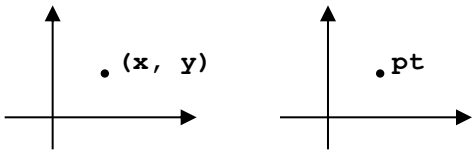
什麼是 C Structures？如何產生 `int`、`double`、字串、陣列等等基本資料型態之外，更複雜的資料儲存方式？

Structures 是多個相關的變數的集合，用一個共同的名稱來統稱。這樣的方式在使用上較方便，也會讓程式更簡潔易懂。譬如要描述平面上的點座標，可以用

```
int x, y;
```

若使用 structures 則可以自定一個叫做 `t_point` 的資料型態，寫成

```
struct t_point {
   int x;
   int y;
};
```



關鍵字 `struct` 後面接著的就是關於 `t_point` 這個自定的資料結構的宣告，`t_point` 是自己替這個資料結構取的名字，括號中間就是這個 structure 所包含的資料結構，括號後面要記得加上分號做結束。宣告過 structure 之後可以用它來定義變數

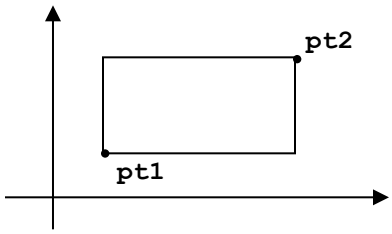
```
struct t_point pt;
```

所以 `pt` 會包含 `x` 和 `y` 兩個 members，要設定或更改 members 的內容，最直接的方式是使用 member operator `.` 來存取。

```
pt.x = 10;
pt.y = 20;
```

已經宣告過的 structure 可以再拿來宣告另一個 structure，例如

```
struct t_rect {
   struct t_point pt1;
   struct t_point pt2;
};
```



然後用它來產生變數

```
struct t_rect screen;
```

以及存取 members

```
printf("%d %d\n", screen.pt1.x, screen.pt1.y);
```

如何使用 C Structures 來做運算？

Structures 可以使用的運算元只有 `=`、`&`、`.`，其他的運算則必須自己寫 functions 來達到我們想要的功能，例如，想要比較兩個 structure 變數相不相等，不能直接用 `==` 或 `!=`， 另外相加 `+` 和相減 `-` 也不能用。接下來就來看幾個自定 functions 的例子。

```
struct t_point makePoint(int, int);
struct t_point midPoint(struct t_point, struct t_point);

struct t_point makePoint(int x, int y)
{
   struct t_point pt;
   pt.x = x;
   pt.y = y;
   return pt;
}
struct t_point midPoint(struct t_point p1, struct t_point p2)
{
   p1.x = (p1.x + p2.x) / 2;
   p1.y = (p1.y + p2.y) / 2;
   return p1; /* 把 p1 回傳，會將 p1 的 members 的值一一複製 */
}
```

在主程式裡呼叫 `makePoint()` 來設定 `struct t_point` 變數

```
struct t_rect screen;
struct t_point mpt;
screen.pt1 = makePoint(1, 1);
screen.pt2 = makePoint(1024, 768);
mpt = midPoint(screen.pt1, screen.pt2);
```

[練習 WD_01]
寫一個 function **ptInRect()** 來判斷某個點 **p** 是否落在某個長方形 **r** 裡面，是則回傳 **1**，否則回傳 **0**。

```
int ptInRect(struct t_point p, struct t_rect r);
```

Structures 參數的傳遞方式

傳遞 structure 變數到 function，會用 call-by-value 方式，所以在 function 裡改變 structure的 members 的值，並不會影響外部的 structure 變數的內容。

Structures 同樣可以用指標方式來達到 call-by-reference 的效果，通常用傳遞指標的方式會比傳遞整個 structure 來得有效率 (call-by-value 需要複製整個 structure 變數的內容) 。

在討論指標之前，先來看一個額外的功能。C 提供 **typedef** 來讓我們宣告新的型別名稱，這樣接下來宣告變數會比較方便。譬如，在原來 **struct t_point** 的宣告之後，使用 **typedef** 宣告一個新名字來取代原來的寫法。

```
struct t_point {
   int x;
   int y;
};
typedef   struct t_point  Point;
```

原本產生變數必須使用

```
struct t_point pt;
```

現在只要用

```
Point pt;
```

所以現在宣告一個指到 **struct t_point** 的指標可以用

```
struct t_point *pp;
```

或

```
Point *pp;
```

把 **pp** 指到 **pt** 就和一般變數一樣，要使用 **&** 來取得 structure 的位址：

```
pp = &pt;
```

透過指標來存取 structure 的 members 有兩種寫法，一種是用標準的 ***** 先找到指標指的內容，再用 **.** 來取出 member：

```
(*pp).x = 10;  /* 一定要加括號 */
```

另一種則是用 **->** 符號，相當於上面的縮寫

```
pp->x = 10;
```

[練習 WD_02]
宣告新的型別 **Rect** 來代表 **struct t_rect**，產生指到 **Rect screen** 的指標，透過指標修改 **screen** 範圍。

Structures 陣列

陣列的使用更能說明 structures 的必要性。以 **struct t_point** 為例，要產生 100 個平面上的點來記錄 100 組座標，如果不使用 structure 要寫成

```
int x[100], y[100];
```

使用 structure 變成

```
struct t_point pt[100]; /* 或使用前面 typedef 所宣告的新型別名稱 Point pt[100]; */
```

由於在這個例子中，平面點座標應該是一體的資料，包含 **x** 和 **y** 座標，所以使用 structure 的方式更合理。另外，計算 structure 陣列的大小，仍然可以使用 **sizeof**

```
sizeof(pt) / sizeof(Point)
```

和一般變數相同， structure 陣列參數的傳遞是用 call-by-reference 的方式， 譬如 **f(pt)** 會傳陣列的起始位址。

[練習 WD_03]
寫出 function **randPoint()** 隨機在某個長方形 r 範圍內產生 **n** 個點。

```
void randPoint(Rect r, Point p[], int n);
```

產生的點儲存在 **p[]** 裡面。然後再寫出 funciton **meanPoint()** 計算 **p[]** 裡頭的 **n** 個點的 **mean**。

```
Point meanPoint(Point p[], int n);
```

Unions

除了 structures 之外，C 還提供另一種衍伸型別 unions，讓我們用**同一塊區域**來儲存不同類型的資料，例如

```
union u_tag {
   int ival;
   float fval;
   char *sval;
};
union u_tag u;
```

用這樣方式產生的變數 **u** 會有剛好足夠的空間來儲存 **int**、**float**、**char*** 三種型別中最大的型別資料。變數 **u** 每次只能儲存三種型別資料中的一種，我們必須自己記得目前 **u** 所儲存的資料是哪一種型別。
執行程式看看下面的輸出是什麼，就會對 **union** 的作用更了解。

```
printf("%d\n", sizeof(u)); /* 4 雖然三個成員各為 4 bytes 但必須共用同一個空間*/
u.ival = 10;
printf("%d\n", u.ival);    /* 10 */
u.fval = 20.0;             /* 後來設的值會把前面設的蓋掉 */
printf("%f\n", u.fval);    /* 20.000000 */
printf("%d\n", u.ival);    /* 1101004800 沒有意義 因為浮點數和整數的儲存方式不同*/
u.sval = "abcde";
printf("%d\n", u.ival);    /* 4202504 指標所指到的位址*/
printf("%s\n", u.sval);    /* abcde */
printf("%s\n", u.ival);    /* abcde 位址的儲存方式如果和整數的方式相同就能顯示出字串 */
```

Bit Manipulations

C 語言提供了一些 operators，讓我們能用更低階的方式存取和修改資料。這些 operators 包括

& (AND)、**|** (OR)、**^** (XOR)、**>>** (right shift)、**<<** (left shift)、 **~** (one's compliment)

有時候我們需要儲存的資訊可能只有 **1** 和 **0** 兩種值，譬如記錄某種狀態 "有" 或 "無" (所謂的 flag)。當我們需要記錄大量這一類的資料時，若使用 **int** 來記錄會太浪費空間，這種情況就適合使用 bit 運算，只需用到原本的空間的 1/32。

```
#include <stdio.h>
void dispBits( unsigned val )
{
   int i;
   unsigned mask = 1 << 31;              /* 1000...000    1 後面接 31 個 0          */
   for (i=0; i<32; i++) {                /*(mask & val)!=0 如果是true會印出 1        */
      printf("%d", ((mask & val) != 0) );/*(mask & val)!=0 如果是false會印出 0       */
      mask = mask >> 1;                  /* mask 的 1 向右移動 1 格  用來取下一個 bit */
   }
   printf("\n");
}
int main(void)
{
   unsigned n;
   n = 100;
   dispBits(n);
   dispBits(0x1f);   /* 0x1f 是 31 的十六進位表示, 換成二進位是 000...00011111 */
   dispBits(n & 0x1f);
   dispBits(n | 0x1f);
   dispBits(n ^ 0x1f);
   dispBits( ~n );
   return 0;
}
```

[練習 WD_04]
用圖示來解釋下面的運作原理

```
unsigned getBits(unsigned x, int p, int n)
{
   return ( x >> (p-n) ) & ~( ~0 << n );  /* 取出 x 的第 p 位置起 n 個 bits */
}
```

寫出 **unsigned invert(x, p, n)** 把 **x** 第 **p** 位置起 **n** 個 bits 由 **0** 變 **1**，**1** 則變為 **0**。
寫出 **unsigned rightRotate(x, n)** 傳回 **x** 向右 rotate **n** bits 之後的結果。

