筆記10927260_劉宗諺

單元6

原理與堆疊很類似

堆疊與佇列基本上都是儲存一堆資料，只不過會限制你使用的方式

基本上queue會從一群資料找出兩個主要的對象，一個是最前面(front)，一個是最後面(rear or back)。

若你要存取資料，就從第一筆資料(front)開始取。

Queue的另外一個稱呼叫做排隊。把持先近來先服務的形式。

有跟排隊這個現象符合的，都可以使用Queue來處理

模擬:

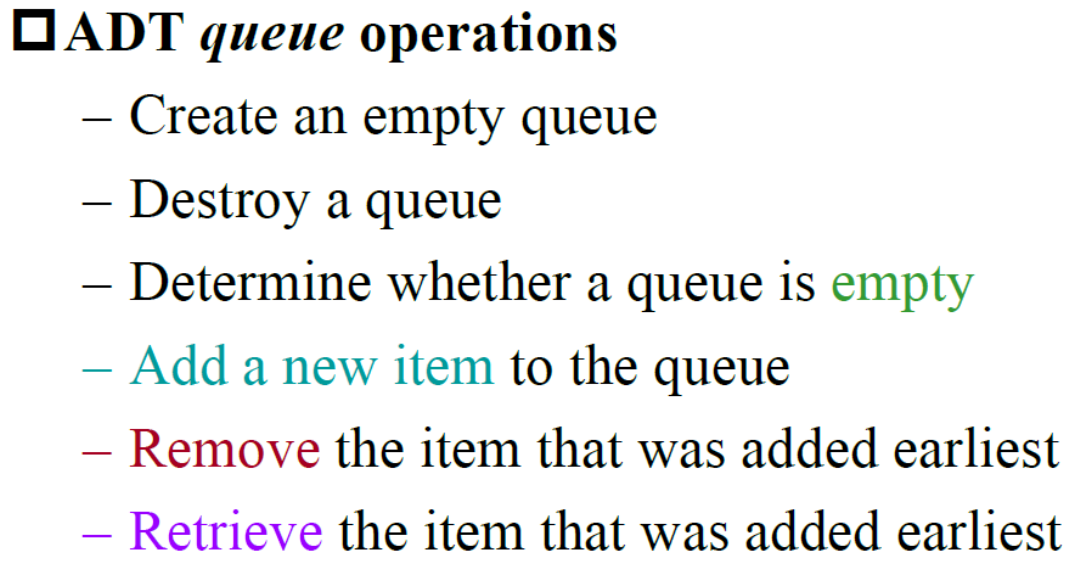
使用電腦把資料產生出來，希望這些產生出來的資料跟大自然或是人類的行為很像。這樣一個過程就稱為模擬。

這個過程中，這個資料都有一個順序，希望這個順序是保持不變的時候，我們就可以使用Queue來記錄。

ADT Queue operations

Queue比較特別的地方在於刪除或是取得取得一筆資料都是對最開始的資料做動作

只有Remove、Retrieve這兩個運算是跟stack的是不一樣的



常用的運算名稱

dequeue通常具備傳回值的功能

```
isEmpty():boolean {query}
enqueue(in newItem:QueueItemType)
        throw QueueException
dequeue() throw QueueException
getFront(out queueFront:QueueItemType)
   {query} throw QueueException
dequeue(out queueFront:QueueItemType)
        throw QueueException
```

把5、2、7依序加入Queue裡面，然肉當我使用getFront()的時候，我就會得到5(第一筆資料)

當我做dequeue的時候，因為getfront並不會把第一筆資料刪掉。所以這邊呼叫dequeue會把5(第一筆資料)刪掉，而後面的2會變成第一筆資料。

再呼叫一次就是把2刪掉

enqueue做的事就是把資料加到Queue的最後面，也就是back(rear)的後面，就像排隊一樣，排隊只能排在後面，不能插隊到前面

Operation	Queue after operation
	front
aQueue.createQueue()	
aQueue.enqueue(5)	5
aQueue.enqueue(2)	5 2
aQueue.enqueue(7)	5 2 7 ← back
aQueue.getFront(queueFront)	5 2 7 (queueFront is 5)
aQueue.dequeue(queueFront)	2 7 (queueFront is 5)
aQueue.dequeue(queueFront)	7 (queueFront is 2)

基本上在程式上面，我們會用比較熟悉的方式來去描述類似的現象

Queue應用: Reading a string of characters

一個字傳奇時就是排隊，所以我們自然可以使用Queue

一開始我們先create一個Queue

```
aQueue.createQueue(); // create a queuen
```

接下來再做enqueue

```
while ( not end of line ) {
  read a new character ch
  aQueue.enQueue( ch ); // push a char to Queue
} // while
```

我做完這些事就可以做到其他的事，ex: 把字串轉成數值

以247為例

247 = (2 * 10 + 4) * 10 + 7

```
do {
  aQueue.deQueue(ch);
} while( ch is blank ) // while

n = 0;
done = false;
while ( !done and ch is digit ) {
  n = n * 10 + integer represented by ch;
  if (aQueue.isEmpty())
    done = ture;
  else
    aQueue.deQueue(ch);
} // while
```

有小數點就只是把小數點給考慮進去而已

e.g. 247.35 = ((((2 * 10 + 4) * 10 + 7)* 10 + 5)* 10 + 3) + (0 .1)^2

```
do {
  aQueue.deQueue(ch);
} while( ch is blank ) // while

n = 0;
done = false;
while ( !done and ch is digit ) {
  n = n * 10 + integer represented by ch;
  if (aQueue.isEmpty())
```

```
    done = ture;
  else
    aQueue.deQueue(ch);
} // while

if ( !done and ch == '.' ) {
  // 有小數點繼續往下做
  aQueue.deQueue(ch);
  p = 0;
  while(!done and ch is digit) {
    n = n * 10 + integer of ch;
    p++;
    if (aQueue.isEmpty()) {
      done = true;
    } // if
    else
      aQueue.deQueue(ch);
  } // while
  n = n * (0.1)^p;
} // if
```

大數問題?

