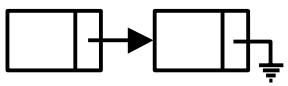
Data Structures & Algorithms

STACKS & QUEUES



Lists (รายการ)

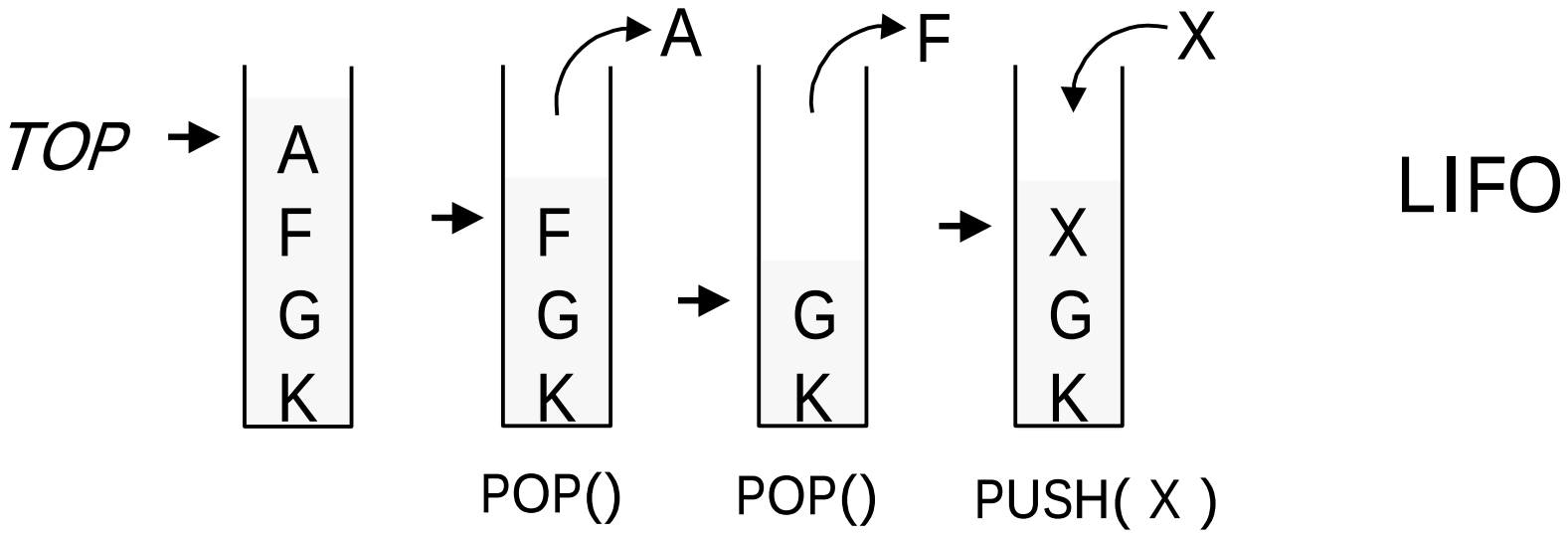
- a list is a sequence of zero of more elements of a given type.
- elements of a list are linearly ordered according to their position on the list.

a,b,d,f,e,c,f,g,c

- operations on a list :
  - Length, Insert, Delete, Locate, Retrieve, Next, Previous, First, End, Initialize,...

