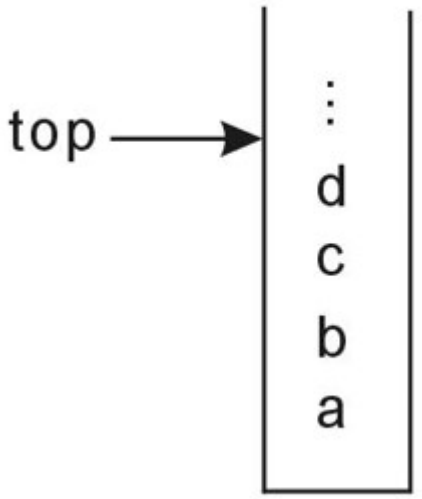
Chapter 3 堆疊與佇列

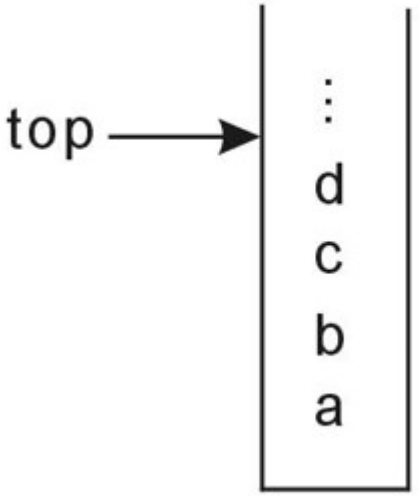
堆疊基本觀念

- 堆疊是一有序串列（order list），其加入（insert）和刪除（delete）動作都在同一端，此端通常稱之為頂端（top）。
- 加入一元素於堆疊，此動作稱為推入（push），與之相反的是從堆疊中刪除一元素；此動作稱為彈出（pop）。
- 由於堆疊具有先進去的元素最後才會被搬出來的特性，所以又稱堆疊是一種後進先出（Last In First Out，LIFO）串列。



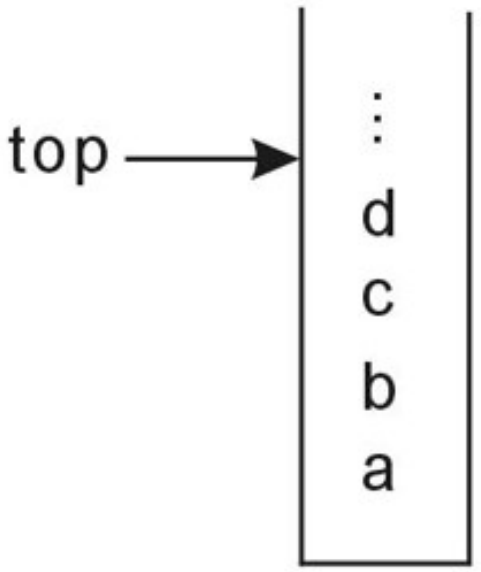
Stack Abstract Data Type

```
public class Stack {
        public boolean isEmpty()
        public boolean isFull()
        public int Top()
        //return top element of stack
        public void push(int value)
        //insert item into the top of the stack
        public int pop()
        //Delete the top element of the stack
}
```



Stack Implementation (1)

```
public class MyIntStack {
    private int[] myStack;
    private int top;
    private int capacity;
    MyIntStack( int stackCapacity) throws Exception{
        if(stackCapacity<1) throw new Exception("StackCapacity需大於0");
        this.capacity = stackCapacity;
        myStack = new int[stackCapacity];
        top = -1;
    }
    public boolean isEmpty(){
        if(top<0)
                return true;
        else
                return false;
    }
    public boolean isFull(){
        if(top==capacity-1)
                return true;
        else
                return false;
    }
```



Stack Implementation(2)

```
    public int top() throws Exception{
        if(isEmpty()) throw new Exception("目前Stack是空的!");
        return myStack[top];
    }

    public void push(int value) throws Exception{
        if(isFull()) throw new Exception("目前Stack已滿，不可push物件!");
        top= top+1;
        myStack[top] = value;
    }

    public int pop() throws Exception{
        if(isEmpty()) throw new Exception("目前Stack是空的，沒有物件可
        pop!");
        int curTop = top;
        top = top -1;
        return myStack[curTop];
    }
}
```