Queue應用2:回文

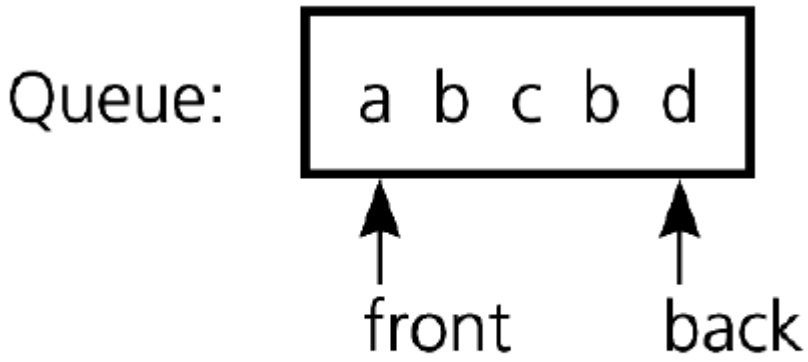
左邊念過來=右邊念過來

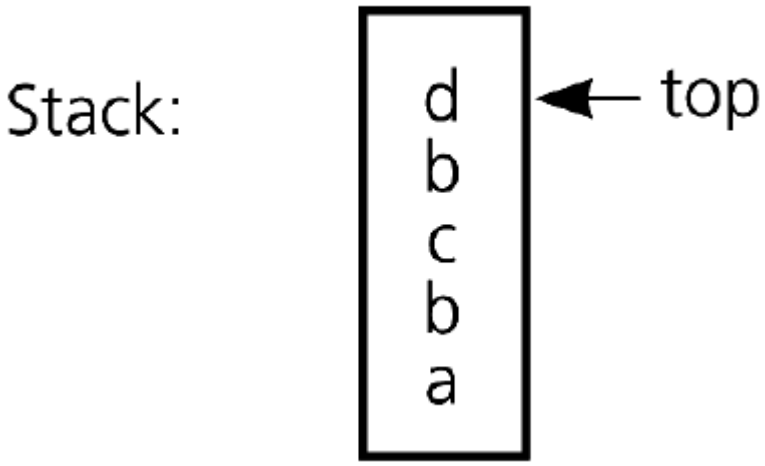
可以使用堆疊和佇列

如果你的資料結構可以幫你紀錄資料的順序，你就可以不用使用遞迴

最主要是比Queue的第一個字元和stack的top

因為top是紀錄最後一個字，而front是紀錄第一個字，所以我只要檢查字串的第一個字和最後一個字是不是一樣就好了






```
bool isPal() {
  aQueue.createQueue();
  aStack.createStack();
  for (next char ch in str) {
    // store ch into queue and stack;
    aQueue.enqueue(ch);
    aStack.push(ch);
  } // for
  charEqueal = true; // 檢查是不是找到一樣的 是false or true與問題有關
  while (Queue is not empty) {
    compare front and top;
  } // while

  while (!aQueue.isEmpty() && charEqual) {
    aQueue.dequeue(front);
    aStack.pop(top);
    if ( front != top )
      charEqual = false;
  } // while
} // isPal()
```

實作Queue

直覺: 使用pointer實現

對於Queue來說用linked list來實現是比較直覺的

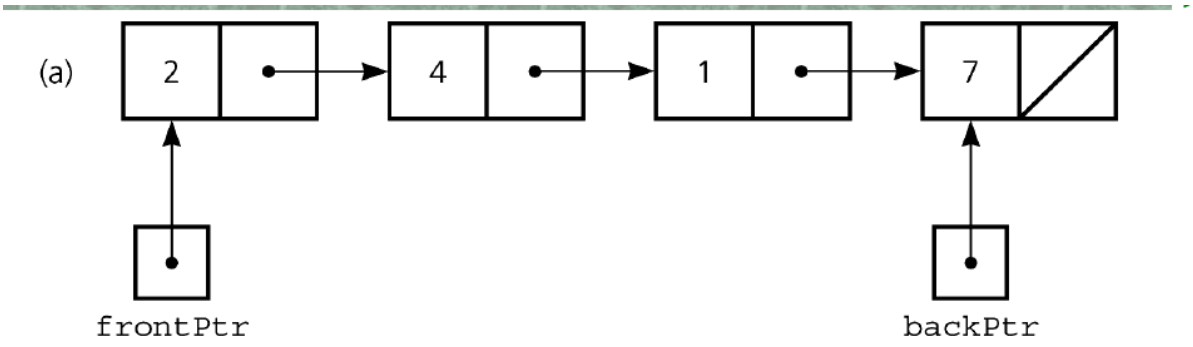
對於front: dequeue是對front做dequeue，de=delete

對back的操作是enqueue，en是指enter

我們實作一個Queue可以使用一般的linked list或是環狀的linked list

環狀的linked list來實作的優點就是他只有一個變數

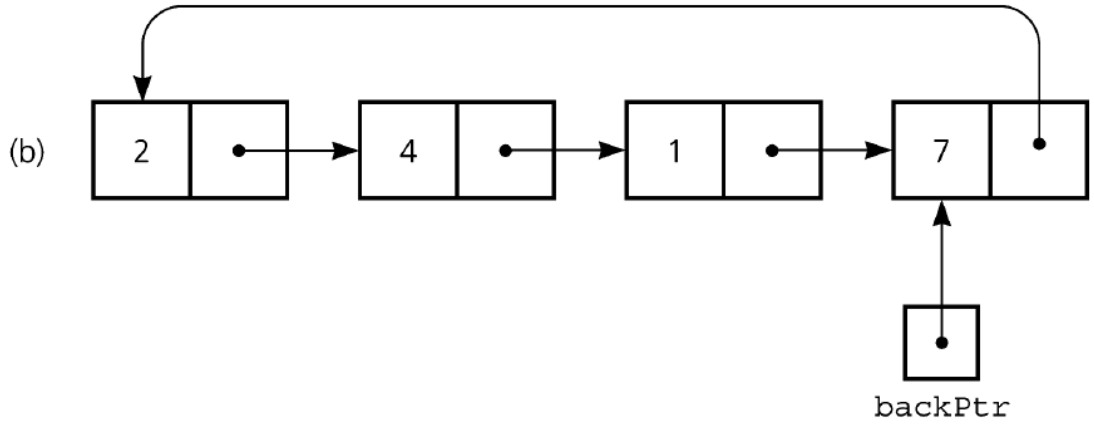
第一種Queue:



第二種Queue: circular

只要一個pointer紀錄尾巴

第一個資料2:就是backPtr→next



使用pointer來實作queue是比較不會有問題，但是使用array來實作queue會出現一些問題

```
class Queue {
  public:
    Queue();
    Queue(const Queue& Q)l
    ~Queue();

    bool isEmpty() const;
    void enqueue(const QueueItemType& newItem);
    void dequeue();
    void dequeue(QueueItemType& queueFront);
    void getFront(QueueItemType& queueFront) const;
  private:
    struct QueueNode {
      QueueItemType item;
      QueueNode* next;
    }; // QueueNode
    QueueNode* backPtr;
    QueueNode* frontPtr;
} // end Queue
```

Queue記錄了兩個資訊(pointer)，因為每個pointer是4 bytes，所以總共只要8個byte就可以把這個物件記錄下來

enqueue實作

新增都固定在尾巴，我只需要分兩個case來處理他

1. Queue是空的
2. Queue有東西

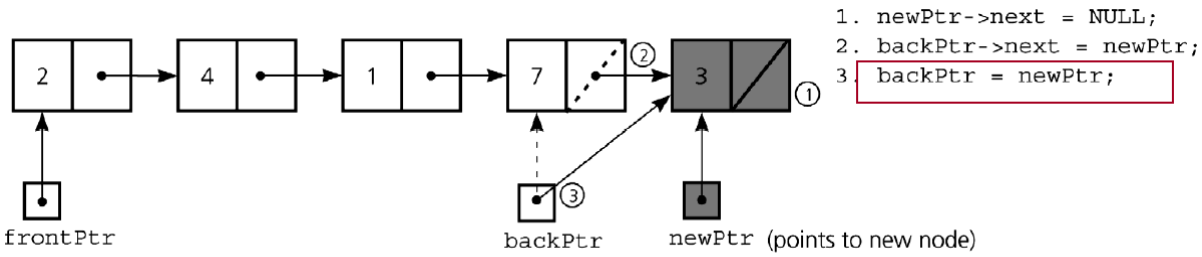
資料不是空的

我要做的是把backPtr的next指向新的pointer(newPtr)

```
backPtr->next = newPtr
```

再把backPtr移到newPtr

```
backPtr = newPtr;
```

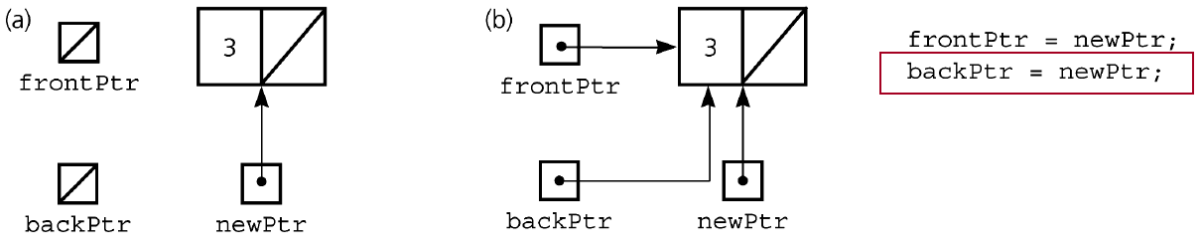


資料是空的

首先我的front跟back都是NULL

所以在空的時候，我新增資料會動到front

```
frontPtr = newPtr;
backPtr = newPtr;
```



Queue constructor

```
Queue::Queue():backPtr(NULL), frontPtr(NULL){
}
```

```
void Queue::enqueue(const QueueItemType& newItem) {
  QueueNode* newPtr = new QueueNode;
  newPtr->item = newItem;
  newPtr->next = NULL; // 接到最後
  if (isEmpty())
    frontPtr = newPtr; // 若Queue是空的，要動到front
  else
    backPtr->next = newPtr; // 把newPtr接到backPtr後面

  backPtr = newPtr; // 更新backPtr
} // enqueue()
```