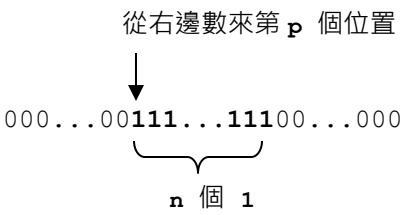
以 **getBits(100, 6, 4)** 為例，會從右邊數來第 **6** 個 bit 開始，取出 **4** 個 bits
十進位 **100** 用二進位表示為 00000000000000000000000001**1001**00 深色是我們要取出的 bits
dispBits(getBits(100, 6, 4)); 會得到
00000000000000000000000000**1001**

運作原理：
~0 會得到 111...11111111 三十二個 **1**
~0 << n 會得到 111...11110000 在這個例子 **n = 4**，所以最右邊四個 bits 因為向左 shift 的關係會補零
~(~0<<n) 會得到 000...00001111 反轉的結果，這個結果會被當作 mask

x>>(p-n) 把 000...01100100 向右 shift 兩個 bits (**6-4**)
變成 000...00011001 左邊多出來的空位會補零
最後把 **(x>>(p-n)) & ~(~0<<n)** 相當於 000...00011001 & 000...0000**1111**
得到我們要的結果 000...0000**1001**

invert() 可能的寫法之一：

```
unsigned invert( unsigned x, int p, int n )
{
   return  x ^ (~(~0 << n) << (p-n));
}
```



想辦法產生一個 mask, 能讓 **p** 位置後接 **n** 個 **1**, 其餘位置都是零：
這樣的 mask 可以用 **~(~0 << n) << (p-n)** 產生
有了這個 mask, 只要把它和 **x** 做 **XOR** 就可得到 **invert()** 要做的效果
因為任何 bit 和 0 做 XOR 的結果會維持不變，
而任何 bit 和 1 做 XOR 則會變成 complement

rightRotate() 可能的寫法之一：

```
unsigned rightRotate(unsigned x, int n)
{
   return ((x & ~(~0 << n)) << (sizeof(x)*8 - n)) | (x >> n) ;
}
```

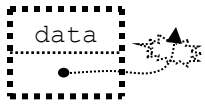
第一部份用 **(x & ~(~0 << n))** 取出最右邊 **n** 個 bits，然後向左 shift **(sizeof(x)*8 - n)** bits,
第二部份用 **(x >> n)** 把 **x** 向右 shift **n** bits，把兩個部份做 **OR** 組合起來就可以做出向右 rotate 的效果。

Self-Referential Structures

學過指標和 **struct** 之後，就可以靠結合這兩樣東西，變出很多花樣。我們會介紹一些基本的動態資料結構，包括 linked lists、 stacks、 queues、 trees。這些資料結構搭配演算法，就會有很多實際用途。

先來看 Self-Referential Structures，如何讓 **struct** 中包含指向同一種 **struct** 的指標。標準寫法如下：

```
struct t_node {
   int data;
   struct t_node *nextPtr;
};
```



關鍵就是第二個 member，是一個能夠指向 **struct t_node** 的指標。我們用這樣的 **struct** 來產生兩個變數，

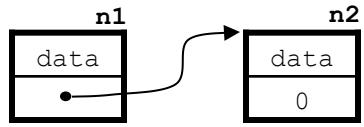
```
struct t_node n1, n2;
```

還記得當你產生指標的時候，牠還沒被初始化，也就是還沒指到有意義的位址，所以還不能拿來使用。所謂有意義的位址，必須是已經產生的變數所在的位址，用 **&** 符號來取得，譬如 **&n2** 就是一個有意義的位址。所以，我們可以寫出下面的敘述：

```
n1.nextPtr = &n2;
```

再來 **n2.nextPtr** 也可幫牠指到一個合法的地方，譬如 **NULL**：

```
n2.nextPtr = NULL;
```

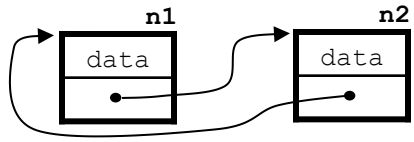


當然也可以把牠指向 **&n1** 位址

```
n2.nextPtr = &n1;
```

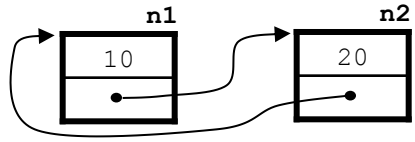
n1 和 **n2** 的 **data** 都還沒設定，我們現在可以用 **n1.netxPtr** 和 **n2.nextPtr** 互相幫對方設定。

```
(*n1.nextPtr).data = 20;
(*n2.nextPtr).data = 10;
```



上面的敘述還有另一種寫法：

```
n1.nextPtr->data = 20;
n2.nextPtr->data = 10;
```

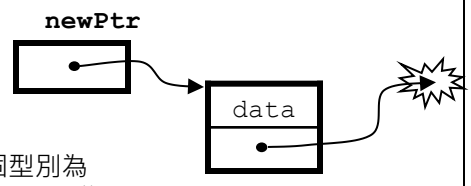


[練習 WE_01]
先產生三個 **struct t_node** 變數，分別叫做 **n1**、**n2**、**n3**
n1.nextPtr 指向了 **&n2** 位址， **n2.nextPtr** 指向 **&n3** 位址， **n3.nextPtr** 指向 **NULL** 位址，然後用一個新的指標 **struct t_node *p** 指向 **&n1** 位址。寫一個迴圈，只使用 **p** 來把 **n1**、**n2**、和 **n3** 的 **data** 都設成 **500**。

Dynamic Memory Allocation

前面的例子中，**n1**、**n2**、**n3** 都是在 compile 的時候就決定了，因為一開始我們就假設要用到三個變數。假如我們想讓程式在執行的時候，才依照外在狀況產生新的 **struct t_node**，就必須動態取得記憶體來儲存變數。在 stdlib.h 裡頭有兩個 functions: **malloc()** 和 **free()**，我們可以靠這兩個 functions 動態取得和釋放記憶體。用 **malloc()** 取得的記憶體大小是動態設定的， **malloc()** 會傳回取得的整塊記憶體的**起始位址**，你的程式就要靠這個起始位址找到你取得的整塊記憶體。程式開頭要加 **#include <stdlib.h>**，

```
struct t_node *newPtr;
newPtr = (struct t_node *)malloc(sizeof(struct t_node));
```



用 **sizeof(struct t_node)** 算出需要多少個 bytes，
然後用 **malloc()** 取得足夠的記憶體區塊的起始位址。**malloc()** 會傳回一個型別為 **void*** 的指標代表起始位址，**void*** 的意思是指向不特定的型別，所以你可以把這樣的指標的值設給「宣告成其他任意型別」的指標，譬如 **int*** 的指標。如果 **malloc()** 無法取得記憶體則會傳回 **NULL**。
(註：不管是指向哪一種型別的指標，儲存的值都是長度 4 bytes 的記憶體位址，所以這樣的設定很合理。真正的差異是當你要做類似 **p++** 這樣的指標增加的動作時，必須要知道 **p** 究竟是指向哪一種型別，才能知道每次 **++** 到底要移動多少個 bytes 的位址。你可以試試看對一個 **void*** 的指標做 **++** 的動作，compiler 可能會有會有類似 warning: wrong type argument to increment 的警告訊息。)

Dynamic Memory Allocation

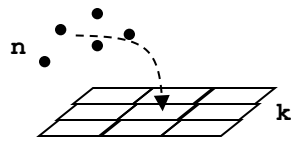
用 **malloc()** 取得記憶體，使用完後要記得釋放，不然記憶體會越用越少。下面的敘述會把剛才前面取得的記憶體釋放

```
free(newPtr);
```

可見得，記住原先取得的記憶體的起始位址，是一件非常重要的事情。要是忘了那個起始位址，到時候就無法正確釋放記憶體。
你可以試試看如果釋放一個錯誤的位址會有什麼後果，或是把已經 **free()** 的位址再 **free()** 一次看看。 

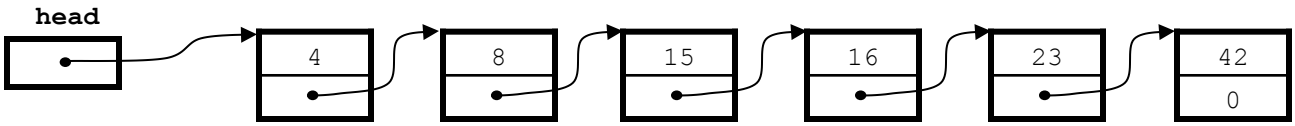
(註：執行 **free(newPtr)** 會把 **newPtr** 指到的位址傳給 **free()**，然後 **free()** 會負責把對應位址原先佔用的那些記憶體釋放掉， 但 **newPtr** 的值並不會變 (因為 call-by-value)，也就是說 **newPtr** 還是指到相同的記憶體位址，但是這塊記憶體的內容已經不再受到保護，所以不該再用 **newPtr** 去存取那塊記憶體，雖然不一定會立刻發生錯誤。更進一步說明，對大多數 **malloc()** 和 **free()** 的implementation 來說，所謂取得和釋放其實只是改變記憶體的使用狀態。**malloc()** 和 **free()** 做的事情只是去記錄哪些記憶體區塊被佔用以及哪些可用。所以，執行完 **free(newPtr)** 之後，乍看之下可能會覺得沒發生什麼變化，因為原先那些記憶體內的資料可能都還存在，但是無論如何你的程式都不該再去存取已經被 **free()** 的位址的記憶體內容。總之，使用 **malloc()** 和 **free()** 要特別小心。另外順帶提一下，**malloc()** 和 **free()** 通常就是用接下來要介紹的 linked lists 來管理記憶體。)