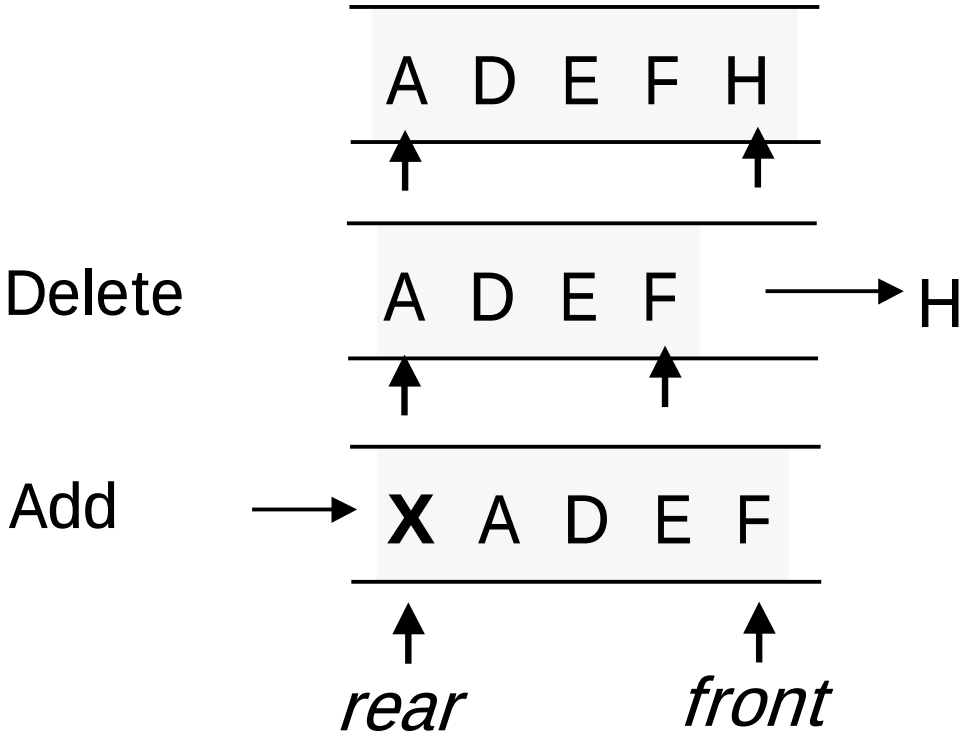
Stacks (กองซ้อน)

- a stack is an ordered list in which all insertions and deletions are made at <u>one</u> end, called the *top* of the stack.
- PUSH - add an item to a stack.
- POP - remove an item from a stack.



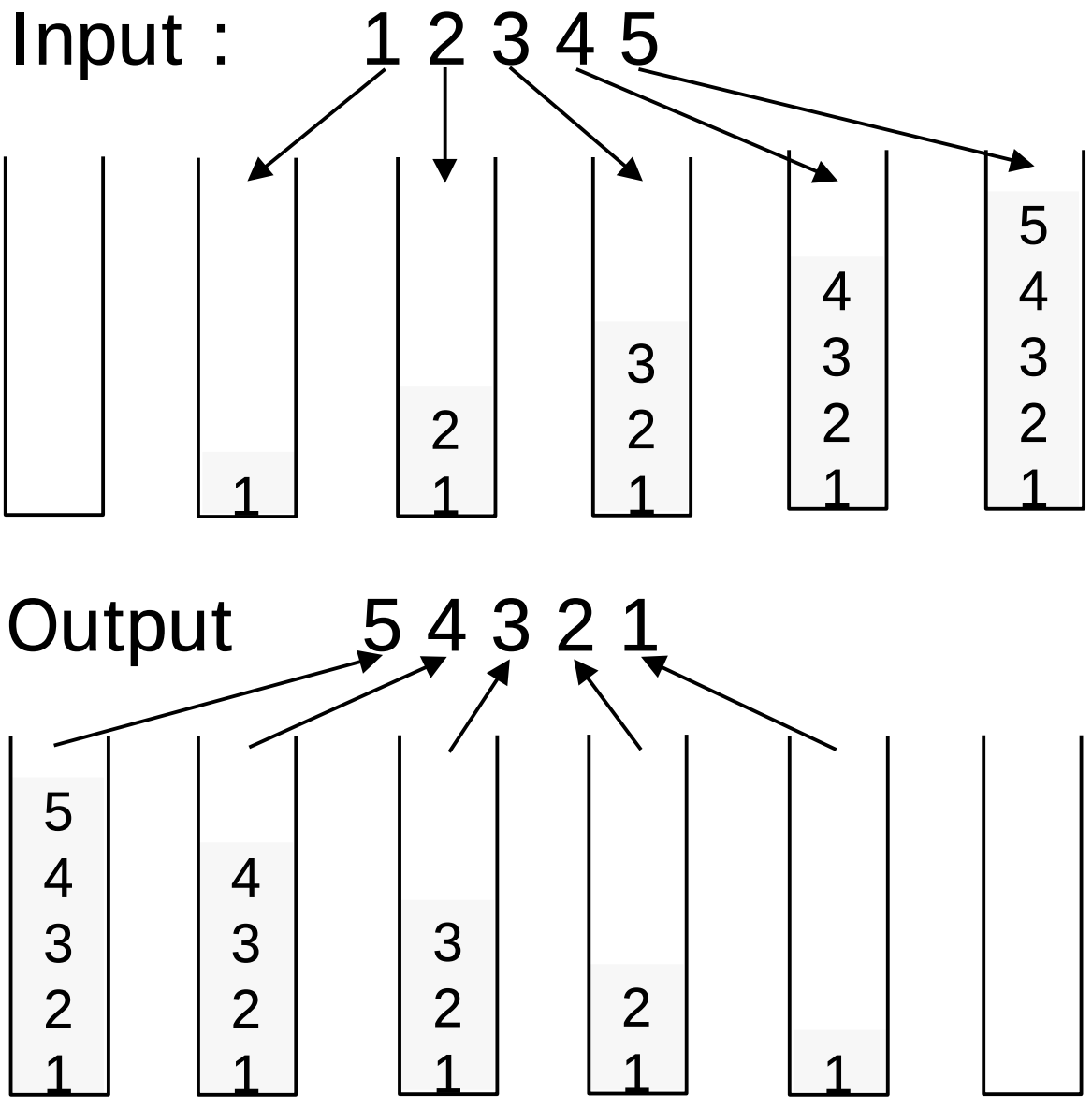
Queues (แถวคอย)

