心得:

心得

CH1

在看完教學影片後讓更深刻的了解遞迴的運作，老師舉了許多的實例，收穫良多。

CH2.

在大一時就有聽過抽象化，卻不知道是什麼意思，在老師的講解後終於知道意思了，也懂了如何使用 class.

CH3.

在大一的時候最害怕碰到的就是 linked-list 了，老師細心的講解用列車、門牌等來比喻讓我輕易的了解如何正確的使用。

CH4.

以前都沒有聽過前中序的觀念，讓我知道原來有別的運算方式，可以不用考慮括弧、沒有結合律等的優勢使用前、後序。

CH1.

以遞迴找第k小的值： 樞紐

第1章

想法：

找一個字當樞紐 $\boxed{<p \mid p \mid >p}$

使用二分法

找其中一邊

48 and 39 大於 30 → 放到最右邊

(排序)

ex. k=4

30 7 25 39 19 48 || 2 16 12

因30的位置>4 → 後面的值不用看

12 7 25 || 19 2 16 30 ~~39 48~~

可找到位置

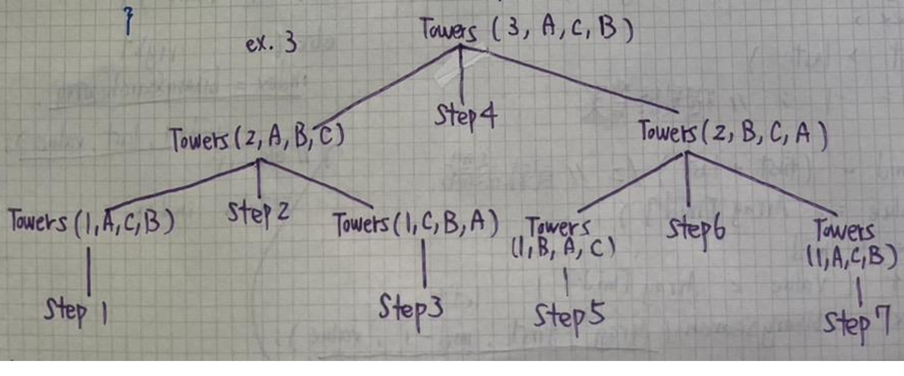
7 2 12 16 19 25 30 39 48

```
kSmall ( k : integer, anArray : ArrayType, first : integer, last : integer ) : ValueType
  if ( k < pivotIndex - first + 1 )
    return kSmall ( k, anArray, first, pivotIndex - 1 )   // 前半 擇一
  else if ( k == pivotIndex - first + 1 )
    return p
  else
    return kSmall ( k - (pivotIndex - first + 1), anArray, pivotIndex + 1, last )   // 後半
```

河內塔：

```
solveTowers ( count, source, destination, spare )
             個數    起點     終點         輔助
  if ( count == 1 )
    print ( "Move from", source, "to" destination )
  else {
    solveTowers ( count - 1, source, spare, destination ) ;
    solveTowers ( 1, source, destination, spare ) ;
    solveTowers ( count - 1, spare, destination, source ) ;
  }
```

ex. 3



Binary recursion

求最大公因數：

```
gcd1 ( int x , int y ){
  if ( y == 0 )      if(x<y)次數與2一樣
    return x ;
  else if ( y > x )
    return gcd1 ( x, y%x )
  else
    return gcd1 ( y, x%y )
} // gcd1()
```

if(x>=y)效率較高(遞迴次數較少)

```
gcd2 ( int x , int y ){
  if !( x%y )
    return y ;
  else
    return gcd2 ( y, x%y ) ;
} // gcd2()
```

ex. x = 9, y = 6

gcd1 (6, 3) = 3
↓
gcd1 (3, 0) = 3
↓
x = 3 , y = 0
return 3

gcd2 (6, 3) = 3
↓
x = 6 , y = 3
return 3

Binary search with an Array

想法： if (Array 只有一個值)
　　那個值就是最大；
　else if (Array 不只一個值)
　　max Array (Array) is the maximum of
左右都要找 { max Array (left 一半的 array) 和
　　　　　　max Array (right 一半的 array)

ex. maxarray [1, 6, 8, 3]
return max (maxArray [1, 6] , maxArray [8, 3])

maxArray [1, 6]
return (maxArray [1], maxArray [6]

maxArray [8, 3]
return (max[8], max[3]