[練習 WE_02]
讓使用者輸入 **n** 和 **k** 兩個整數。宣告一個 **struct t_point** 包含 **int x** 和 **int y**。用 **malloc()** 產生一個型別為 **struct t_point**，長度為 **n** 的陣列，取名為 **P**。 隨機設定每個 **P[i]** 的 **x** 和 **y** (**0~k-1** 的整數，總共有 **k*k** 個格子)，把落在每個格子內的點數秀出來。最後把 **P** 佔用的記憶體釋放掉。



Linked Lists

結合前面兩項功能，self-referential structures 和 dynamic memory allocation，就可以動態地做出 linked lists 這種資料結構。



接下來要討論的問題就是如何產生一個 linked list，如何在某個位置插入一個 node，如何拿掉某個 node。假設我們希望維持 linked list 所存的資料的順序，node 的 data 值要由小排到大。
需要寫出幾個 functions:

```
struct t_node* insert(struct t_node *np, int val);
struct t_node* delete(struct t_node *np, char val);
void dispList(struct t_node *np);
```

主程式從產生 **head** 開始

```
struct t_node *head;
head = NULL;
```

呼叫 **insert()** 把某個值加入 **head** 所指到的 linked list 的適當位置 (node 的 data 值要從小排到大)

```
head = insert(head, 4); /* 經過 insert() 之後， head 可能需要指到不同位址 */
head = insert(head, 8); /* 所以要傳回新的位址給 head */
head = insert(head, 13);
```

或是呼叫 **delete()** 把值從 **head** 所指到的 linked list 中刪除

```
head = delete(head, 13);
head = delete(head, 4);
head = delete(head, 11);
```

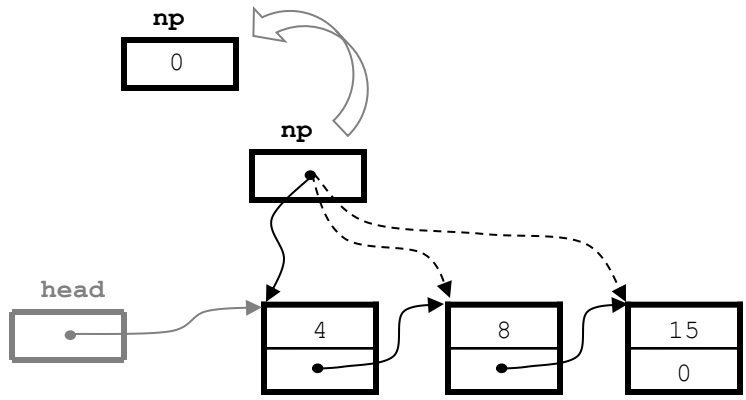
或是呼叫 **dispList()** 把 **head** 所指到的 linked list 的內容顯示出來

```
dispList(head);
```

Linked Lists

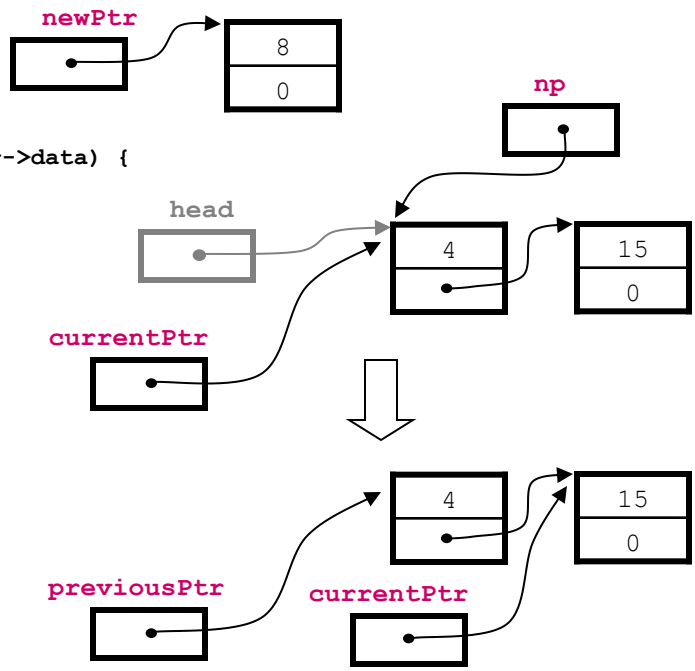
先來看怎麼寫 `void dispList(struct t_node *np);`

```
void dispList(struct t_node *np)
{
   if ( np == NULL ) {
      printf("List is empty.\n\n");
   }
   else {
      printf("The list is:\n");
      while (np != NULL) {
         printf("%d--> ", np->data);
         np = np->nextPtr;
      }
      printf("NULL\n\n");
   }
}
```

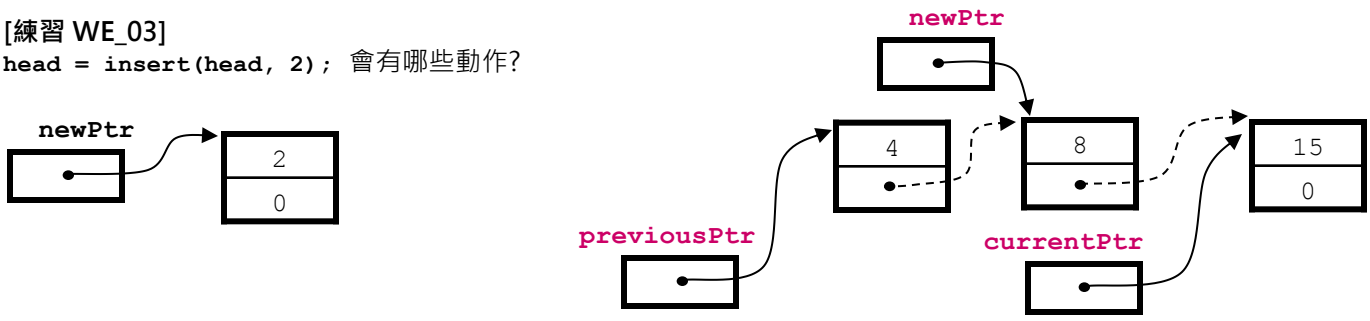


再來看 `struct t_node* insert(struct t_node *np, int val);`

```
struct t_node* insert(struct t_node *np, int val)
{
   struct t_node *newPtr, *previousPtr, *currentPtr;

   newPtr = (struct t_node *)malloc(sizeof(struct t_node));
   if (newPtr != NULL) {
      newPtr->data = val;
      newPtr->nextPtr = NULL;
      previousPtr = NULL;
      currentPtr = np;
      while (currentPtr!=NULL && val>currentPtr->data) {
         previousPtr = currentPtr;
         currentPtr = currentPtr->nextPtr;
      }
      if (previousPtr == NULL) {
         newPtr->nextPtr = np;
         np = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
      return np;
   }
   else {
      printf("記憶體不夠啦.\n");
      return NULL;
   }
}
```



[練習 WE_03]
`head = insert(head, 2);` 會有哪些動作?

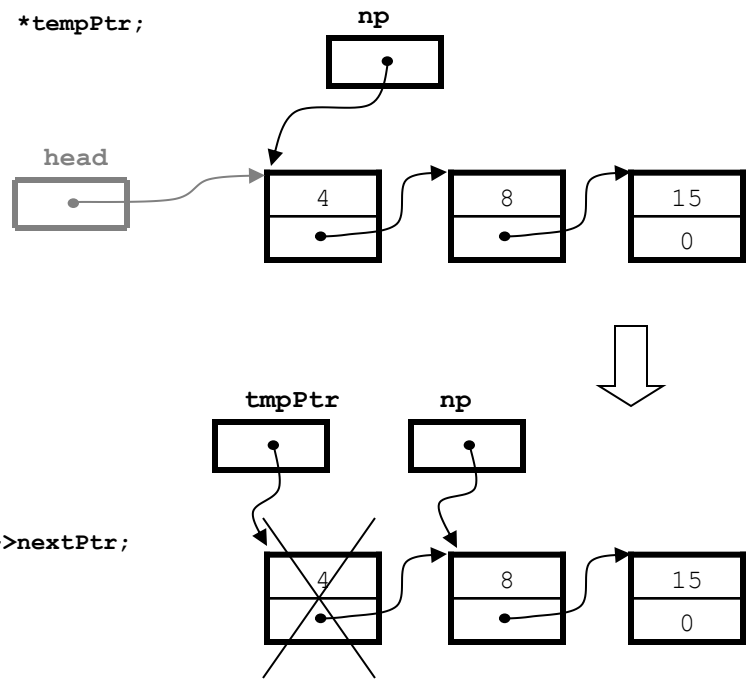


Linked Lists

最後看怎麼寫 `struct t_node* delete(struct t_node *np, int val);`

```
struct t_node* delete(struct t_node *np, int val)
{
   struct t_node *previousPtr, *currentPtr, *tempPtr;

   if (val == np->data) {
      tempPtr = np;
      np = np->nextPtr;
      free( tempPtr );
      return np;
   }
   else {
      previousPtr = np;
      currentPtr = np->nextPtr;
      while (currentPtr != NULL &&
                currentPtr->data != val) {
         previousPtr = currentPtr;
         currentPtr = currentPtr->nextPtr;
      }
      if (currentPtr != NULL) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return np;
      }
   }
   return NULL;
}
```