- A queue is an ordered list in which
  - all insertions take place at <u>one</u> end (*rear*),
  - all deletions take place at <u>the other</u> end (*front*)

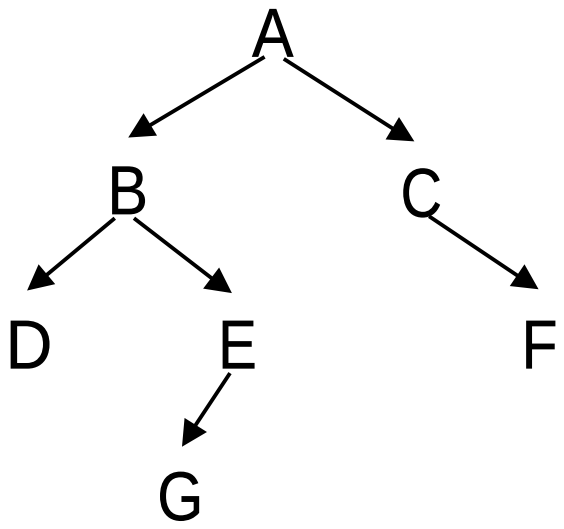


FIFO

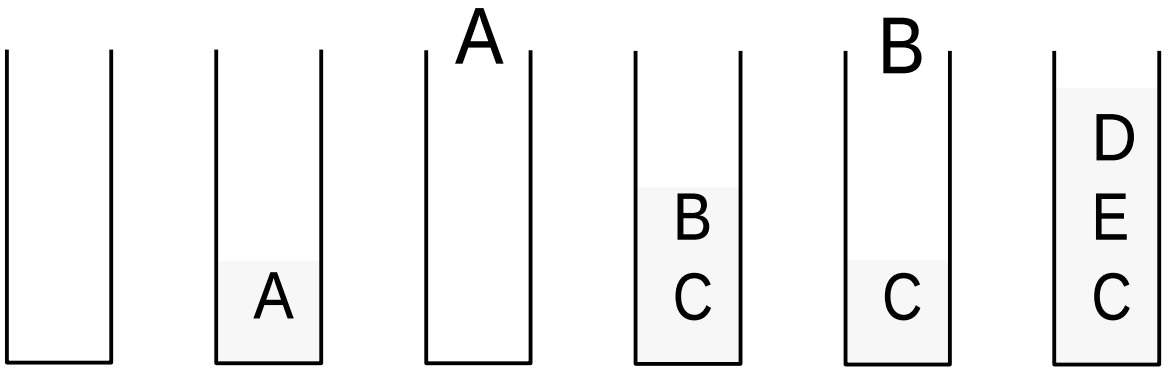