Stack 泛型版

```
public class MyGeneralStack<T> {
    private T[] myStack;
    private int top;
    private int capacity;
    MyGeneralStack( int stackCapacity) throws
    Exception{
        if(stackCapacity<1) throw new
        Exception("StackCapacity需大於0");
        this.capacity = stackCapacity;
        myStack = (T[])new Object[stackCapacity];
        top = -1;
    }
    public boolean isEmpty(){
        if(top<0)
                return true;
        else
                return false;
    }
    public boolean isFull(){
        if(top==capacity-1)
                return true;
        else
                return false;
    }
```

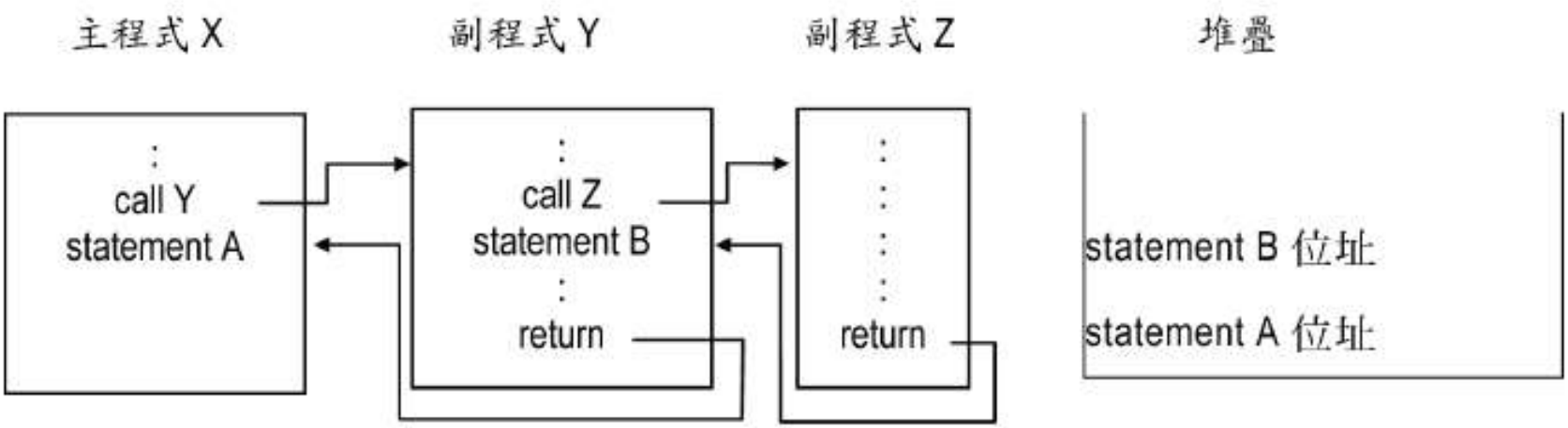
```
    public T top() {
        if(isEmpty()) {
                System.out.println("目前
        Stack是空的!");
        }
        return myStack[top];
    }

    public void push(T t){
        if(isFull()) {
                System.out.println("目前Stack
        是滿的! 不可push物件");
        }
        else
        {
                top= top+1;
                myStack[top] = t;
        }
    }
    public T pop(){
        if(isEmpty()){
                System.out.println("目前Stack是
                空的，沒有物件可pop!，回傳null");
                return null;
        }
        else
        {
                int curTop = top;
                top = top -1;
                return myStack[curTop];
        }
      }
}
```

作業02-01

- 請實作一個泛型版的Stack

堆疊的應用 - function呼叫



堆疊的應用 -中序表示式(infix expression)轉為後序表示式(postfix expression)

- 一般的算術運算式皆是中序表示式，亦即運算子(operator)置於運算元(operand)的中間(假若只有一個運算元，則運算子置於運算元的前面)。

  如：A * B / C

- 後序表示式則是將運算子置於其對應運算元後面。

  如A * B / C運算式的後序表示式為AB * C /。

- 編譯器不易處理中序表示式的計算，但易於處理後序表示式的計算
  - 如何將中序轉為後序?
  - 如何計算後序表示式?