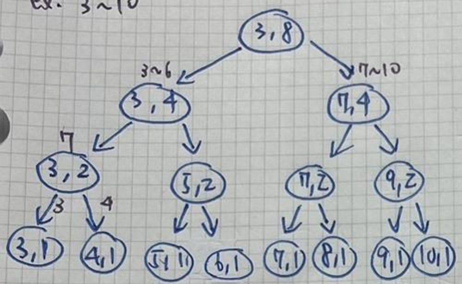
maxArray [1] return 1
maxArray [6] return 6
maxArray [8] return 8
maxArray [3] return 3

```
int binarySearch ( const int Array[ ] , int first , int last , int value ){
 int index ;
 if ( first > last )
   index = -1 ;  // 提前終答案
 else {
   int mid = ( first + last ) /2 // 設定終點
   if (value == Array [mid] )
     index = mid
   else if ( value < Array [mid] )
     index = binarySearch ( Array , first , mid - 1 , value );   left
   else
     index = binarySearch(array , mid+1 , last , value );   right
 } // else if
 return index ;
```

Binary recursion

```
int sumB( int a, int n)
if (n ==1)
  return a;
                 7     3
return sumB(a, n/2) + sumB(a+n/2, n-n/2)
```

ex. 3~10



費式數列 :

二元 I

n^k at least double every other time

$n_k \ge 2^{k/2}$, It is exponential (呼叫次數以指數成長)

(5,3) ← (3,2) ← (2,1) ← (1,1) ← (1,0)

ex. 5 → 4 → 3 → 2 → 1

```
Fibonacci( k )
if(k=1)
  return (k, 0)   // base case: k=1 → (F1, F0)
else
  (i, j) = Fibonacci(k-1) // (Fk-1, Fk-2)
  return (i+j, i)   // (Fk = Fk-1 + Fk-2 , Fk-1)
```

(呼叫次數以線性成長)

CH2.

第2章：

ADT (Abstract Data Type)　　抽象化：不須要知道裡面的細節，直接拿來用

高內聚：highly cohesive modules desired

低耦合：Loosely coupled modules desired

Concept：

描述：只須看懂用途就可使用 (Specifications)

實作：製造的人才須要知道如何實作 (Implementation)

Data Abstraction：

Asks you to think what you can do to a collection of data independently of how you do it.（不須管如何達成目的）

ADT is composed of：

A collection of data

A set of operations on that data.

Specifying ADTs：Grocery list

Except for the first and last items in a list, each item has a unique predecessor (先行者) and a unique successor (後繼者)

Operation Contract for the ADT List

插入 insert (int index : integer, in newItem : ListItemType, out success : boolean)

刪除 remove (in index : integer, out success : boolean)

檢索 retrieve (in index : integer, out dataItem : ListItemType, out success : boolean)

reserveList (in aList : List, out source : boolean)　反轉整個序列

```
for ( i=1 to aList.getlenghth() - 1 ) {
    aList.retrieve(1, dataItem, success);
    aList.remove(1, success);
    aList.insert(aList.getlength() - i + 2, dataItem, success);
}//for
```

先刪後插

The ADT Sorted List :

Maintain items in sorted order

Inserts and deletes items by their values, not their positions

新增 sortedInsert(in newItem : ListItemType , out success : boolean)

移除 sortedRemove(in index : integer , out success : boolean)

檢索 sortedRetrieve(in index : integer , out dataItem : ListItemType , out success : boolean)

定位 locatePosition(in anItem : ListItemType , out isPresent : boolean) : integer {query}

return 位置

自動維護順序(不需要知道順序)

C++ Class :

An object is an instance of a class.

a class defines a new data type.

A class contains data members and methods 成員

By default , all members in a class are private

※ You can specify them as public

Constructors :

Create and initialize new instances of a class

Have the same name as the class.

No return type.

Destructors :

Destroys an instance of an object when the object's liftime ends.

CH3.

第3章

linked list

int x ;

p = new int ; 配製 (Dynamic allocation 動態配製)

p = & x → 房子 x 的門牌 (address)

delete p ;

p = NULL ; 徹底刪除

(a) 申請空白門牌 int *p , *q ;
int x

(b) 抄寫別人的門牌 p = & x ;

(c) 鳩佔鵲巢 *p = 6 ;

(d) 配製 p = new int ;

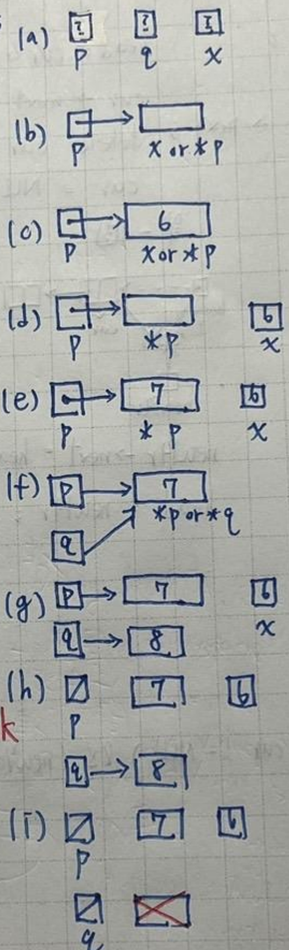
(e) 堆放 *p = 7 ;

