->next;`

中間



第一個



`head = cur->next;`

把節點刪除　　※順序不能換

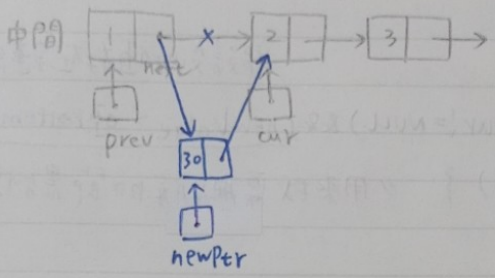
```
cur->next = NULL;
delete cur;
cur = NULL;   ⇒ Avoid dangling reference
```

③新增

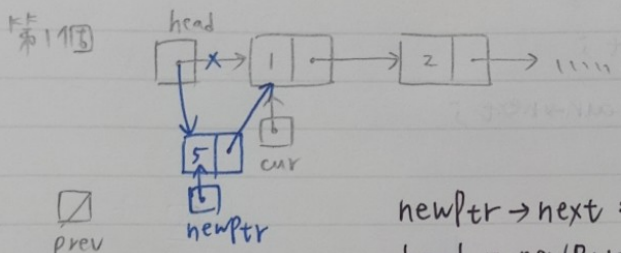
```
newPtr → next = cur;
prev → next = newPtr;
```

※順序可換

中間



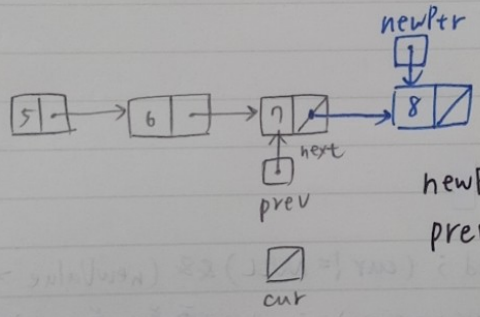
第1個




```
newPtr → next = head;
head = newPtr;
```

※順序不能換

尾巴




```
newPtr → next = NULL;
prev → next = newPtr
```

※順序可以換

• Sorted Linked List

① 走訪&刪除

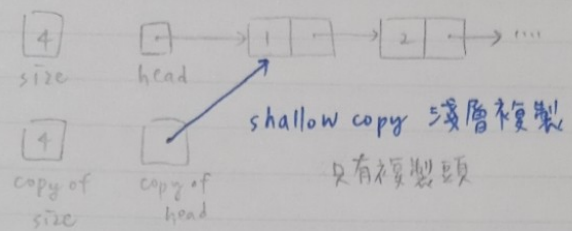
```
Node *prev, *cur;
if (head != NULL) {
    for( prev = NULL, cur = head; (cur != NULL) && (newValue > cur->item);
         prev = cur, cur = cur->next) ;   // 用來找要刪除的節點位置

    if (prev == NULL)   // 第1個
        head = cur->next;
    else prev->next = cur->next;

    cur->next = NULL;
    delete cur;
    cur = NULL;
} // if
```

假設此值存在鏈結串列中

② 走訪&新增

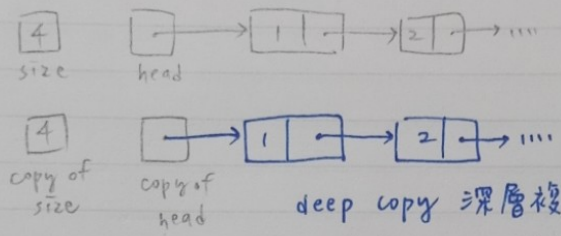
```
for ( prev = NULL, cur = head; (cur != NULL) && (newValue > cur->item);
      prev = cur, cur = cur->next ) ;   // 找要新增的位置

if (prev == NULL) {   // 第1個
    newPtr->next = head;
    head = newPtr;
} // if()
else {
    newPtr->next = cur;
    prev->next = newPtr;
} // else()
```

• shallow copy v.s. Deep copy



shallow copy 淺層複製

只有複製頭



deep copy 深層複製

擁有自己的一列(相同資料ba)

constructor
&
destructor

3-09

Deep copy
&
shallow copy