如果執行
`head = delete( head, 4);`



[練習 WE_04]
畫圖解釋上面的例子如果接著執行

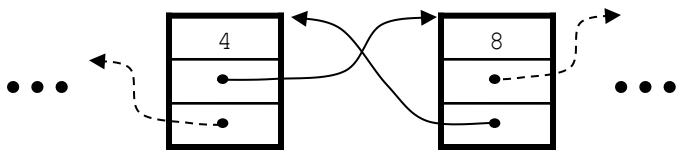
```
head = delete(head, 15);
```

以及

```
head = delete(head, 7);
```

會做哪些動作。

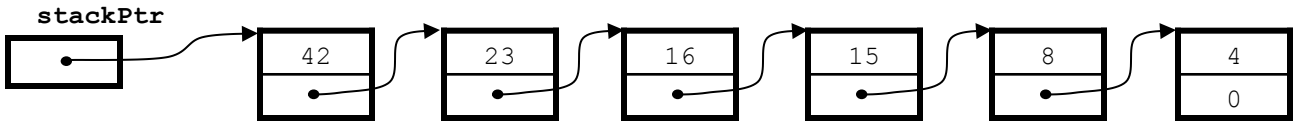
[練習 WE_05]
試著建立 doubly linked lists。



Stacks

看過 linked lists 的操作方式，再來就要介紹 stacks 這種資料結構。Stacks 是想像將一筆筆資料疊在一起，新加入的資料只能加在 stack 最上層，而且也只能從 stack 最上層移除資料，就是所謂的 last-in first-out (LIFO) 原則。可想而知，實作 stacks 的方式其實和 linked lists 一樣，還是用 **struct** 和指標的結合來做，只是對於 stacks 來說 **struct t_node** 的 **insert()** 和 **delete()** 只能發生在最前面的 node，一般習慣將這兩個動作稱作 push 和 pop。此外，和 linked lists 一樣，需要用一個指標指到 stack 的最頂端位址，而 stack 最底層的 node 的 **nextPtr** 欄位也要指到 **NULL**。先來看用下面這種 **struct t_node** 製作出的 stack:

```
struct t_node {
   int data;
   struct t_node *nextPtr;
};
```



我們需要寫出能達到 push 和 pop 兩種效果的 functions。有了製作 linked lists 的經驗，這次我們就來看看如何用 call-by-reference 來寫這兩個 functions。
為了要將 stack 的內容印出來，我們可以直接拿 **dispList()** 來修改寫出 **printStack()**。

```
void printStack(struct t_node *currentPtr)
{
   if (currentPtr == NULL) {
      printf("The stack is empty.\n\n");
   }
   else {
      printf("The stack is:\n");
      while (currentPtr != NULL) {
         printf("%d --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }
      printf("NULL\n\n");
   }
}
```

將新的資料加入 stack 需要 **void push(struct t_node * *topPtr, int val);**

```
void push(struct t_node * *topPtr, int val)
{
   struct t_node *newPtr;
   newPtr = (struct t_node *)malloc(sizeof(struct t_node));
   if (newPtr != NULL) {
      newPtr->data = val;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   } else {
      printf("%d not inserted. No memory available.\n", val);
   }
}
```

參數 **topPtr** 出現兩個星號是因為我們想要有 call-by-reference 的效果。在 **main()** 裡假設我們有一個指標指向 stack 的最上層位址:

```
struct t_node *stackPtr = NULL;
```

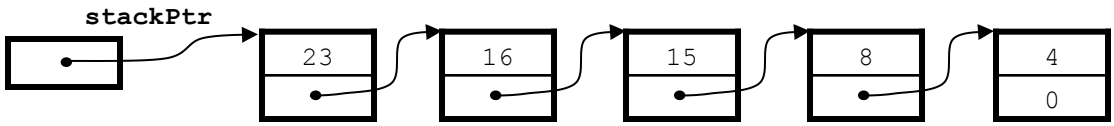
用 call-by-reference 方式呼叫，要用類似下面的方式，把 **stackPtr** 這個**指標變數**本身的位址傳給 **push()**:

```
push(&stackPtr, 42);
```

由於 **stackPtr** 已經是 **struct t_node *** 這樣的型別，所以 **&stackPtr** 的型別就是 **struct t_node **，出現兩個星號。如此一來，在 **push()** 裡面對 **topPtr** 所做的修改，才能夠真的影響到 **main()** 裡的 **stackPtr**。

push() 的示意圖

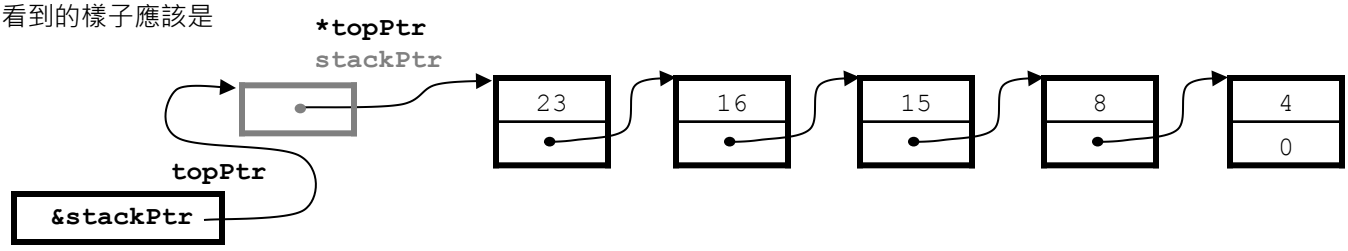
假設我們已經用了一連串 **push()** 產生了下面的 stack (這些數字是依照什麼順序 push ?) :



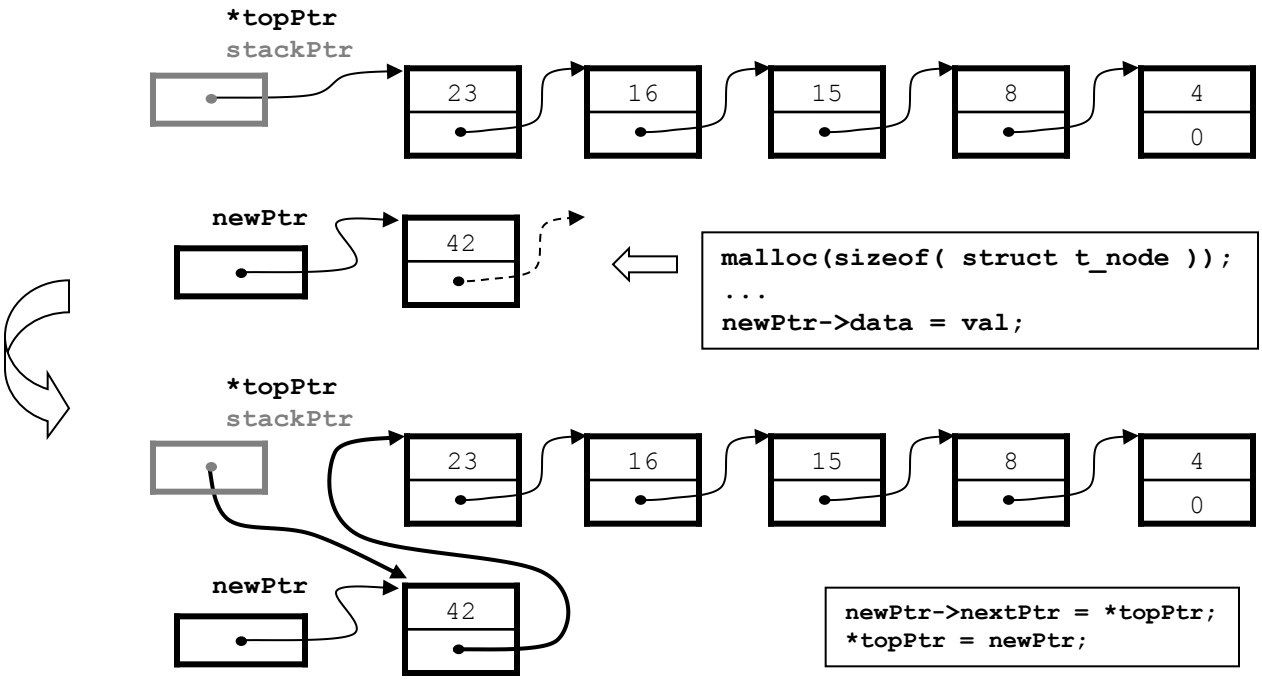
在這樣的情況下再呼叫 **push(&stackPtr, 42);** 進入

```
void push(struct t_node * *topPtr, int val)
{ ...
}
```

看到的樣子應該是

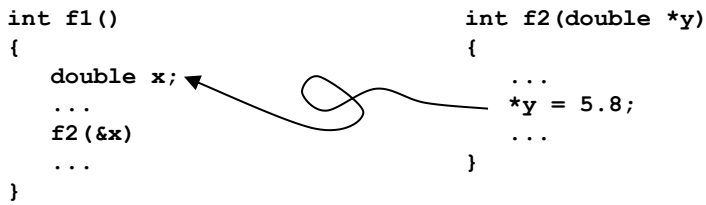


topPtr 這個指標所記錄的位址是 **&stackPtr**，所以 ***topPtr** 相當於 **stackPtr**，
記錄的是由 **struct t_node** 組成的 stack 的起始位址。