(f) copy 另一張門牌 q = p

(g) q 配製並堆放 q = new int ;
*q = 8 ;

~~(h)~~ 刪除 p = NULL ; X memory would leak

(i) 刪除 delete q ;
q = NULL ;



動態陣列配製

ex. int arraysize =50 ;

double * anArray = new double[arraysize] ;

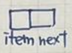
delete[ ] oldArray ;

a node in a linked list usually a struct

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

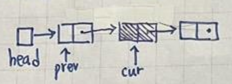
→

```
Node *p;
p = new Node;
```



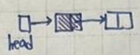
配製新的

Delete : 中間




```
prev -> next = cur -> next;
        or
prev -> next = prev -> next -> next;
```

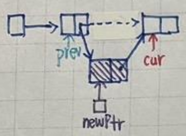
第一個




```
head = cur -> next;
cur -> next = NULL;
delete cur;
cur = NULL;
```

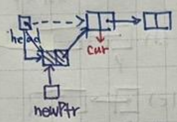
Insert :

中間



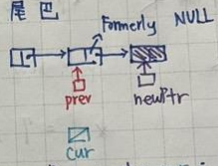

```
newPtr -> next = cur;
prev -> next = newPtr;
```

第一個




```
newPtr -> next = head;
head = newPtr;
```

尾巴



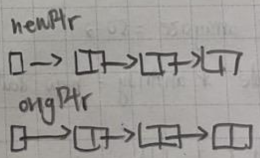

```
newPtr -> next = cur;
prev -> next = newPtr;
```

Visting : (走訪)

```
Node * prev, * cur;
for ( prev = NULL, cur = head, (cur != NULL) && (newValue > cur->item)){
  prev = cur;
  cur = cur -> next;
}
```

```
List :: List ( const List & aList )
  : size ( aList. size )
{
if ( aList.head == NULL ) // original is empty
  head = NULL ;
else {
  head = new ListNode ;
  head -> item = Alist.head -> item ;
  ListNode * newPtr = head ;
  for ( ListNode * origPtr = aList.head -> next ; origPtr != NULL ; origPtr = origPtr -> next )
  newPtr -> next = new ListNode ; newPtr = newPtr -> next ; newPtr -> item = origPtr -> item ;
```

(Deep copy)



continue →

```
} //end for
newPtr → next = NULL ;

find ( int index ) const {
  if ( (index < 1 ) || (index > getLength() ) )
    return NULL ;
  else {
    ListNode * cur = head ;
    for ( int skip = 1 ; skip < index ; skip++ )
      cur = cur → next ;
    return cur ;
  } // else
} // find()
```

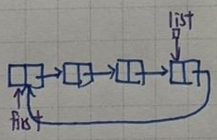
Array and 動態陣列 (linked list) 的比較

Array :	size 固定	linked list :	size 不定
	較省空間		較花空間
	retrieve 較慢		retrieve / insert / delete
	insert / delete 效率較差		較有效率

Circular linked list :

從最後移到最前面

```
if ( list != NULL ){
  Node * first = list → next ;
  Node * cur = first
  do{
    show ( cur → item )
    cur → cur → next
  } while ( cur != first );
}
```



Dummy head Node :

★ Eliminate special case

```
for ( prev = head , cur = prev → next ; (cur != NULL) &&
    (newValue > cur → item) ;
  prev = cur ;
  cur = cur → next ;
if (cur != NULL){      Delete
  prev → next = cur → next ;
  cur → next = NULL
  delete cur
  cur = NULL ;
}
```

Insert

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

CH4.

第4章

Infix expressions (中序)

ex. a + b , ((a+b)*c)/d

Prefix expressions (前序) 特性：一個前序式後面再接上非空字串一定不是前序式

ex. +ab , /*+abcd

Postfix expressions (後序)

ex. ab+ , ab+c*d/

前序.後序的 advantages：

沒有優先權

沒有結合律

沒有括弧

順序是唯一的

Grammer：

中序：<infix> = <identifier> | <infix><operator><infix>

<operator> = + | - | * | /

<identifier> = a | b | c | ... | z

前序：<prefix> = <identifier> | <operator><prefix><prefix>

<operator> = + | - | * | /

<identifier> = a | b | ... | z

後序：<postfix> = <identifier> | <postfix><postfix><operator>

<operator> = + | - | * | /

<identifier> = a | b | ... | z

Backtracking(持續尋找路不通就跳回再找)