如何將中序轉為後序 (1)

- 可依下列三步驟進行即可：
  1. 將中序表示式加入適當的括號，此時須考慮運算子的運算優先順序。
  2. 將所有的運算子移到它所對應右括號的右邊。
  3. 將所有的括號去掉。

- 如將A * B / C 化為後序表示式：
  1. ((A * B) / C)
  2. ((A * B) / C) => ((AB) * C) /
  3. AB * C /

練習一

- 請將以下的中序表示式轉為後序表示式
  1. a*(b*-c+d)/(e*f)
  2. a*b/(d/e)+c*f*g

-c中的-是負號

- 請將以下的後序表示式轉為中序表示式
  3. abcd*+*e/f-

如何將中序轉為後序 (1)

- 觀察 A/B-C+D*E-A*C，其後序表示式為 AB/C-DE*+AC*-
  - 輸出運算元的順序都一樣，都為ABCDEAC。因此在轉換過程中，若遇到運算元就將其輸出
  - 若遇到運算子就將其放入stack，在適當時機再將其取出
    - 只要堆疊最上層的in-stack priority <= in-coming priority，就將最上層的運算子取出
    - 若遇到的是')'， 將stack中的運算子都輸出直到遇到')'

Operator	In-stack priority	In-coming priority
Unary minus, !	1	1
*, /, %	2	2
+, -	3	3
<, <=, >=, >	4	4
==, !=	5	5
&&	6	6
||	7	7
(	8	0
#	8	

如何將中序轉為後序 (2)

A+B*C

Next token	Stack	output
None	Empty	None
A	Empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
	empty	ABC*+

A*(B+C)*D

Next token	Stack	output
None	Empty	None
A	Empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
	empty	ABC+*D*

如何將中序轉為後序 (3)

```
void Postfix (Expression e)
{
   //the last token in e is '#.' Also, '#' is used at the bottom of the stack
   Stack<Token> stack; // initialize stack
   stack.Push('#');
   for(Token x=NextToken(e);x!='#';x=NextToken(e))
      if(x is an operand)
         輸出x;
      else if(x == ')') {// unstack until '('
      {
         for(;stack.Top()!='(';stack.Pop())
          輸出 stack.Top();
         stack.Pop(); (); // unstack '('
      }
      else{// x is an operator
         for(;isp(stack.Top())<=icp(x); stack.Pop())
          輸出 stack.Top();
         stack.Push(x );
      }
   // end of expression; empty the stack
   for (;!stack.IsEmpty(); stack.Pop())
      輸出 stack.Top();
}
```

作業02-02

請完成一個function，function傳入一個中序運算式，回傳此中序運算式的後序運算式 (傳入的中序運算式中，運算元以英文字母表示，如A*(B+C)*D)

如何計算後序表示式(1)

1. 將後序表示式以一字串表示之。
2. 每次取一個token，
   - 若此token 為一運算元，則將它push 到堆疊。
   - 若此token 為一運算子，則自堆疊pop 出適當數量運算元(如-pop 1個、+-*/ pop 2個)，並計算。若此token 為 '#' (假設最後一個字元是'#.')，則跳到步驟4。
3. 將步驟2 的結果，push 到堆疊，之後再回到步驟2。
4. 彈出堆疊的資料，此資料即為此後序表示式計算的結果。

AB/C-DE*+AC*-

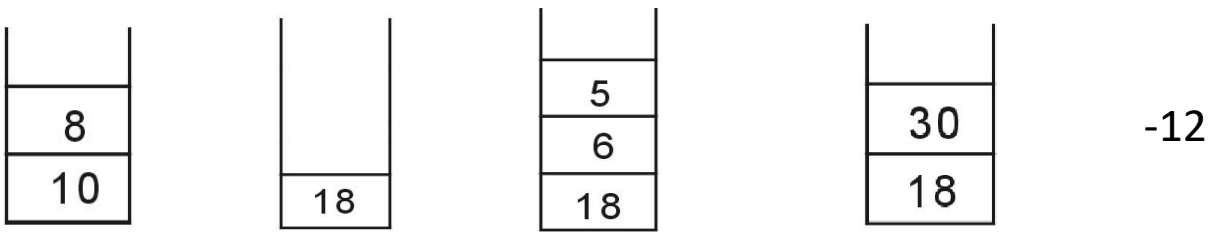
operation	postfix
T_1 = A/B	T_1C–DE*+AC*-
T_2 = T_1-C	T_2DE*+AC*-
T_3 = D*E	T_2T_3+AC*-
T_4 = T_2+T_3	T_4AC*-
T_5 = A*C	T_4T_5-
T_6 = T_4-T_5	T_6