Reversing a Line

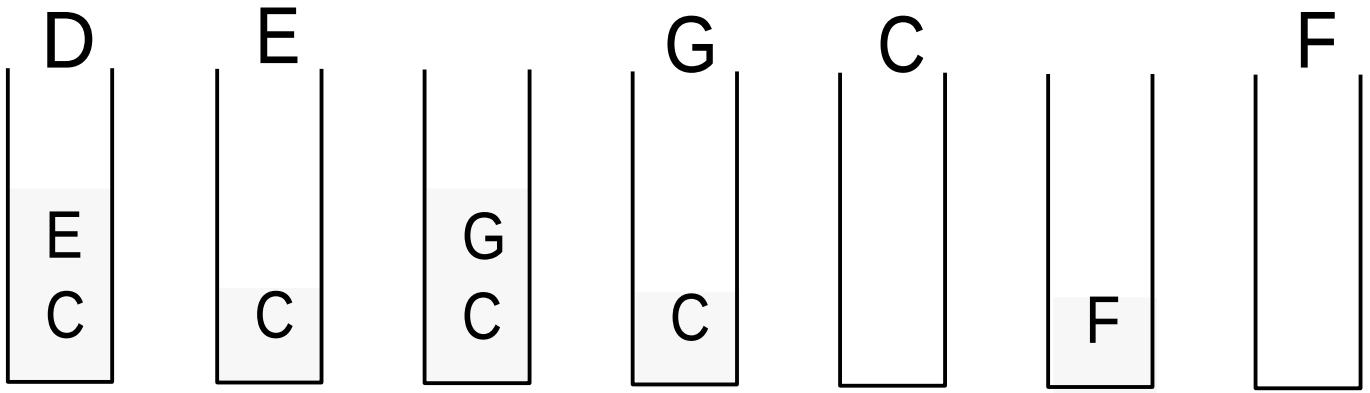


Depth-First Search (การค้นหาตามแนวลึก)

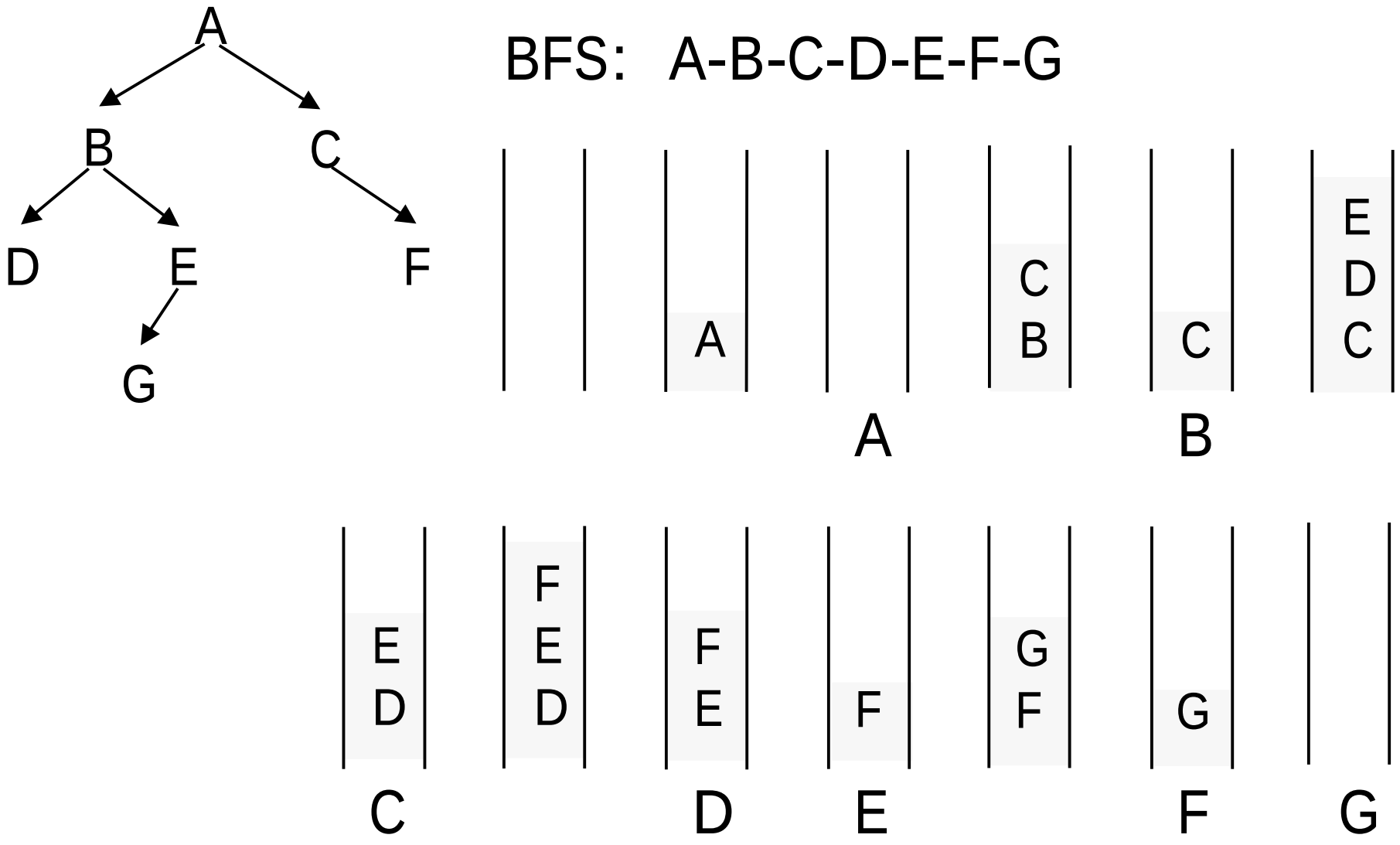


DFS: A-B-D-E-G-C-F





Breadth-First Search (การค้นหาตามแนวกว้าง)