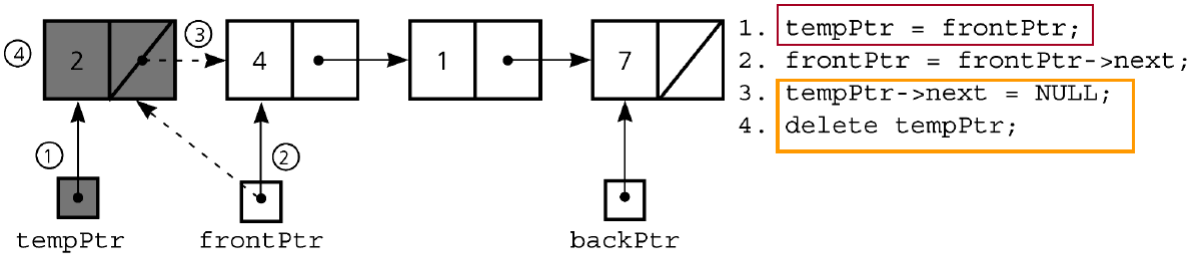
dequeue實作:

一樣分成兩個case

1. 後面有資料
2. 空的

有資料

```
tempPtr = frontPtr;
frontPtr = frontPtr->next;
tempPtr->next = NULL;
delete tempPtr;
```

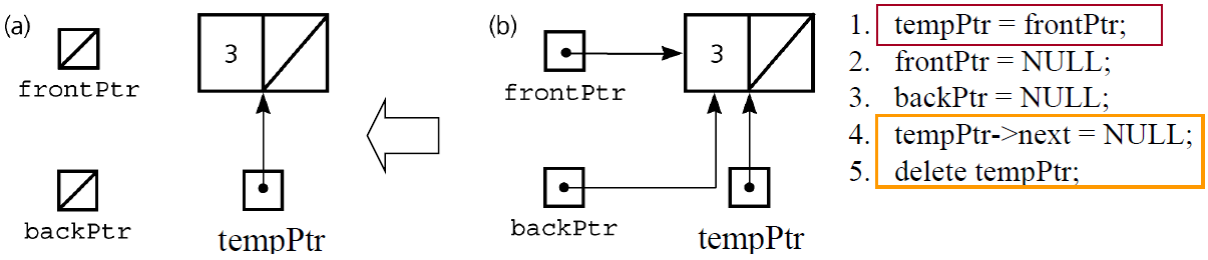


空的

這時候我要做的事情比較多

在這裡我的fonrtPtr是undefined的，所以我要把fonrtPtr跟backPtr設為NULL

```
tempPtr = frontPtr;
frontPtr = NULL;
backPtr = NULL;
tempPtr->next = NULL;
delete tempPtr;
```



```
void Queue::dequeue() {
  if (isEmpty())

  else {
    QueueNode* tempPtr = frontPtr;
    if (frontPtr == backPtr) {
      frontPtr = NULL;
      backPtr = NULL;
    } // if
    else {
      frontPtr = frontPtr->next;
    } // else
    tempPtr->next = NULL;
    delete tempPtr;
  } // else
} // dequeue()
```

實作circular Queue

每次enqueue我要做的是把新的節點加到backPtr指向的位置，而dequeue是要拿掉backPtr的next

首先我要先區分這個Queue是不是空的(backPtr沒有指向任何東西)

第一個節點是由backPtr的next來取得

```
void Queue::enqueue(const QueueItemType& newItem) {
  QueueNode* newPtr = new QueueNode;
  newPtr->item = newItem;
  if (isEmpty())
    newPtr->next = newPtr; // 把自己指向自己
  else {
    newPtr->next = backPtr->next; // 指到第一個位置
    backPtr->next = newPtr; // 把newPtr變成新的尾巴
  } // else
  backPtr = newPtr;
} // enqueue()
```

我要做dequeue也是要分成兩個情況

假設我有兩個節點，我就要把第一個節點抓出來清掉

或是要考慮我只有一個節點的情況下

```
void Queue::dequeue() {
   if (isEmpty())
    ...
  else {
    QueueNode* tempPtr = backPtr->next;
```

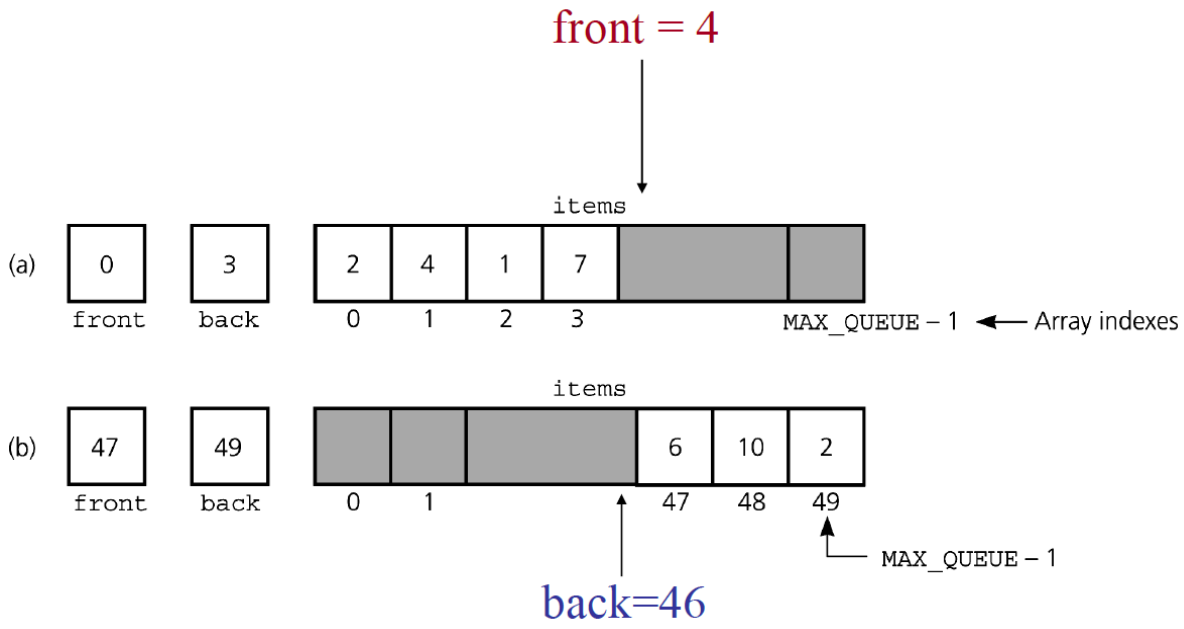
```
    if (backPtr==backPtr->next) //下一個還是自己
      backPtr = NULL; // 這時候才是一個節點
    else //因為tempPtr紀錄第一個節點
      backPtr->next = tempPtr->next;

    tempPtr->next = NULL;
    delete tempPtr;
  } // else
} // dequeue()
```