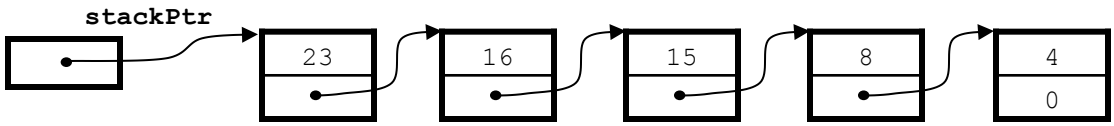
Call-by-Reference 的觀念

假設你有兩個 functions: **f1()** 和 **f2()**，在 **f1()** 裡面會呼叫 **f2()**，而 **f1()** 裡面有一個變數 **x**。如果想要在 **f2()** 裡面修改 **f1()** 的變數 **x**，就必須用 call-by-reference 的方式把 **x** 傳給 **f2()**，也就是把 **x** 的位址告訴 **f2()** 而讓 **f2()** 能直接對那個位址裡面儲存的資料做修改。



pop()

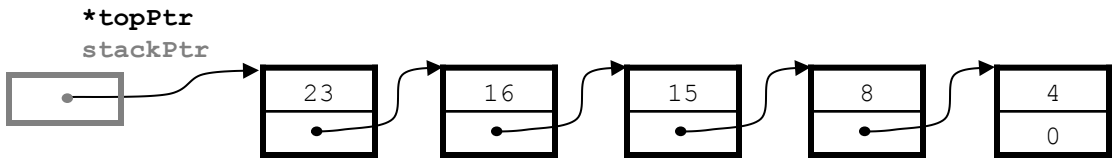
假設我們已經用了一連串 **push()** 產生了下面的 stack:



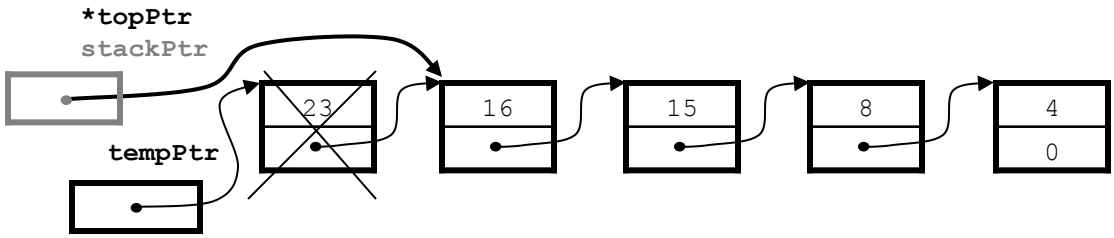
在這樣的情況下再呼叫 **pop(&stackPtr);** 進入

```
int pop(struct t_node * *topPtr)
{
   struct t_node *tempPtr;
   int popValue;
   if (*topPtr == NULL) return -1;
   tempPtr = *topPtr;
   popValue = (*topPtr)->data;
   *topPtr = (*topPtr)->nextPtr;
   free(tempPtr);
   return popValue;
}
```

在 **pop()** 裡看到的樣子應該是



執行 **pop()** 應該要傳回 **23** 這個整數，然後 **stackPtr** 要指到儲存 **16** 的 **struct t_node**，再把原本儲存 **23** 的 **struct t_node** 所佔用的空間釋放掉。



主程式的內容

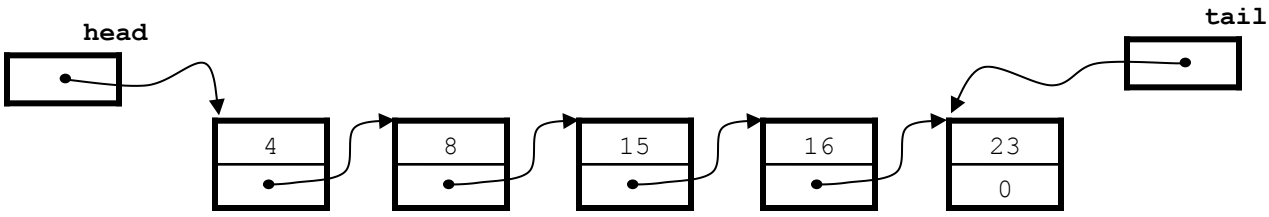
```
int main()
{
   struct t_node *stackPtr = NULL;
   int i;
   push(&stackPtr, 4);
   push(&stackPtr, 8);
   push(&stackPtr, 15);
   push(&stackPtr, 16);
   push(&stackPtr, 23);
   printStack(stackPtr);
   i = pop(&stackPtr);
   printStack(stackPtr);
   printf("%d is popped.\n", i);

   return 0;
}
```

[練習 WE_06]
把 **printStack()** 改寫成 recursive 版本。

[練習 WE_07]
使用 stack 判斷一個字串的括號是否正確對應，例如 **"(((1+2) + (3+4))*5 + (6+7)*8) * 9"**。

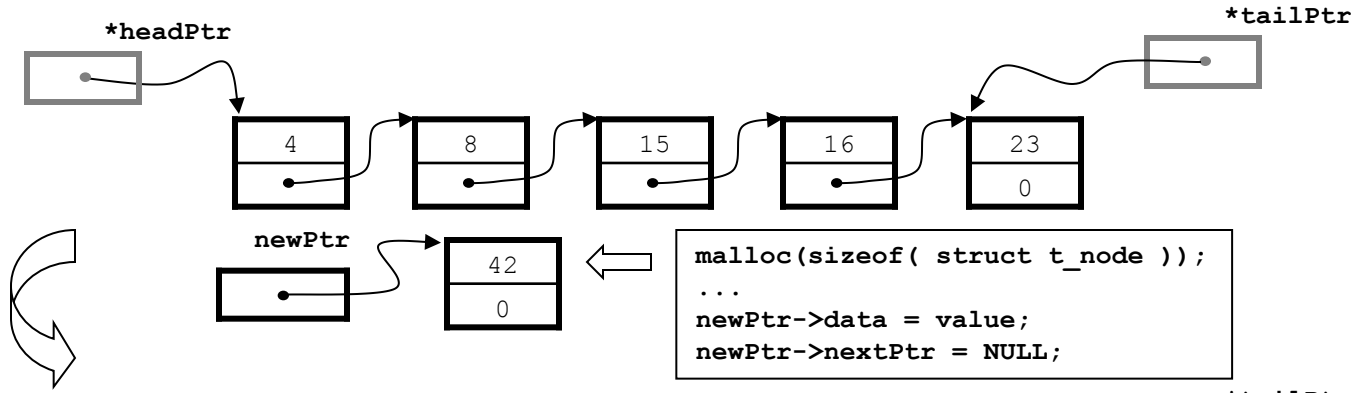
Queues

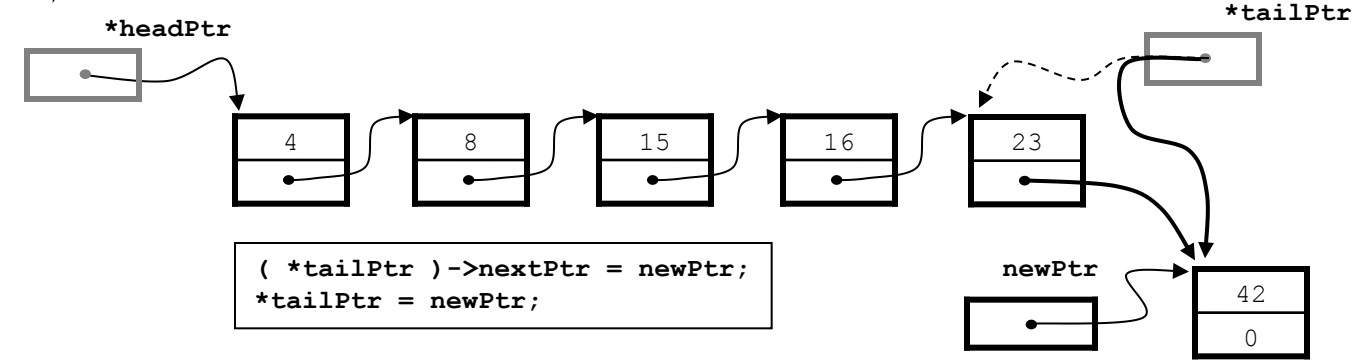
雖然 queues 和 stacks 都是用 linked lists 的形式來儲存資料，但是 queues 的操作原則是 first-in first-out (FIFO)，也就是平常的排隊原則，先排隊的先結帳，後來加入的要排在隊伍最後面。相對於 stacks 的 **`push()`** 和 **`pop()`** 這兩個動作，queues 需要 **`enqueue()`** 和 **`dequeue()`** 兩個 functions。至於 **`printQueue()`** 則和 **`printStack()`** 的方式完全一樣。除此之外，由於 stack 不論加入或移除都是從 stack 的最上端，但是 queue 則是從末端加入、前端移除，所以 queue 除了要有一個 **`head`** 指標指到 queue 的前端，還需要 **`tail`** 指標指到 queue 的末端。



在這樣的情況下再呼叫 **`enqueue(&head, &tail, 42);`**

```
void enqueue(struct t_node * *haedPtr, struct t_node * *tailPtr, int val)
{ ...
}
```

