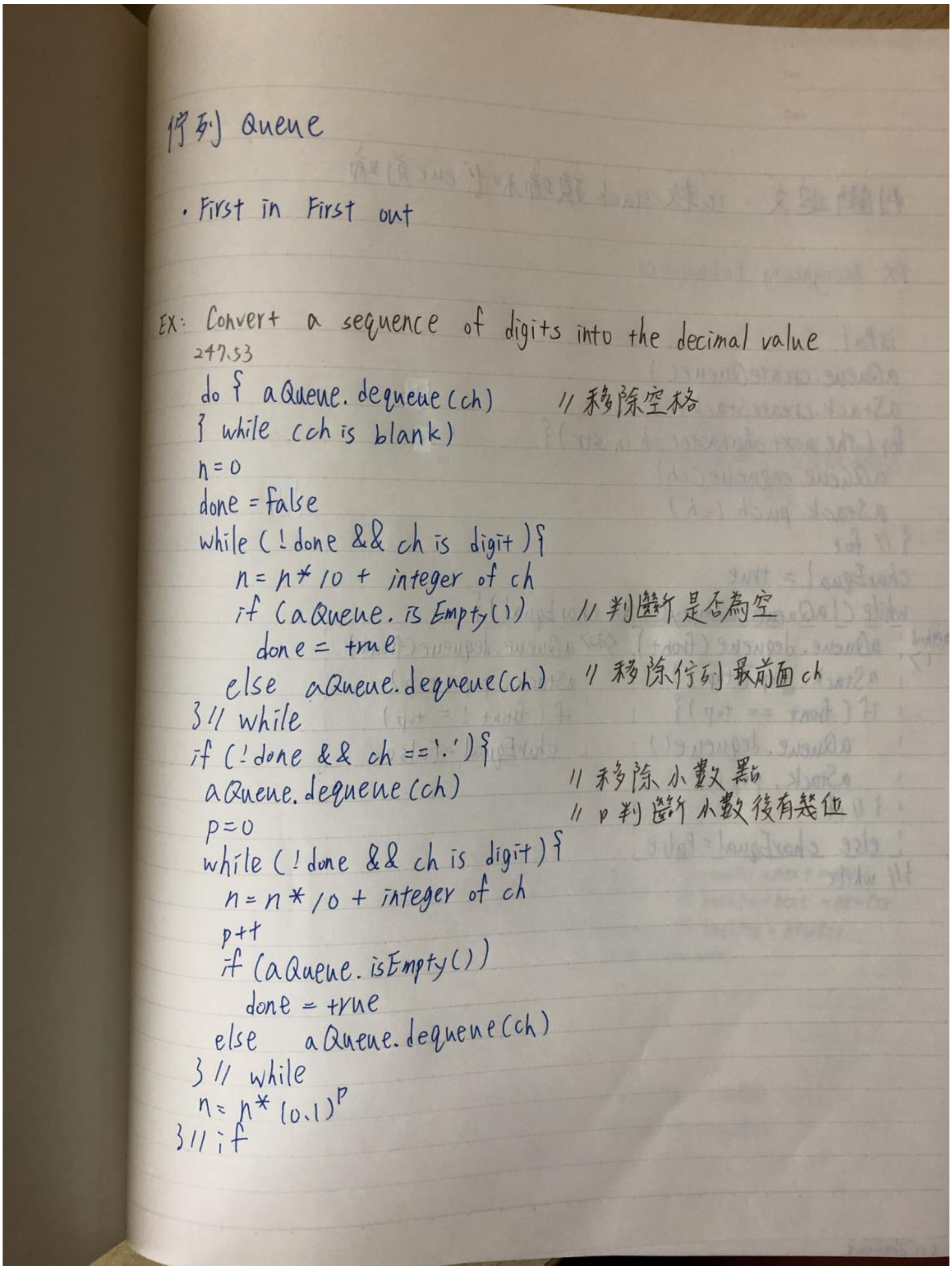


佇列 Queue

- First in First out

EX: Convert a sequence of digits into the decimal value

247.53

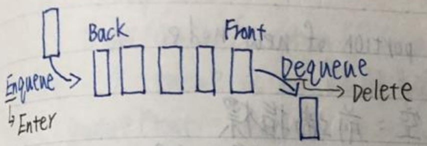
```
do { aQueue.dequeue(ch)          // 移除空格
} while (ch is blank)
n=0
done = false
while (!done && ch is digit){
   n = n*10 + integer of ch
   if (aQueue.isEmpty())          // 判斷是否為空
     done = true
   else  aQueue.dequeue(ch)       // 移除佇列最前面ch
}// while
if (!done && ch =='.'){
 aQueue.dequeue(ch)               // 移除小數點
 p=0                              // p 判斷小數後有幾位
 while (!done && ch is digit){
   n = n*10 + integer of ch
   p++
   if (aQueue.isEmpty())
     done = true
   else  aQueue.dequeue(ch)
 }// while
 n= n*(0.1)^p
}//if
```