陣列實作Queue:

相較於linked list比較麻煩

會遇到的麻煩: 資料不斷地往右，後面都沒有把資料拿掉。在環狀的時候會滿足front = back + 1，代表是全滿的情況

全滿的時候基本上front跟back會差一個位置



當我要使用array來實作一個queue的時候就必須注意到這個問題

所以有一種解法是設置一個count來去計數

另外也有多宣告一個空間來解決這問題

或是設立一個旗標來檢查是不是滿的

單元7

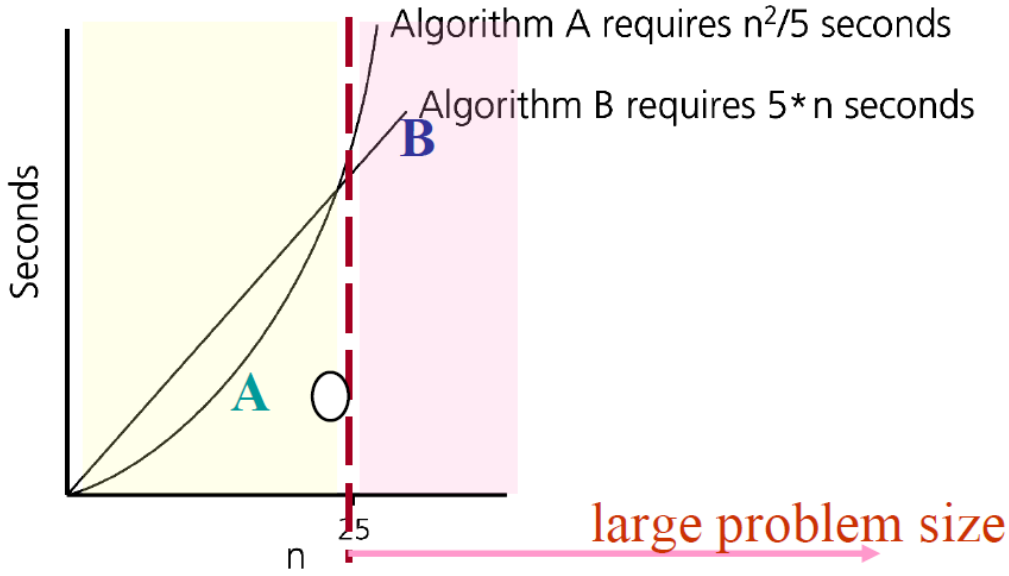
目標: 知道跑某個程式要花多少時間

n代表資料數目

y軸代表速度，所以越大越不好

在25筆資料的情況下上面的那條線(A)是比較差的，而n = 10，A的比較好

在我們比較兩個方法的時候，我們比較在意右邊(粉色)這一塊，因為右邊的問題需要比較多的時間來處理好，左邊通常忽略(資料少)



Algorithm A requires time proportional to n^2

Algorithm B requires time proportional to n

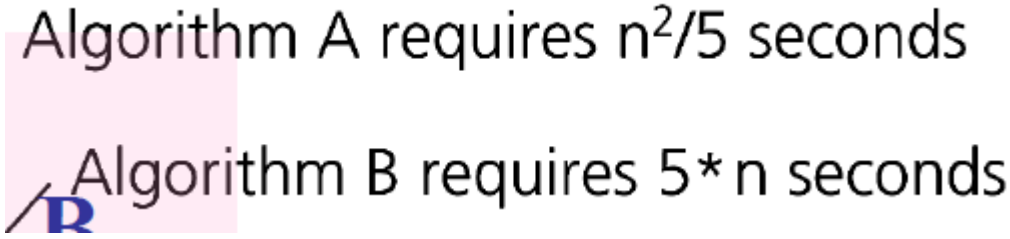
結論: B比較快，A比較慢

通常表示方式: A: O(n^2) b: O(n)

根據這個表示式可以區分出誰比較快、比較慢

n代表的是時間、運算次數。所以成長越快越不好

看到這邊有*5跟/5，但是在資料量很大的時候乘除5都變得不重要了。所以主要由n來決定誰比較大或是比較小



Bubble sort

第一種方式:

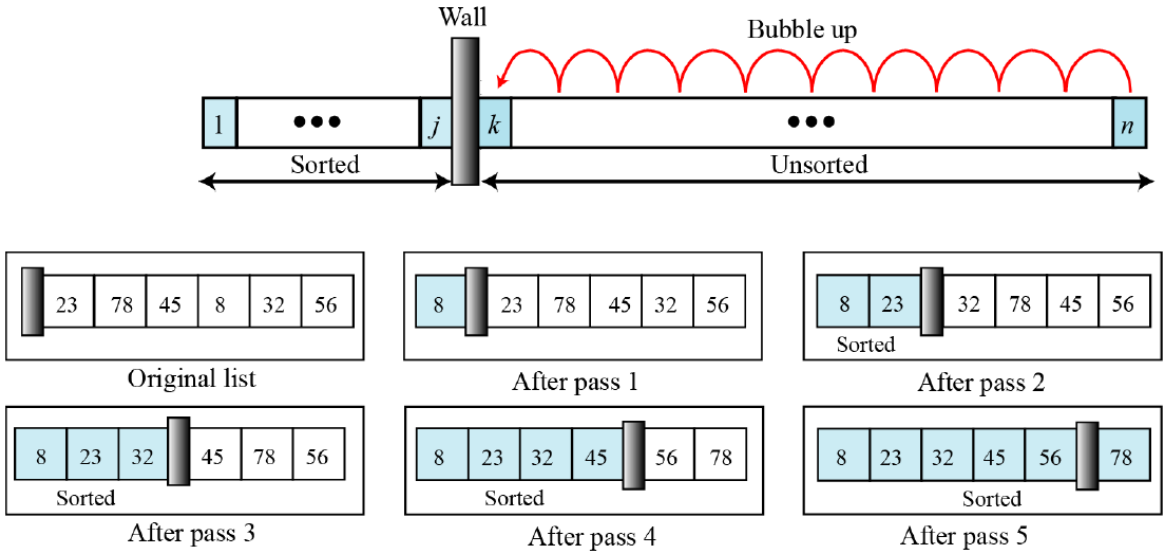
如果是由右邊往左邊做的話，就會有一個氣泡從右到左

如果我要的資料排序是遞增的，那我浮出來的氣泡就會是最小的

每一個圖就是一個回合的結束(每個回合做一樣的事情)

而當我浮出來一次，我的終點站就會提早

總共要做幾次: n - 1(n是資料數目)