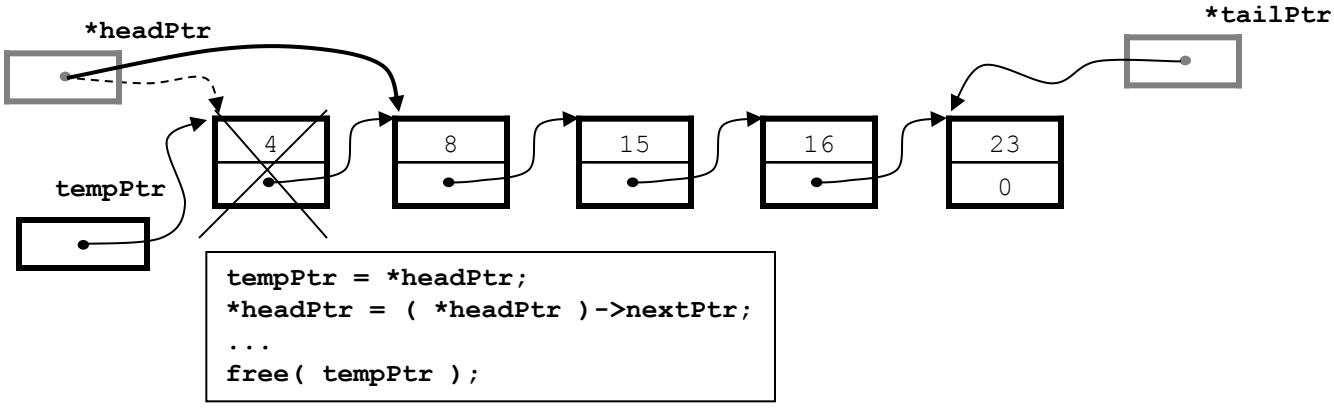





```
void enqueue(struct t_node * *headPtr, struct t_node * *tailPtr, char val)
{
   struct t_node *newPtr;
   newPtr = (struct t_node *)malloc(sizeof( struct t_node ));
   if (newPtr != NULL) {
      newPtr->data = val;
      newPtr->nextPtr = NULL;
      if (*headPtr == NULL) {    /* 假如 queue 還是空的, 則在第一個 node 被加入之後, */
         *headPtr = newPtr;      /* headPtr 和 tailPtr 會暫時都指向第一個 node     */
      } else {
         (*tailPtr)->nextPtr = newPtr;
      }
      *tailPtr = newPtr;
   } else {
      printf("%c not inserted. No memory available.\n", val);
   }
}
```

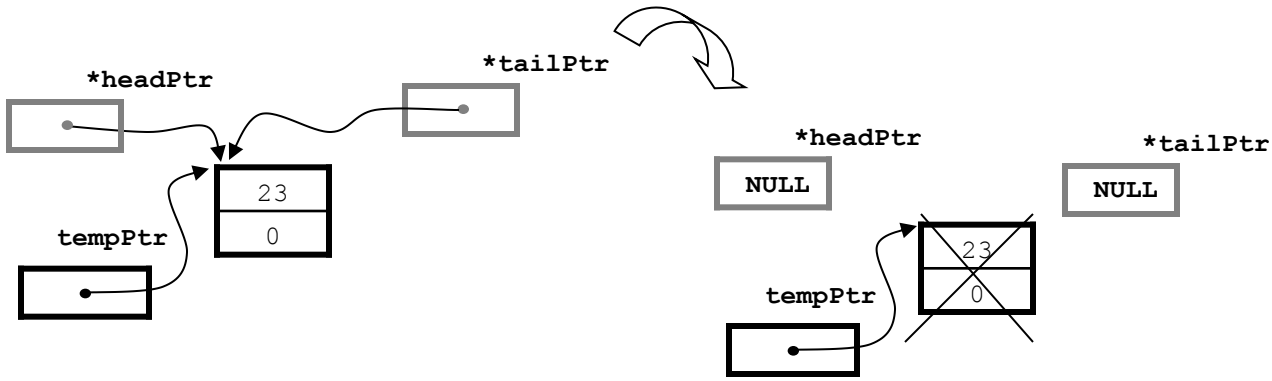
dequeue()

dequeue() 的動作很簡單：把 **head** 指標從第一個 node 移到第二個 node，然後把第一個 node 的空間釋放。同樣地，因為 **head** 指標一旦移動到第二個 node，第一個 node 的位址就沒有人記得，所以要先用 **tempPtr** 把第一個 node 的位址先記住，才能移動 **head** 指標。

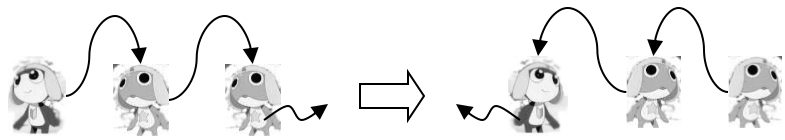



```
int dequeue(struct t_node * *headPtr, struct t_node * *tailPtr)
{
   int val;
   struct t_node *tempPtr;
   if (*headPtr == NULL) return -1;
   val = (*headPtr)->data;
   tempPtr = *headPtr;
   *headPtr = (*headPtr)->nextPtr;

   if (*headPtr == NULL) {
      *tailPtr = NULL;
   }
   free(tempPtr);
   return val;
}
```

當只剩下一個 node 的時候， **(*headPtr)->nextPtr** 指到 **NULL**，所以執行 ***headPtr = (*headPtr)->nextPtr;** 之後 ***headPtr** 指向 **NULL**，在這種情況下 ***tailPtr** 也應該指到 **NULL**，回到 empty queue 的狀態。

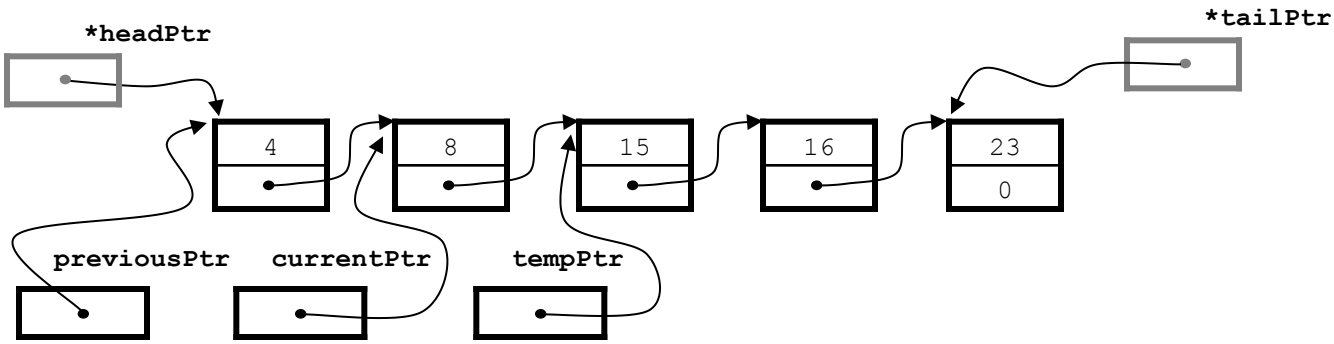


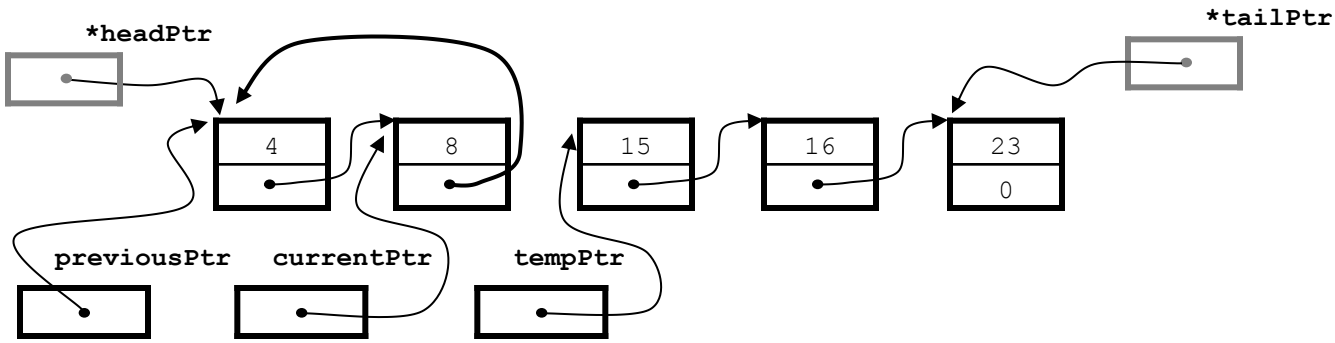
[練習 WE_08]
寫一個 function
void reverse(struct t_node * *headPtr, struct t_node * *tailPtr);
把 queue 反過來。