Depth-First Search : Algorithm

```
DFS( root )
{
  InitializeStack( st );
  Push( root, st );
  do {
    node = Pop( st );
    Visit( node );
    for each child cnode of node
      Push( cnode, st );
  } until( StackIsEmpty( st ) );
}
```

Breadth-First Search : Algorithm

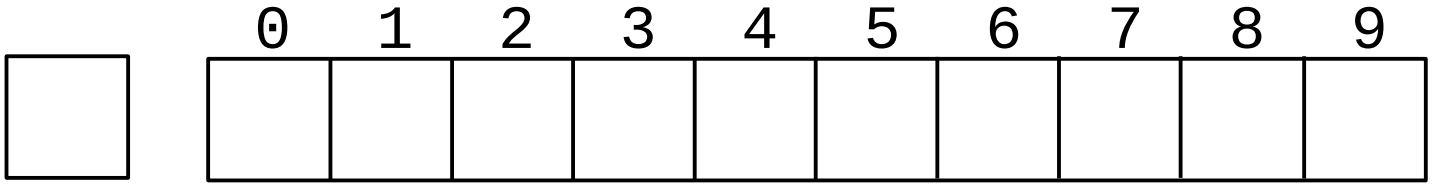
```
BFS( root )
{
  InitializeQueue( q );
  AddQueue( root, q );
  do {
    node = DeleteQueue( q );
    Visit( node );
    for each child cnode of node
      AddQueue( cnode, q );
  } until( QueueIsEmpty( q ) );
}
```

Stack & Queue Operations

- Stack
  - **InitializeStack**(s)
  - **Push**(item,s)
  - **Pop**(item,s)
  - **StackIsEmpty**(s)
  - **StackIsFull**(s)
- Queue
  - **InitializeQueue**(q)
  - **AddQueue**(item,q)
  - **DeleteQueue**(item,q)
  - **QueueIsEmpty**(q)
  - **QueueIsFull**(q)

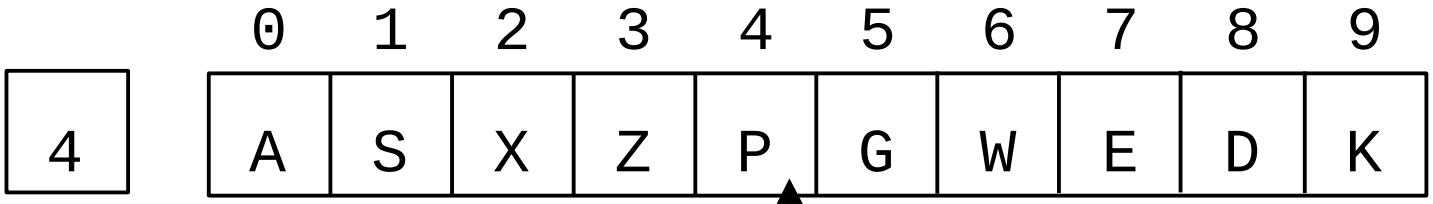
Array Implementation of Stacks

```
#define    STACKSIZE    10

typedef    char    ItemType;
typedef    struct  StackTag  {
    int          top;
    ItemType     entry[ STACKSIZE ];
} StackType;

StackType    stack;
```