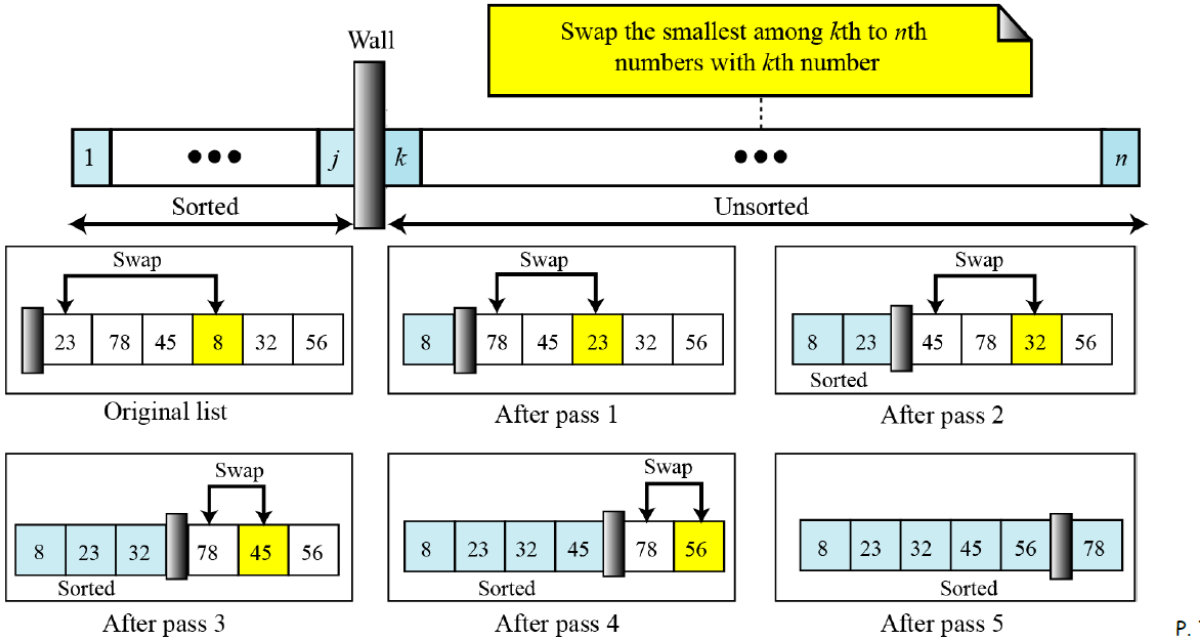


Selection sort

把一個位置，要浮出來的氣泡清出來。用這個位置來去跟其他位置做比較

有可能會交換或是不會交換

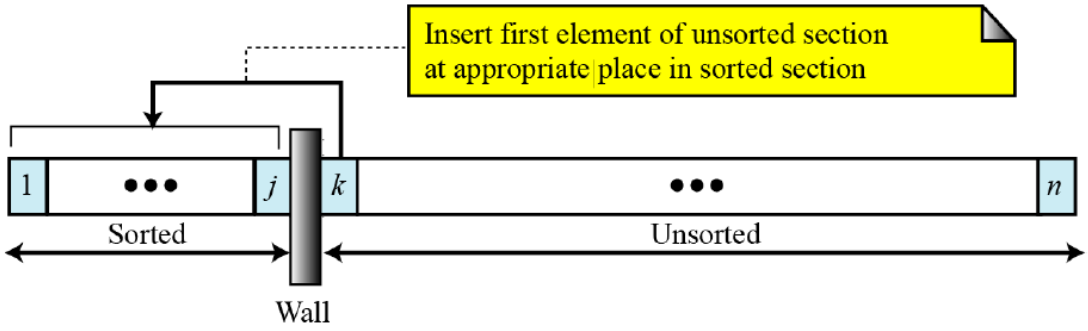
這個方式比bubble sort來得差一些



Insertion sort

例子:撲克牌

把一個"已經排好的東西"放到一個區域裡面，剩下的區域就是一個一個把它插入只要把右邊全部插好，就完成了

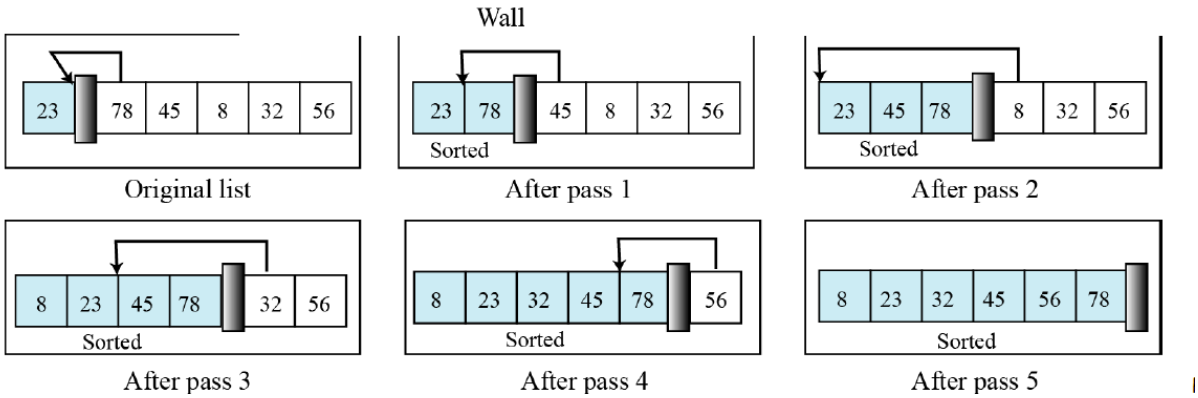


基本上第一個位置是我們預留的，當作它已經排好了，所以會從第二個位置開始做插入

所以以下面After pass 1為例，45是我要插入的數字，我要從右邊開始找，找到一個比45還要小的數字

由左邊做過來的話就是要找一個比45還要大的

一般來說是從右邊找到左邊

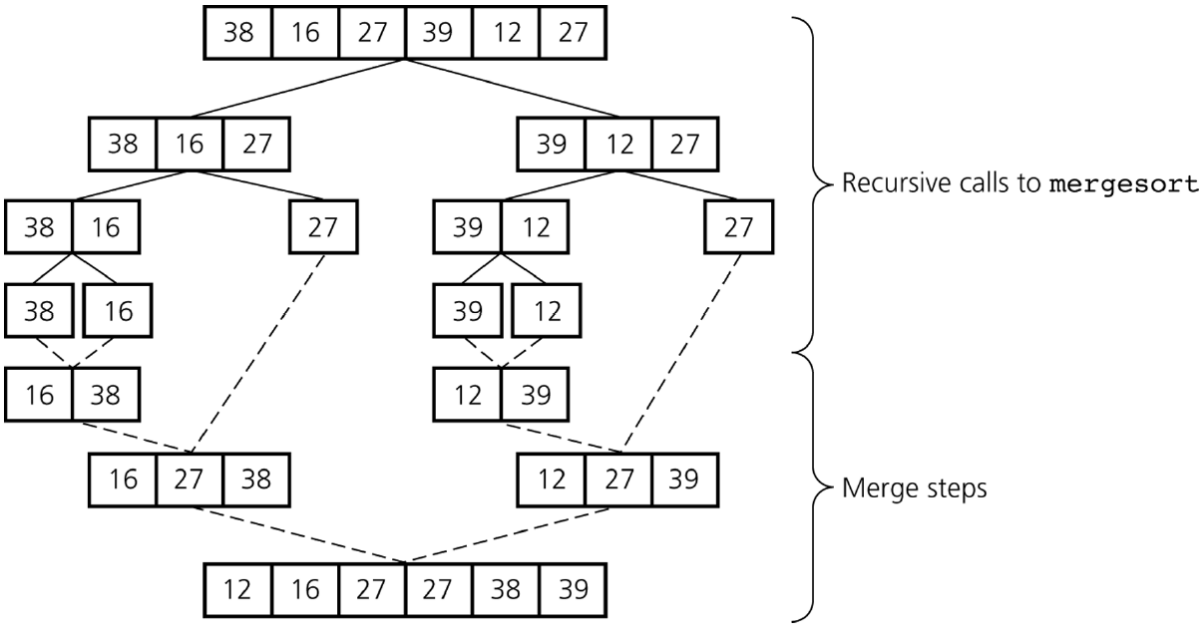


Merge sort

原理: 把資料做等分，不斷做等分，分到最後剩下一筆資料就不用排了。接下來要做的就是合併

虛線的部份就是合併，這時候才是真正花時間的地方。因為合併就要讀資料，資料越多就要讀越久

一開始只有一筆資料的時候只要比一次就知道了，到資料比較長的時候就要比複數次




```
void mergeSort(DataTypetheArray[], intfirst, intlast)
{
  if(first < last)
  {
    int mid = (first + last)/2;// middle point
    mergeSort(theArray, first, mid);// sort the left half
    mergeSort(theArray, mid+1, last);// sort the right half
    merge(theArray, first, mid, last);// merge the two halves
```

```
   }// end if
 }// end mergeSort
```