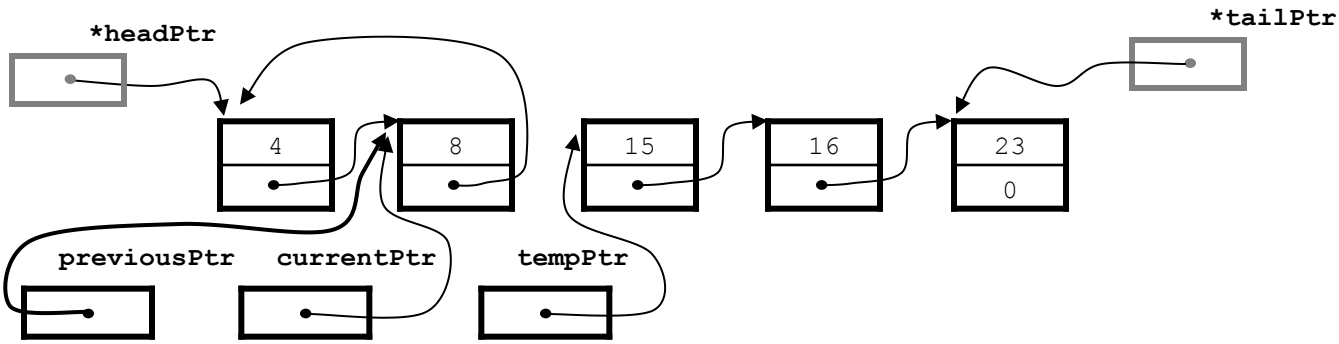


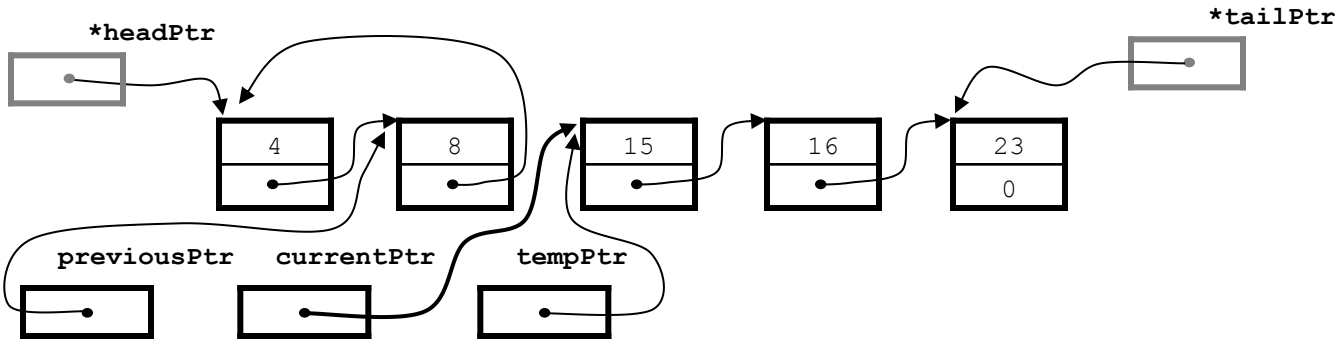
[練習 WE_08] (續)

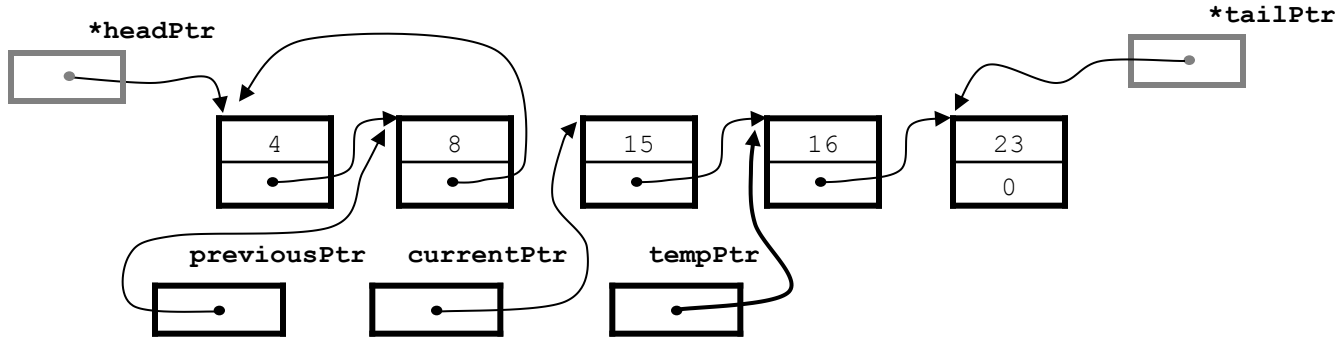
```
void reverse(struct t_node ** headPtr, struct t_node ** tailPtr)
{
   struct t_node *currentPtr;
   struct t_node *previousPtr;
   struct t_node *tempPtr;

   if (*headPtr != *tailPtr) { /* 如果 queue 只剩一個 node, 或是 queue 是空的,
                                  則 *headPtr 和 *tailPtr 會相等, 這時候並不需要做任何動作 */
      previousPtr = *headPtr;
      currentPtr = previousPtr->nextPtr; /* previousPtr、currentPtr 指到相鄰 nodes */

      while(currentPtr != *tailPtr) {
         tempPtr = currentPtr->nextPtr;     /* 先記住下一個 node 的位址            */
         currentPtr->nextPtr = previousPtr; /* 記住後更改指標 讓牠往回指           */
         previousPtr = currentPtr;          /* previousPtr 往下移動一個 node      */
         currentPtr = tempPtr;              /* currentPtr 也往下移到剛才記住的位址 */
      }
      (*tailPtr)->nextPtr = previousPtr;  /* 迴圈結束後還要再收尾
                                             把原本指到 null 的指標改指到前一個 node */
      tempPtr = *tailPtr;  /* swap 三步驟又來了 */
      *tailPtr = *headPtr;
      *headPtr = tempPtr;

      (*tailPtr)->nextPtr = NULL; /* 把新的 *tailPtr 的 nextPtr 指到 NULL */
   }
}
```

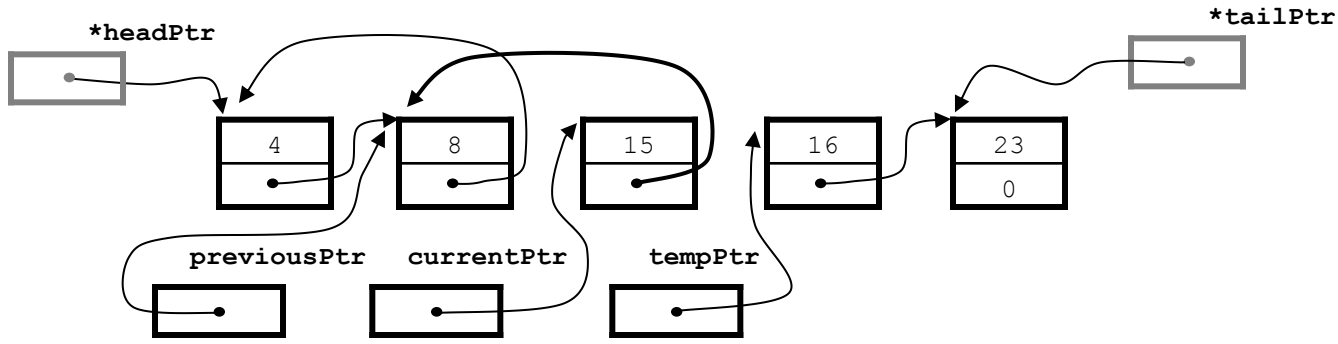





*headPtr
*tailPtr
4
8
15
16
23
0
previousPtr
currentPtr
tempPtr


*headPtr
*tailPtr
4
8
15
16
23
0
previousPtr
currentPtr
tempPtr

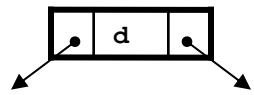

*headPtr
*tailPtr
4
8
15
16
23
0
previousPtr
currentPtr
tempPtr


*headPtr
*tailPtr
4
8
15
16
23
0
previousPtr
currentPtr
tempPtr

Trees

終於要介紹 trees 這種有趣的資料結構。大家以後學 data structures 和 algorithms 的時候會遇到各式各樣奇怪的樹，在這裡我們只簡單介紹 binary search trees (二元搜尋樹)，從名字可看出 binary search trees 其實是 binary trees 的一種特例。所謂 binary trees 是指每個 node 最多可以有兩個指標指向別的 nodes，我們必須稍微修改 **struct t_node** 的宣告:

```
struct t_treeNode {
   struct t_treeNode *leftPtr;
   int data;
   struct t_treeNode *rightPtr;
};
```



所以一個 binary tree 的每個節點都長得像 **struct t_treeNode** 這樣的結構，也就是除了有 **data** 欄位儲存資料之外，還包含了兩個指向 **struct t_treeNode** 的指標。

[回顧]
指標變數，用來記錄記憶體位址，在32-bit 定址系統下，每個指標變數都用 4 bytes 來記錄某個記憶體位址。

既然都是 4 bytes，為什麼指標變數還要有型別，譬如 **struct t_treeNode *leftptr;** ?
這是因為我們對指標變數本身並不是很感興趣，比較在乎的是指標變數所記錄的那個記憶體位址裡面所存放的資料，但是指標變數記錄的只是資料的開頭位址，到底資料本身有多大，就要靠資料的型別來決定。
另外當我們對指標變數作加減的運算時，也必須靠資料的型別才能知道 **leftPtr+1** 這樣的動作到底會移動多少位址。

當我們使用 **leftPtr** 的時候，我們得到的是 **leftPtr** 這個變數裡面的值，也就是某個記憶體位址。
當我們使用 ***leftPtr** 的時候，我們得到的是 **leftPtr** 所記錄的那個記憶體位址的記憶體裡面所存放的資料。
當我們寫 **rightPtr = leftPtr;** 我們的意思是把 **leftPtr** 所記錄的「記憶體位址」複製給 **rightPtr**，所以 **rightPtr** 和 **leftPtr** 就會記錄了相同的位址。
當我們寫 ***rightPtr = *leftPtr;** 我們的意思是把 **leftPtr** 所記錄的「記憶體位址裡面的資料」複製到 **rightPtr** 所記錄的記憶體位址的記憶體裡面。