判斷迴文：比較 stack 頂端和 queue 前端

EX: Recognizing Palindromes.

```
isPal
aQueue.createQueue()
aStack.createStack()
for(the next character ch in str){
  aQueue.enqueue(ch)
  aStack.puch(ch)
}// for
charEqual = true
while(!aQueue.isEmpty() && charEqual){
<法1>
  aQueue.getFront(front)
  aStack.getTop(top)
  if(front == top){
     aQueue.dequeue()
     aStack.pop()
  }//if
  else charEqual=False
<法2>
  aQueue.dequeue(front)
  aStack.pop(top)
  if(front != top)
    charEqual=False
}// while
```

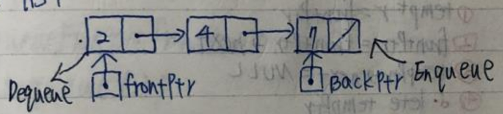
Implementations of the ADT Queue



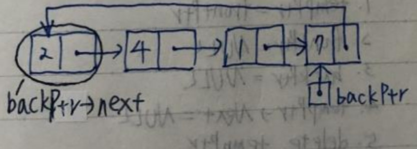
- A circular linked list with one external reference
  - Only a reference to the back (環狀:只有後端)

優:減一個變數

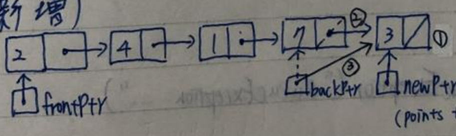
linked list



Circular linked list

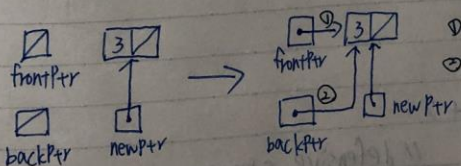


Enqueue (新增)



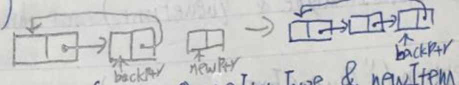
① newPtr→next = NULL
② backPtr→hext = newPtr
③ backPtr = newPtr

若一開始為空



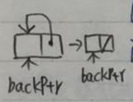
① frontPtr = newPtr
② backPtr = newPtr

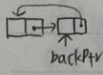
Circular(環狀)

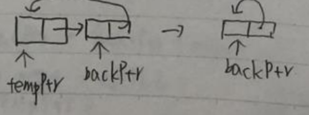


```
void Queue :: enqueue ( const QueueItemType & newItem ) { // 新增
    QueueNode * newPtr = new QueueNode ;
    newPtr → item = newItem ;
    if ( isEmpty() )                              // 0 → 1 node
      newPtr → next = newPtr ;                    // point to itself
  else {                                          // k → k+1 nodes, k>0
      newPtr → next = backPtr → next ;            // point to the front
      backPtr → next = newPtr ;                   // put behind the back
   }// else
   backPtr = newPtr ;        // new node at the back
} // enqueue
```

```
void Queue :: dequeue() throw ( QueueException ) { // 移除
   if ( isEmpty() )
     throw ... ;
   else {
     QueueNode * tempPtr = backPtr → next ;       // tempPtr 指向 front
     if ( backPtr == backPtr → next )             // 只有一個節點
       backPtr = NULL ;                           // one node → empty
    else                                          // the next front
       backPtr → next = tempPtr → next ;
    tempPtr → next = NULL ;                       // defensive strategy
    delete tempPtr ;                              // release space
  }// else
} // dequeue
```

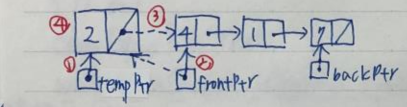




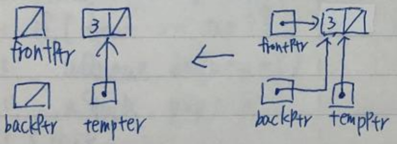


```
void Queue :: enqueue ( const QueueItemType & newItem ){ // 擷取後移除
    QueueNode * newPtr = new QueueNode;
    newPtr → item = newItem;      // set data portion of new node
    newPtr → next = NULL;              新節點及指標
    if (isEmpty()) frontPtr = newPtr;    全空：前端指標
   else backPtr → next = newPtr;         非空：後端指標下一個
   backPtr = newPtr;             // new node is at the back
}// enqueue
```