Quick sort

與merge sort蠻類似的，都是使用遞迴的方式來實現。從演算法的角度來看它們的架構是很像的

合併排序比較制式化，每次都是2等分，不管裡面的內容。

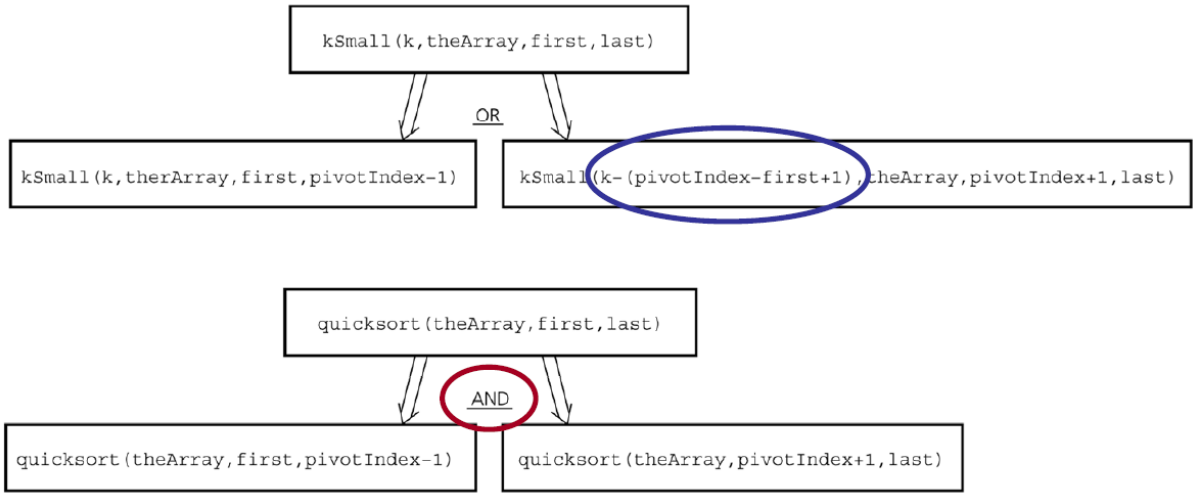
快速排序則是希望透過內容的比較來去判斷怎麼分，當它分成兩等分的時候會有可能是一邊短一邊長的情況

主要觀念: 希望透過某一個數值(從資料裡面挑出來的)去判斷它應該怎麼去分，挑的方式可以有很多種。e.g.: 第一個或是最後一個

挑出來後希望把所有的數字跟他比較。目的是要找到pivot(選擇的資料)在哪個地方出現

希望pivot的後面都是比較大，前面都是比較小。正確的時候就不要去動它。所以會因為我選擇的pivot而產生出小於或是大於的數量是不固定的，所以不會像merge sort一樣會固定等分。