如何計算後序表示式(2) – pseudo code

```
void Eval(Expression e)
{
        Stack<Token> stack; //initialize stack
        for (Token x = NextToken(e) ; x != '#'; x = NextToken(e))
                if (x is an operand)
                        stack.Push(x) // add to stack
                else {// operator
                        remove the correct number of operands for operator x from stack;
                        perform the operation x and store the result (if any) onto the stack;
        }
}
```

後序表示式10 8 + 6 5 * –

1. 因為10為一運算元，故將它push到堆疊，同理8也是，故堆疊有2個資料分別為10和8。
2. 之後的token為+，故pop出堆疊的8和10做加法運算，結果為18，再次將18 push到堆疊。
3. 接下來將6和5 push到堆疊。
4. 之後的token為*，故pop 5和6做乘法運算為30，並將它push到堆疊。
5. 之後的token為-，故pop 30和18，此時要注意的是18減去30，答案為-12（是下面的資料減去上面的資料）。



作業02-03

請完成一個function，function傳入一個後序運算式，回傳計算的結果

如何將中序轉為前序(prefix expression)

- 可依下列三步驟進行即可：
  1. 將中序表示式加入適當的括號，此時須考慮運算子的運算優先順序。
  2. 將所有的運算子移到它所對應左括號的左邊。
  3. 將所有的括號去掉。

- 如將a*b-c/(d+e)化為前序表示式：
  1. ((a*b)-(c/(d+e)))
  2. -(*(ab) / (c+ (de)))
  3. -*ab/c+de

思考：
如何將中序轉為前序?
如何計算前序表示式?

練習

- 請將以下的中序表示式轉為後序表示式
  1. a*(b*-c+d)/(e*f)
  2. a*b/(d/e)+c*f*g
- 請將以下的後序表示式轉為中序表示式
  3. +*/ab-+cde-fg

-c中的-是負號

進階作業02-01

1. 請完成一個function，function傳入一個中序運算式，回傳此中序運算式的前序運算式(傳入的中序運算式中，運算元以數字表示，如3*(6+2)*5)
2. 請完成一個function，function傳入一個前序運算式，回傳計算的結果

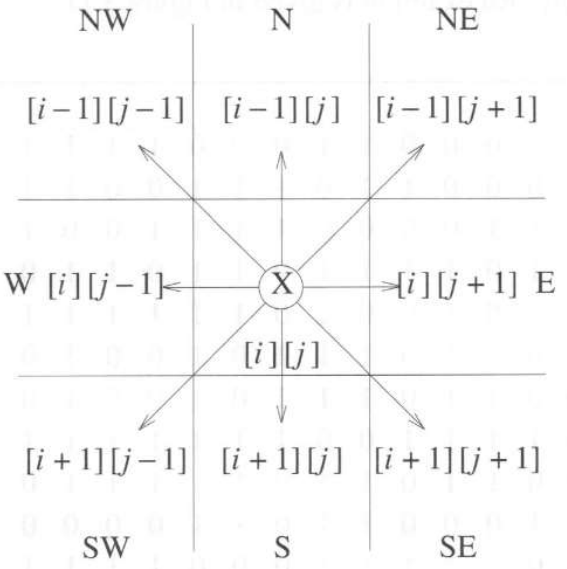
迷宮找路徑問題(1)

- 迷宮以二維陣列表示之maze[i][j]，1 ≤ i ≤ m, 1 ≤ j ≤ p
- 1表示不能走的區域，0表示可以走的區域
- 老鼠從maze[1,1]開始走，從maze[m][p]出來