Dequeue (移除)



①tempPtr = frontPtr
②frontPtr = frontPtr → next
③ tempPtr → next = NULL
④ delete tempPtr



1. tempPtr = frontPtr
2. frontPtr = NULL
3. backPtr = NULL
4. tempPtr → Next = NULL
5. delete tempPtr

```
void Queue :: dequeue( ){
  if (isEmpty() )  throw QueueException ("QueueException: ...")
  else {
    QueueNode * tempPtr = frontPtr;
    if ( frontPtr == backPtr){   // one node only
       frontPtr = NULL;
       backPtr = NULL;
    }//if
    else  frontPtr = frontPtr → next;
    tempPtr → next = NULL;          // defensive strategy
    delete tempPtr;                 //釋放空間
  }// else
}// dequeue
```

陣列實作佇列ADT (使用環狀陣列)

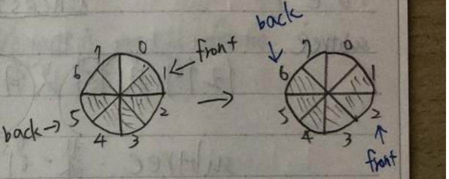
To initialize the queue:

```
    front = 0
    back = Max_Queue-1
    count = 0
```

Insert:

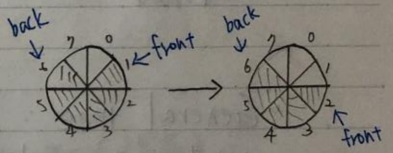
```
    back = (back+1) % Max_QUEUE;
    items[back] = newItem;
    ++count;
```

back初始值 = Max_Queue-1



Deleting:

```
    front = (front+1) % Max_QUEUE;
    --count;
```

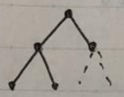


全空條件: count = 0

全滿條件: count = Max_QUEUE

Binary Tree

最多 2個 node



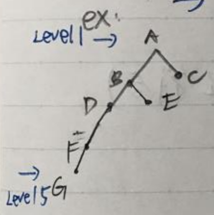
- left subtree
- right subtree

* Height of a tree (樹高)

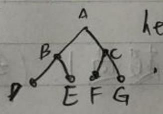
→ number of nodes along the longest path from root to a leaf

→ 影響"效率"(越小越好)

ex.



height 5



height 3

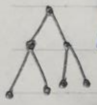
* Level of a node n in a tree T (階層)

* 最大階層 = 樹高

$height(T) = 1 + \max\{height(T_L), height(T_R)\}$

* 樹裡無 cycle

Full Binary Tree (完全樹)



height h is full

- nodes at levels < h have two children each

complete Binary Tree (完整樹)

① full to level-1   ex: level=5, level 4 是 full binary tree

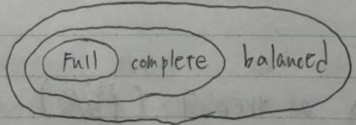
② Level h is filled from left to right

③ at levels <= h-2 have two children each

balanced

→ 左樹高, 右樹高差距不超過 1. (每個點都要符合)

→ 效率穩定, 省空間



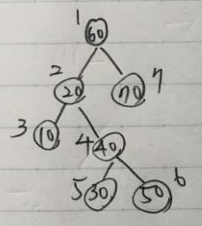
Traversal

- preorder ⇒ visit root before visiting its subtrees.

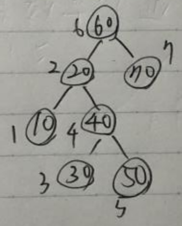
- Inorder ⇒ visit root between visiting its subtrees.

- Postorder ⇒ visit root after visiting its subtrees

Preorder

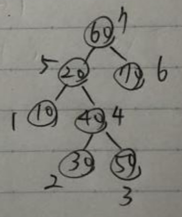


Inorder



左邊先

Postorder



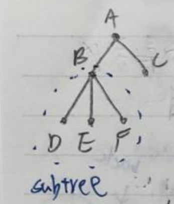
Inorder = sort

Trees

間

目的:紀錄資料關係(二維)

內容導向



Trees are hierarchical

親子

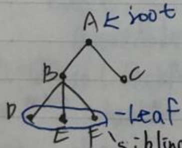
- Parent - child relationship between two nodes
- Ancestor - descendant relationship among nodes

祖孫

任何資料只有一個直屬長官適用於 trees

subtree:某一部份資料形成小樹
子樹

· General tree: one or more nodes
root ... a single node



Parent of node B: A

-Leaf 葉節點:A node with no children

siblings 兄弟節點: Node with a common parent
(父節點為同一人)

Ancestor of node B 祖先節點 A D

Deςcendant of node B 子孫節點 D A