效率比較不穩定



Radix sort

Radix Sort與前面的方法差在它不用比較

只看自己，每筆資料在排序的過程中只需要看自己就好

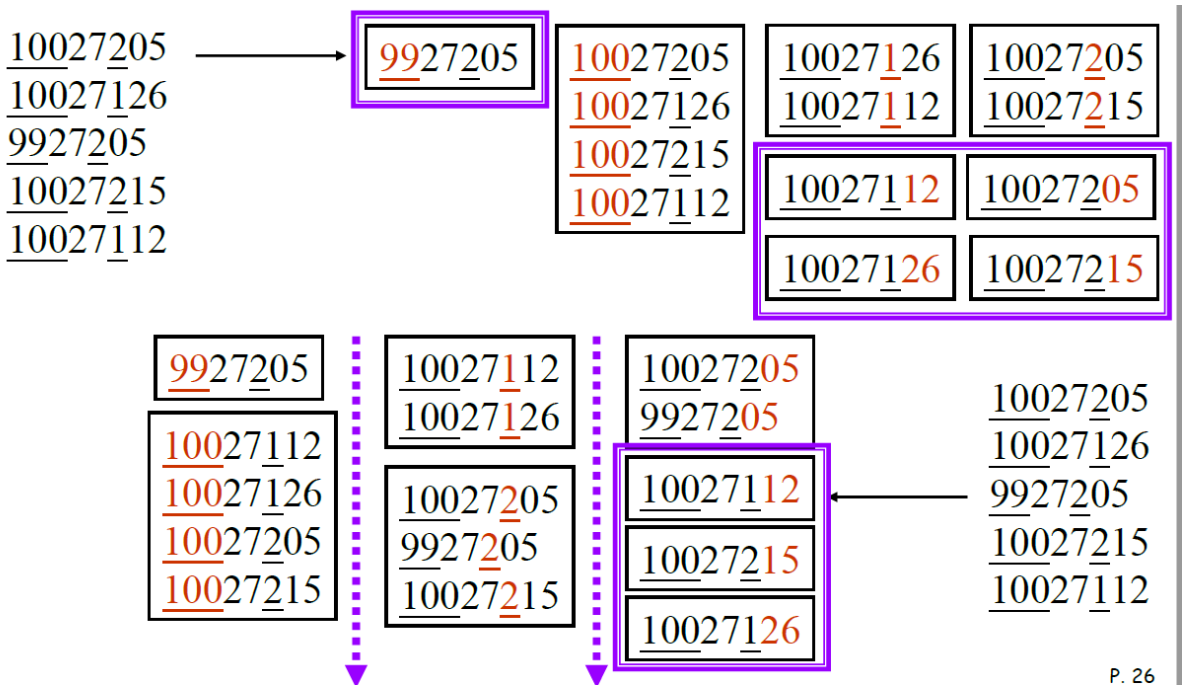
前提是這些資料是可以分解的

分解的意思是資料可以分為一部分一部分的

它花的時間並不是比較，而是資料的搬動

它會有些容器，而這些容器是用來放資料用的，資料會在這些容器移動，決定去哪個容器就只需要看自己就好了

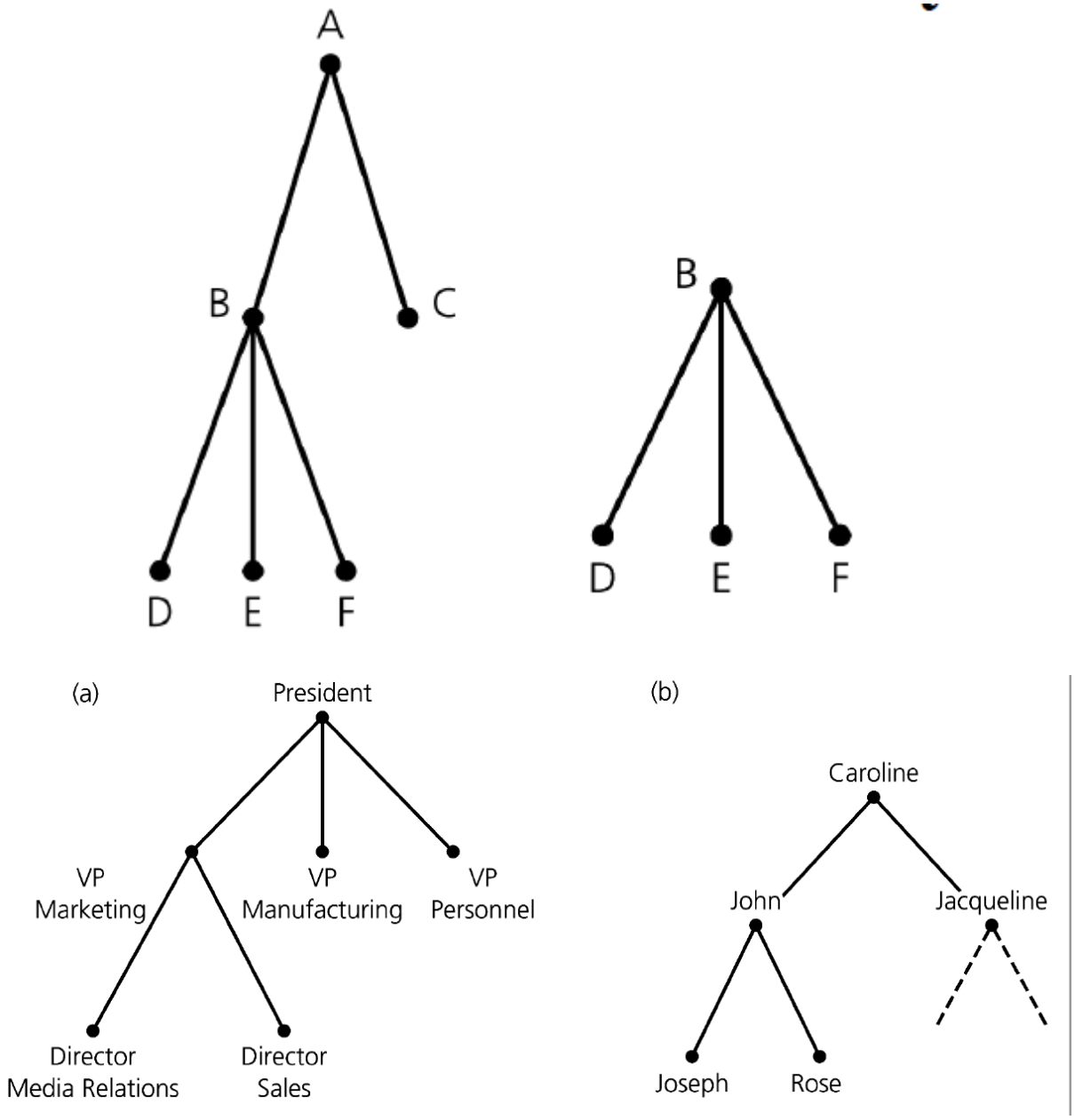
從左邊看過來的時候，我們會知道99比較小，其他100比較大。所以99一定會排在100的前面，所以這個容器的東西就不用跟後面的做排序了。後面有4個，所以要做排序，以相同方式去做拆解



單元8

二元的簡單定義: 底下接的節點只能是兩個

下面的就不是二元樹



樹是有順序的，我們會先選一個節點當作root，然後樹是有階級的分類。樹根就是第一級，下面就是2、3...階級

所以所謂的二元樹就是任何一個點往下看只有兩個child的，就是二元樹

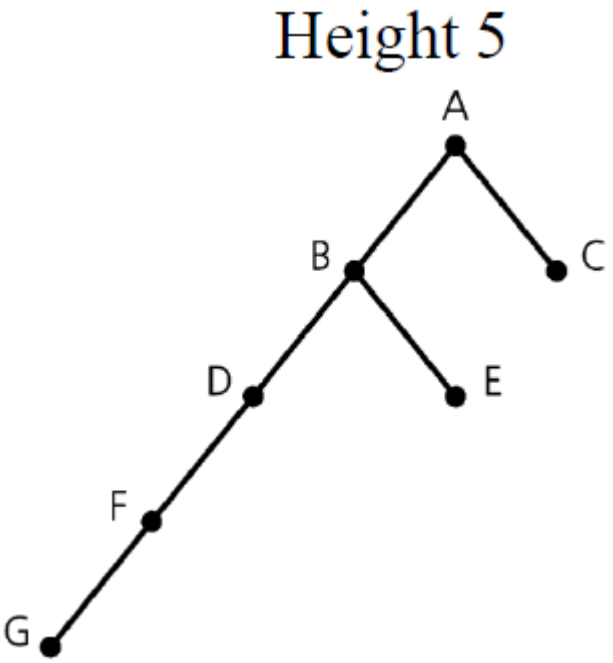
樹高

定義: 從A找到任何一個點，最多需要5個點

樹高可以衡量這個結構效率的好壞

從A→C與A→G是不一樣的。這要代表這棵樹找資料是困難的

如果以邊來看，這個說樹高是4也並不是錯誤的



樹高越低是越好的，因為找資料比較快

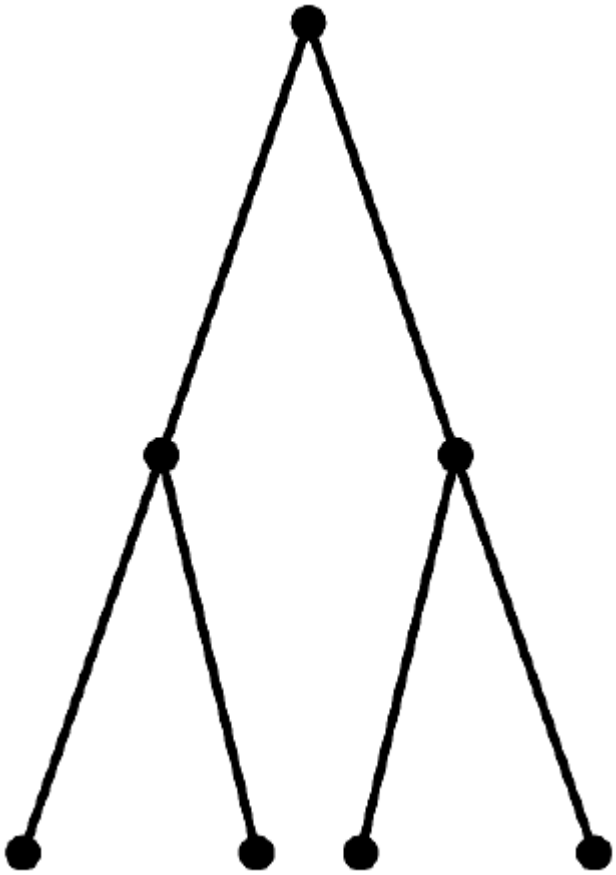
要怎麼擺才能找資料比較快?