entrance

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

exit

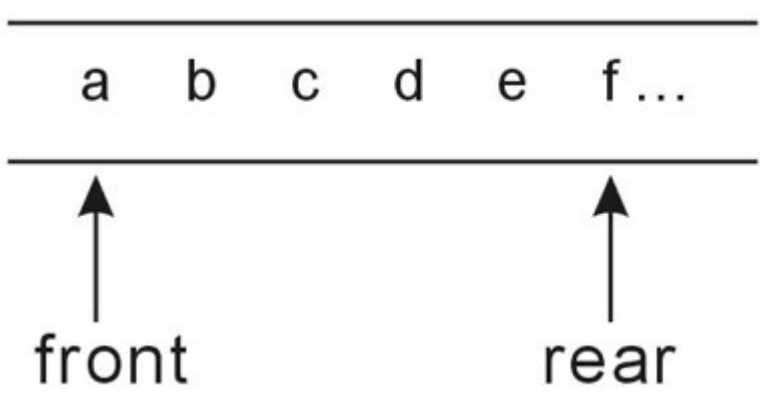


進階作業02-02

- 請完成迷宮找路徑問題

佇列基本觀念

- 佇列也是屬於線性串列，與堆疊不同的是加入和刪除不在同一端，刪除的那一端為前端（front），而加入的一端稱為後端（rear）。
- 佇列的運作，分別利用rear 變數作用在加入的動作；front 變數作用在刪除的動作，佇列的加入要注意它是否超出最大的容量，front and rear 變數的初值為–1。




```
public class Queue {
        public boolean isEmpty();
        public boolean isFull();
        public void push(int value);
        public int pop();
}
```

佇列實作

```
public class Ch03MyIntQueue {
    private int[] myQueue;
    private int front;
    private int rear;
    private int capacity;
    Ch03MyIntQueue( int queueCapacity)
    throws Exception{
        if(queueCapacity<1) throw new
        Exception("QueueCapacity需大於0");
        this.capacity = queueCapacity;
        myQueue = new int[queueCapacity];
        front = -1;
        rear = -1;
    }

    public boolean isEmpty(){
        if(this.front == this.rear)
                return true;
        else
                return false;
    }

    public boolean isFull(){
        if(rear==capacity-1)
                return true;
        else
                return false;
    }
```

```
public void push(int value) throws
Exception{
    if(isFull()) throw new
    Exception("目前Queue已滿，不可push
    物件!");
    rear= (rear+1);
    myQueue[rear] = value;
}

public int pop() throws Exception{
    if(isEmpty()) throw new
    Exception("目前Queue是空的，沒有物
    件可pop!");
    front = front +1;
    return myQueue[front];
}
```

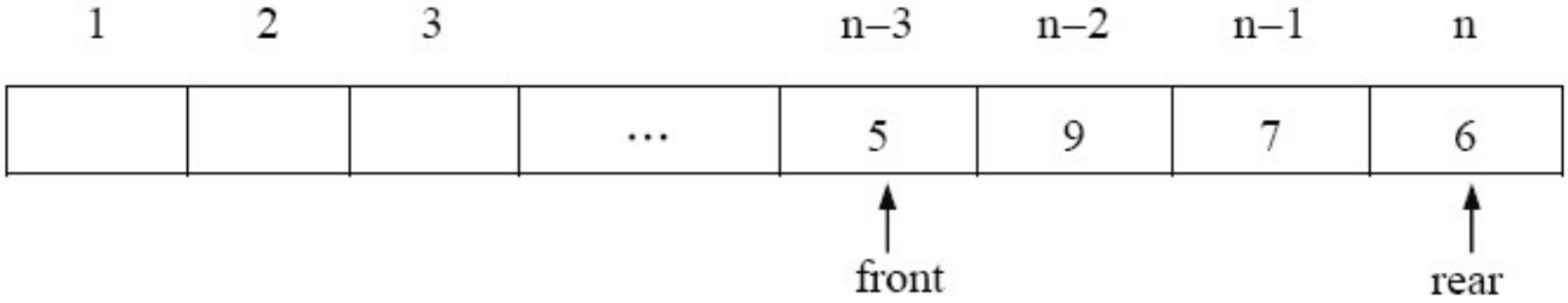
作業02-04

請實作Queue Abstract Data Type

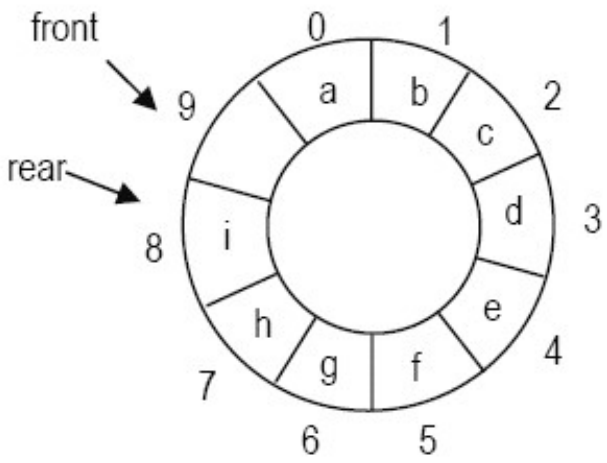
```
public class Queue {
        public boolean isEmpty();
        public boolean isFull();
        public void push(int value);
        public int pop();
        public int front(); //return the element at the front of the queue
        public int rear(); //return the element at the rear of the queue
}
```