Array Implementation of Stacks

```
StackType    stack;
```



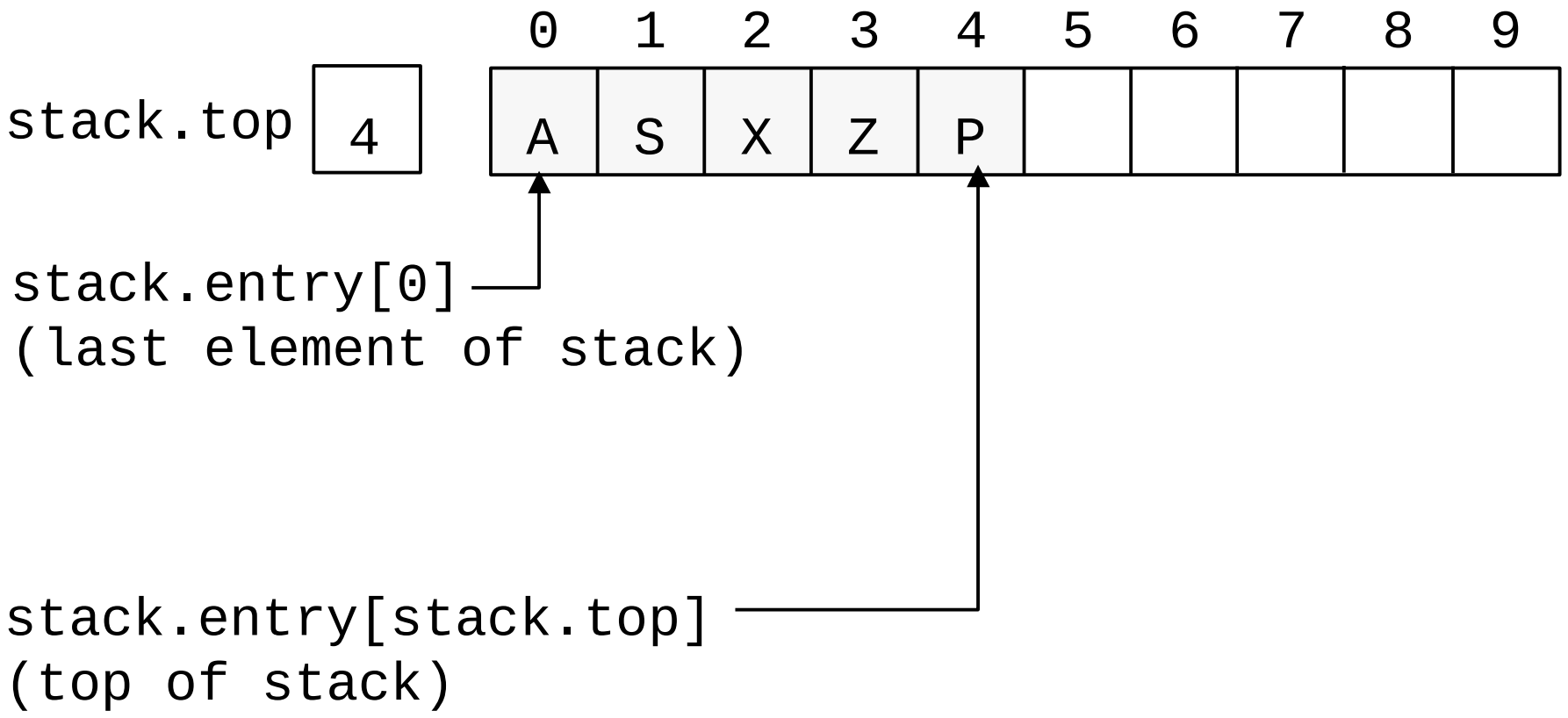

```
stack.top

stack.entry[stack.top]
```

คำถาม : ข้อมูลตัวท้ายสุดของกองซ้อนอยู่ที่ใด ?

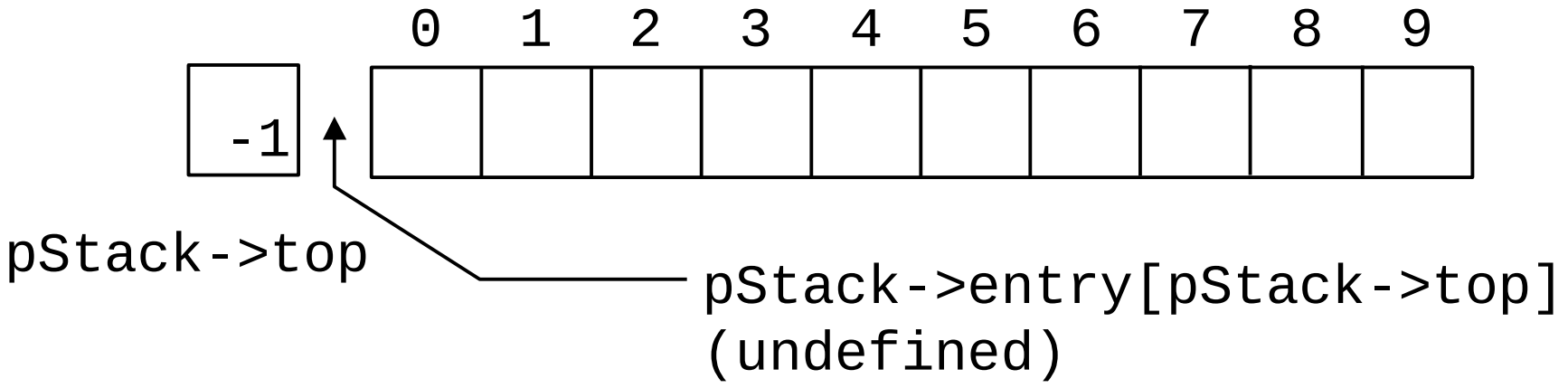
```
   stack.entry[0]    หรือ
   stack.entry[ STACKSIZE-1 ]
```