第一種: Full binary tree

盡量把每一層都塞滿的形式

每一層能容納的數量是固定2^n

若我的樹高是h，在這裡能儲存資料的數量是2^h - 1，沒辦法去處理8、9筆的資料



所以我們有另外一種二元樹

第二種: Complete Binary Trees

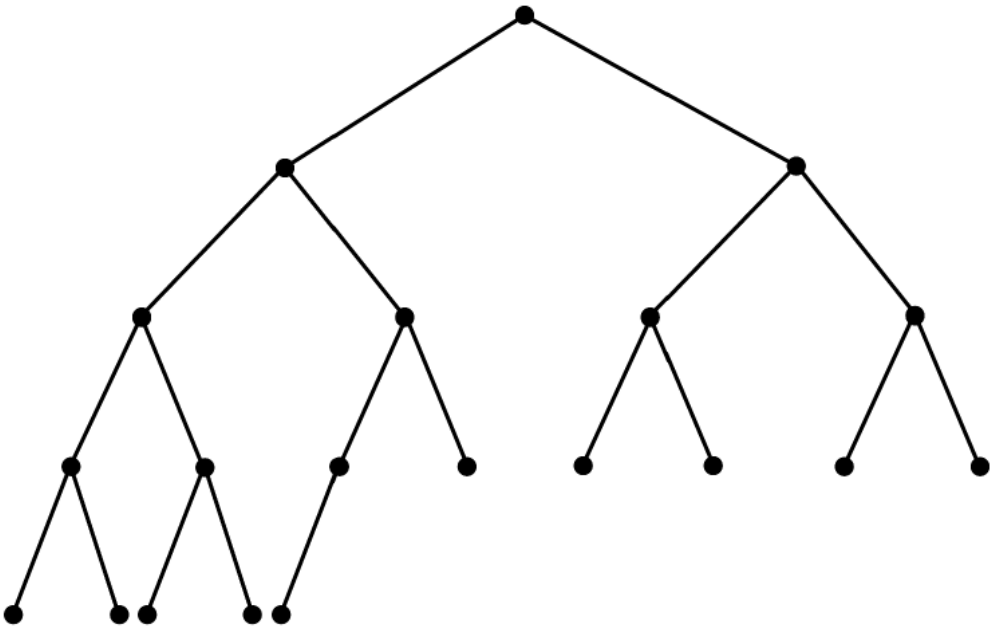
它可以不要是full tree，但是具備full tree的特性

若我今天是15筆資料，我一樣是盡可能地擺滿，而這擺放的方式是固定的

擺放方式:由上而下，由左而右

這樣擺出來的樹就是完整樹

在這樣的資料數量下，這樣的擺放方式是樹高最短的



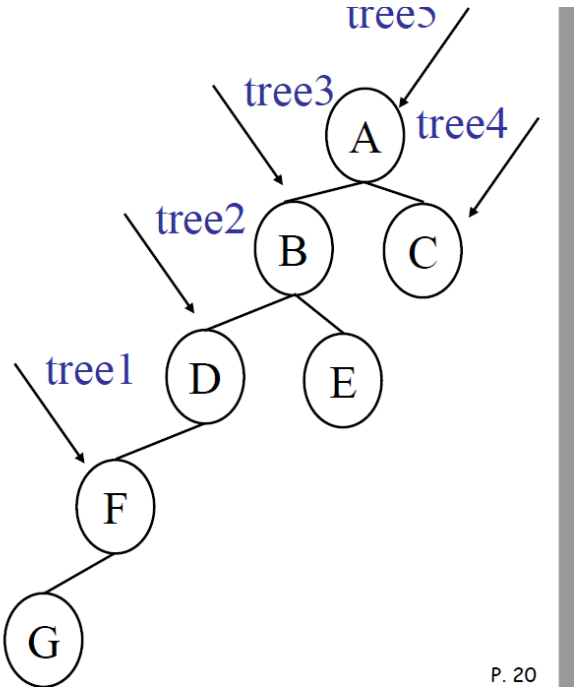
二元樹的ADT

二元樹是可以指定位置幫你存放資料，二元搜尋樹是不能指定位置幫你存放資料(有排序的效果)

樹的一些運算

Building the ADT binary tree

```
tree1.setRootData('F')
tree1.attachLeft('G')
tree2.setRootData('D')
tree2.attachLeftSubtree(tree1)
tree3.setRootData('B')
tree3.attachLeftSubtree(_____)
tree3.attachRight('E')
tree4.setRootData('C')
tree5.createBinaryTree('A',____,
   tree4)
```



P. 20

今天我要實作一個二元樹也是有兩種方式: 指標跟array

陣列實作:

今天我是陣列，所以我只要記得陣列的位置就好。陣列的第幾個元素是leftChild?

所以我用陣列來存放所有節點

```
class TreeNode {
  private:
    TreeItemType item;
    int leftChild;
    int right child;
}; // end TreeNode

TreeBide tree[100];
int root;
int free;
```

指標實作:

在這情況我就不會被震裂限制空間

```
class TreeNode {
  private:
    TreeItemType item;
    TreeNode* leftChild;
    TreeNode* rightchild;
}; // end TreeNode

TreeNode* root;
```

二元樹走訪

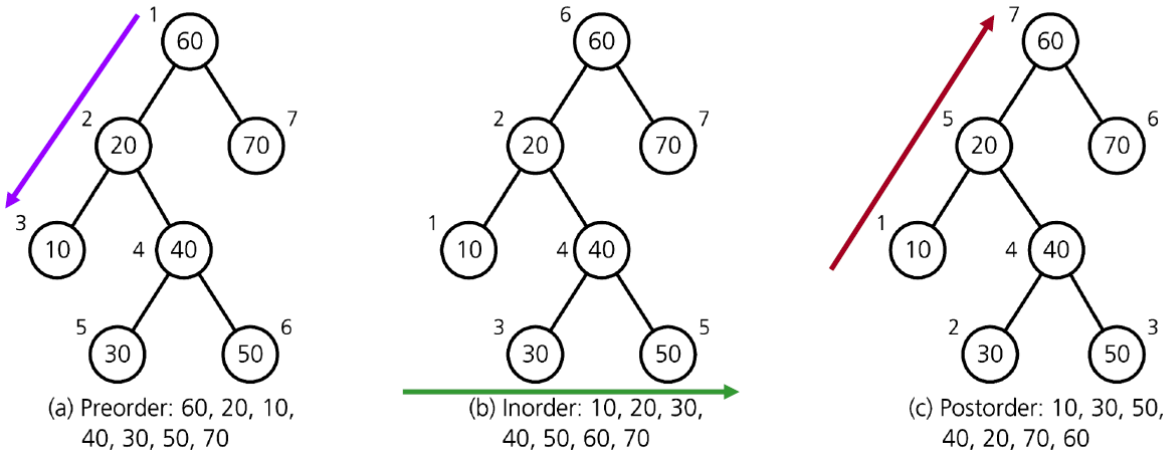
樹這結構很適合使用遞迴來解釋的，因為每個節點扮演的角色是很像的

當我走到任何一個點都會有三個部分，自己、左、右

自己甚麼時候做可以自由決定

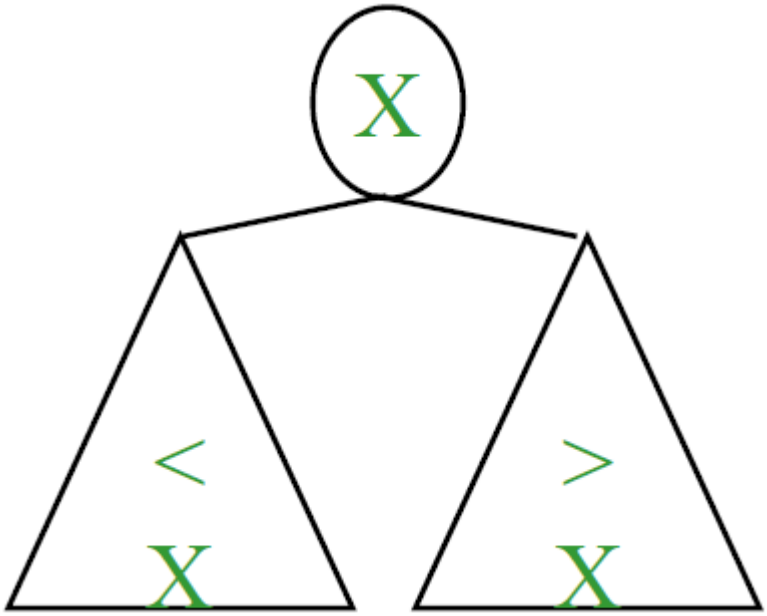
擺的位置不一樣，會造成順序不一樣，所以要注意。

```
traverse(in binTree: BinaryTree) {
  if (binTree is not empty) {
    // ....
    traverse(Left subtree of binTree's root);
    // ....
    traverse(Right subtree of binTree's root);
    // ....
  } // if
} // traverse()
```



二元搜尋樹:

基本定義:左邊比較小、右邊比較大，當等於的時候可以放在左邊或是右邊



效率與樹高有關:

一般情況: O(logn)，worst case:O(n)