Circular Queue

- queue中的elements會往右移，當queue的rear=capacity-1時，須將整個queue中的elements往左移，移動陣列非常耗時



- 為了解決此問題，可使用Circular Queue



Circular Queue Implementation(1)

```
public class Ch03MyIntCircularQueue {
   private int[] myQueue;
   private int front;
   private int rear;
   private int capacity;
   Ch03MyIntCircularQueue( int queueCapacity) throws Exception{
      if(queueCapacity<1) throw new Exception("QueueCapacity需大於0");
      this.capacity = queueCapacity;
      myQueue = new int[queueCapacity];
      front = 0;
      rear = 0;
   }
```

Circular Queue Implementation(2)

```
public boolean isEmpty(){
   if(this.front == this.rear)
      return true;
   else
      return false;
}
public boolean isFull(){
   if((rear+1)/capacity==front)
      return true;
   else
      return false;
}
public int front() throws Exception{
   if(isEmpty()) throw new Exception("目前Queue是空的!");
   return myQueue[(front+1)%capacity];
}
public int rear() throws Exception{
   if(isEmpty()) throw new Exception("目前Queue是空的!");
   return myQueue[rear];
}
```

Circular Queue Implementation(3)

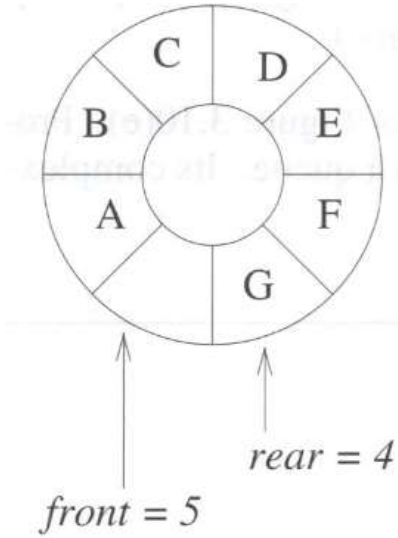
```
public void push(int value) throws Exception{
      if(isFull()) throw new Exception("目前Queue已滿，不可push物件!");
      rear= (rear+1)%capacity;
      myQueue[rear] = value;
   }

   public int pop() throws Exception{
      if(isEmpty()) throw new Exception("目前Queue是空的，沒有物件可pop!");
      front = front +1;
      return myQueue[front];
   }
}
```

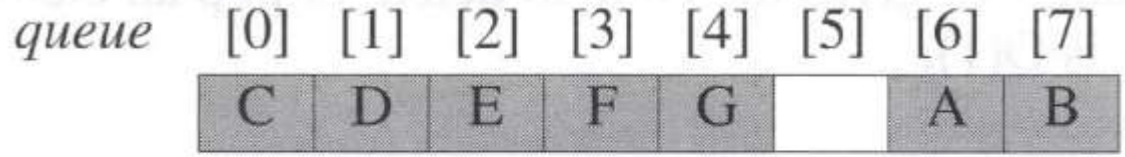
作業02-05

- 請實作Circular Queue

Queue動態增加大小

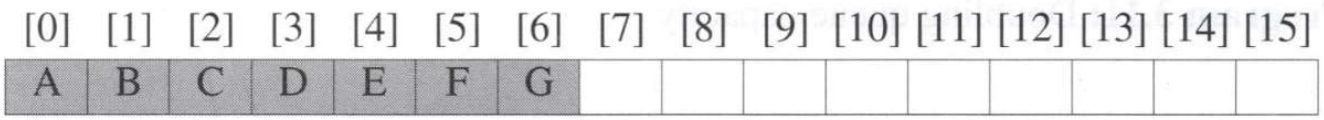


(a) A full circular queue



front = 5, *rear* = 4

(b) Flattened view of circular full queue



front = 15, *rear* = 6

(e) Alternative configuration

1. Create a new array newQueue of twice the capacity
2. Copy the second segment(i.e., the elements queue[front+1] through queue[capacity-1]) to positons in newQueue beginning at 0
3. Copy the first segment(i.e., the elements queue[0] through queue[rear]) to positions in newQueue beginning at capacity-front-1