Array Implementation of Stack

```
StackType    stack;
StackType    *pStack = &stack
```

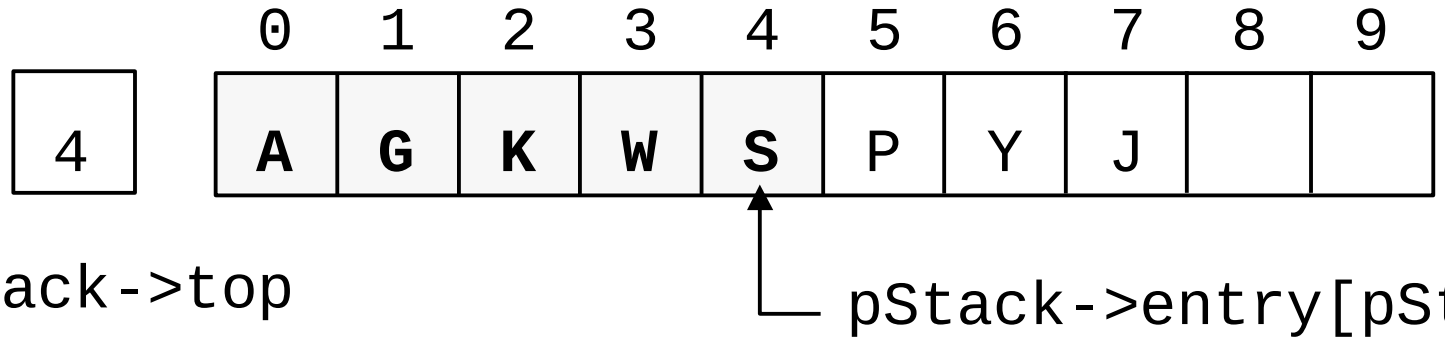


Array Implementation of Stacks



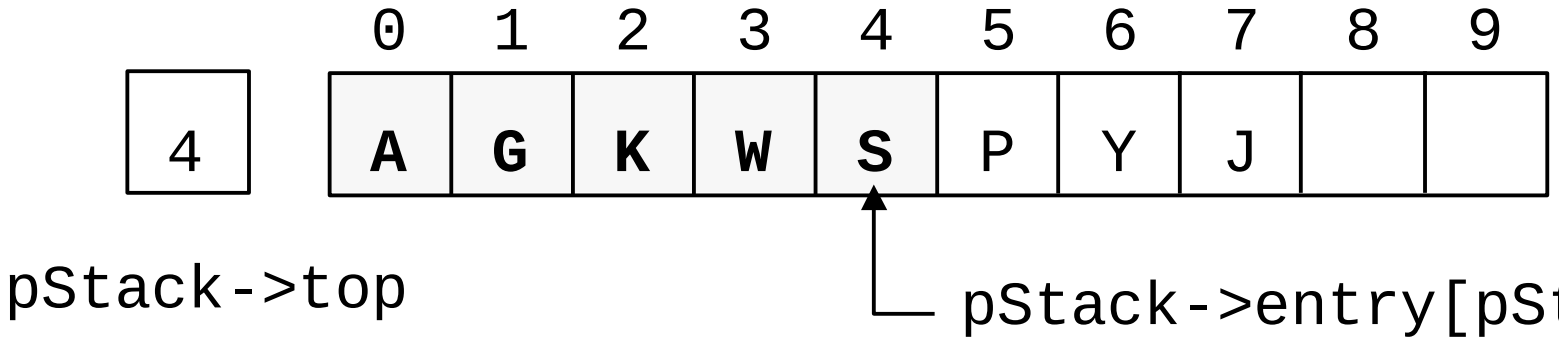

```
void  InitializeStack( StackType *pStack )
{
  pStack->top = -1;
}
Boolean  StackIsEmpty( StackType *pStack )
{
  return(  pStack->top <= -1  );
}
```