用圖書館的比喻，假設我想介紹你看一本書，我可以告訴你請到第幾樓的第幾個書櫃的第幾層，找出第幾本書來看。另一種可能是我自己去把那本書找出來，然後影印一本 (違法，不過這只是比喻)，然後放到你的書櫃裡面。

所以當我們說把某個指標指到另一個指標所指的位址，意思是把指標變數記錄的「記憶體位址」複製給另一個指標變數。如果能牢記這一點，再回頭看看 linked lists、 stacks、 queues 這些動態資料結構的加入和刪除的動作，應該會比較容易想通。

我們已經學過使用 **malloc()** 來取得記憶體，用 **free()** 來釋放記憶體。假設你用 **malloc(sizeof(int)*80);** 取得了一大塊記憶體，你要怎麼使用這塊記憶體呢? 他並沒有一個像是 **a[]** 或 **x** 之類的名字，那要怎麼把資料存到這塊記憶體中，或是到底該怎麼知道這塊記憶體在哪裡。由此可知 **malloc()** 必須傳回這塊記憶體的起始位址，這樣你就可以用一個**指標變數**把這個位址記起來，然後就可以透過這個指標找到這塊記憶體，然後存取裡面的資料。釋放記憶體也是一樣，只要告訴 **free()** 你想要釋放哪個位址的記憶體，譬如 **free(leftPtr);**，它就會找到那個位址，然後依照當初 **malloc()** 取得記憶體時留下的記錄，把該位址之後正確大小的記憶體區塊釋放。

Binary Search Trees

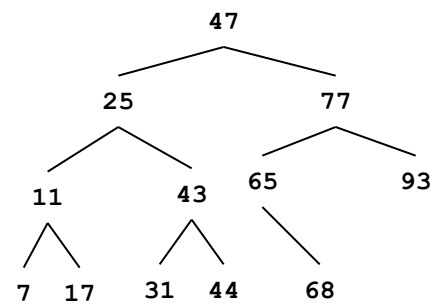
在開始寫程式產生 binary search trees 之前，先來看一個實際例子，並且了解一下它的定義。
1. 最上面的那個 node 稱作 root；root 只會指別人，不會被別人指。

2. 最下面的那些 node 稱作 leaf nodes；leaf nodes 只會被指，不會指別人

3. 相連的 nodes 之間的關係: 指別人的稱作 parent，被別人指的稱作 child。

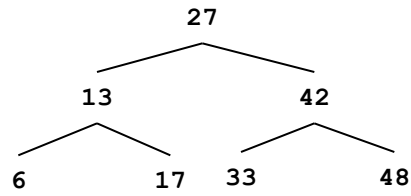
4. 每個 node 左右兩邊連著的
通常被稱作 left subtree 左子樹和 right subtree 右子樹。

5. Binary search trees 的特性是對任意一個 node 來說
牠的左子樹每個 node 的值都比牠小，而右子樹每個 node 的值都比牠大。



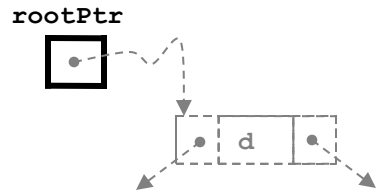
如何走訪一個 Binary Search Tree 的每一個 Node ?

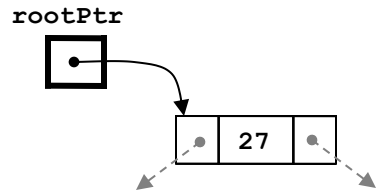
```
preorder  (中左右): 27 13 6 17 42 33 48
inorder   (左中右): 6 13 17 27 33 42 48 (相當於做排序)
postorder (左右中): 6 17 13 33 48 42 27
```



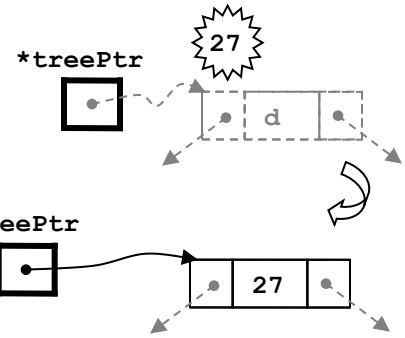
[bst.c]

```
#include <stdio.h>
#include <stdlib.h>
struct t_treeNode {
   struct t_treeNode *leftPtr;  /* 指向左子樹 (記錄左子樹的 root 的位址) */
   int data;
   struct t_treeNode *rightPtr; /* 指向右子樹 (記錄右子樹的 root 的位址) */
};
typedef struct t_treeNode  TreeNode; /* 為了方便 替 struct t_treeNode 取個新名字 */
typedef TreeNode *    TreeNodePtr;   /* 替 TreeNode* 也取一個新名字，可以避免使用兩個星號 */

void insertNode(TreeNodePtr *treePtr, int value); /* 參數使用 call-by-reference */
void inOrder(TreeNodePtr treePtr);
void preOrder(TreeNodePtr treePtr);
void postOrder(TreeNodePtr treePtr);

int main( void )
{
   TreeNodePtr rootPtr = NULL;

   insertNode(&rootPtr, 27);
   insertNode(&rootPtr, 42);
   insertNode(&rootPtr, 13);
   insertNode(&rootPtr, 6);
   insertNode(&rootPtr, 33);
   insertNode(&rootPtr, 48);
   insertNode(&rootPtr, 17);

   printf("\n\nThe preOrder traversal is:\n");
   preOrder(rootPtr);
   printf("\n\nThe inOrder traversal is:\n");
   inOrder(rootPtr);
   printf("\n\nThe postOrder traversal is:\n");
   postOrder(rootPtr);
   return 0;
}
```

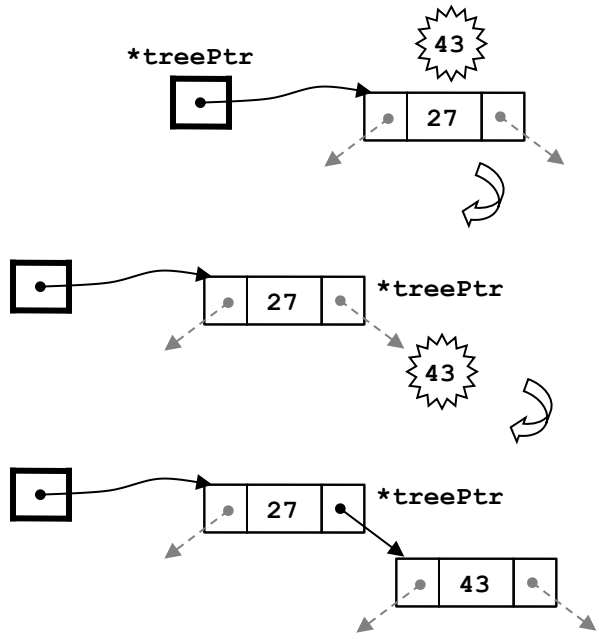




```
/* 用 recursive 方式加入新的 node */
void insertNode(TreeNodePtr *treePtr, int value)
{
   if (*treePtr == NULL) { /* *treePtr 記錄的位址是 NULL */
      *treePtr = (TreeNodePtr)malloc(sizeof( TreeNode ));
      if (*treePtr != NULL) {
         (*treePtr)->data = value;
         (*treePtr)->leftPtr = NULL;
         (*treePtr)->rightPtr = NULL;
      }
      else {
         printf("%d not inserted. No memory available.\n", value);
      }
   } else {
      if (value < (*treePtr)->data) {
         insertNode(&((*treePtr)->leftPtr), value);
      }  else if (value > (*treePtr)->data) {
         insertNode(&((*treePtr)->rightPtr), value);
      }
      else { /* 如果要放入的值重複 (已經在 tree 裡面) 就不再放入 */
         printf("重複");
      }
   }
}
```




```
/* 用 recursive 方式走完一個 tree */
void inOrder(TreeNodePtr treePtr)
{
   if (treePtr != NULL) {
      inOrder(treePtr->leftPtr);
      printf("%3d", treePtr->data);
      inOrder(treePtr->rightPtr);
   }
}

void preOrder(TreeNodePtr treePtr)
{
   if (treePtr != NULL) {
      printf("%3d", treePtr->data);
      preOrder(treePtr->leftPtr);
      preOrder(treePtr->rightPtr);
   }
}

void postOrder(TreeNodePtr treePtr)
{
   if (treePtr != NULL) {
      postOrder(treePtr->leftPtr);
      postOrder(treePtr->rightPtr);
      printf("%3d", treePtr->data);
   }
}
```



[練習 WE_09]

前面介紹的 binary search tree 不允許重複的值出現，每個 node 存放的值都不一樣。如果要把它改成可以允許重複的值，可以在 **struct t_treeNode** 多加一個欄位來記錄重複的次數，這樣做會多佔用一些空間，你可以想想看有沒有其他作法。另外還有一個問題是如何寫 **deleteNode()**，可以傳入某個值，把存放這個值的 node 刪除，

```
int deleteNode(TreeNodePtr *treePtr, int value);
```

關鍵在於如何把斷掉的樹接起來。