Array Implementation of Stacks



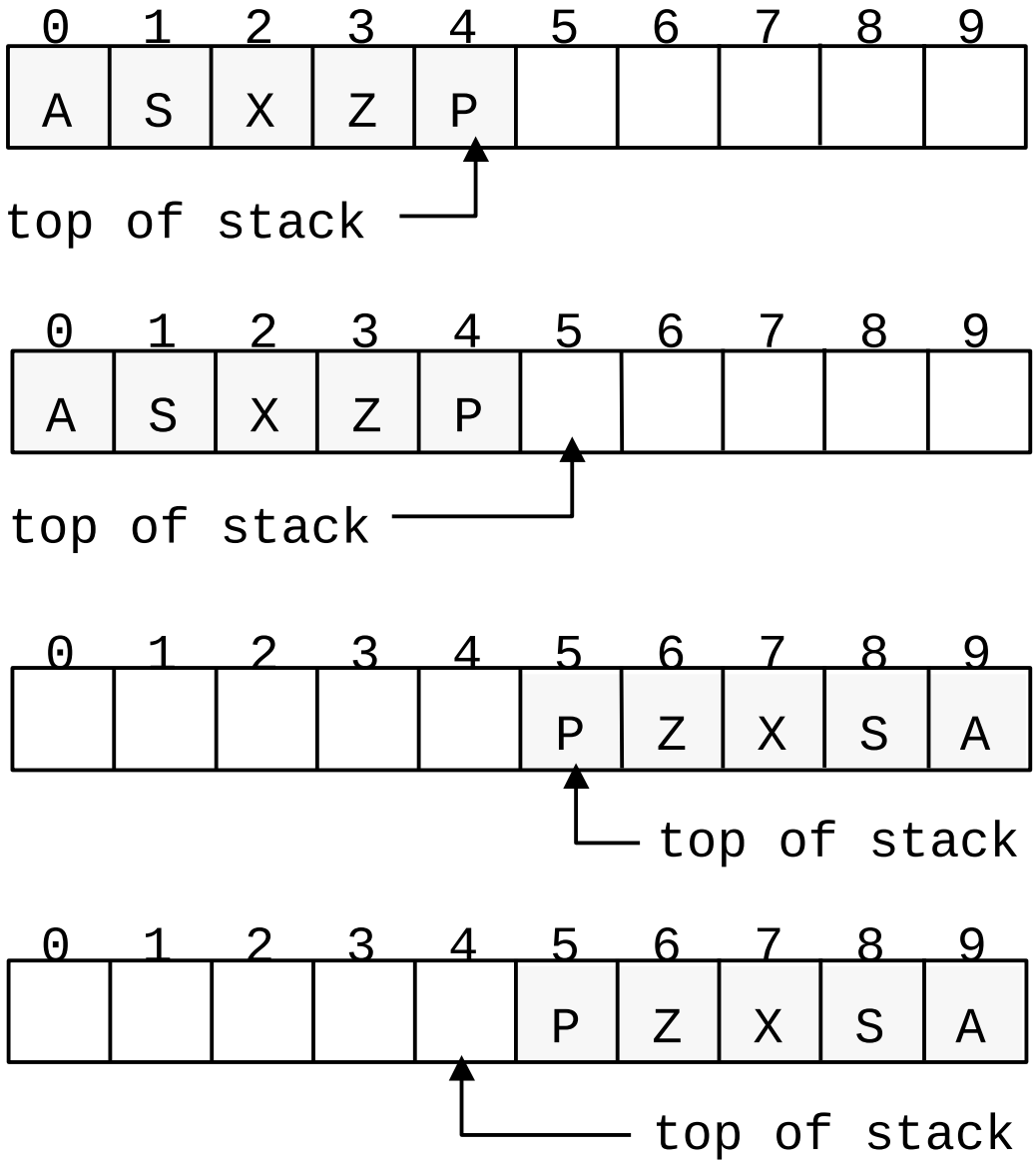

```
Boolean  StackIsFull( StackType *pStack )
{
  return( pStack->top >= STACKSIZE-1 );
}
void  Push( ItemType item, StackType *pStack )
{
  if ( StackIsFull( pStack ) )
    Error("Stack is full");
  else
    pStack->entry[++(pStack->top)] = item;
}
```

Array Implementation of Stacks



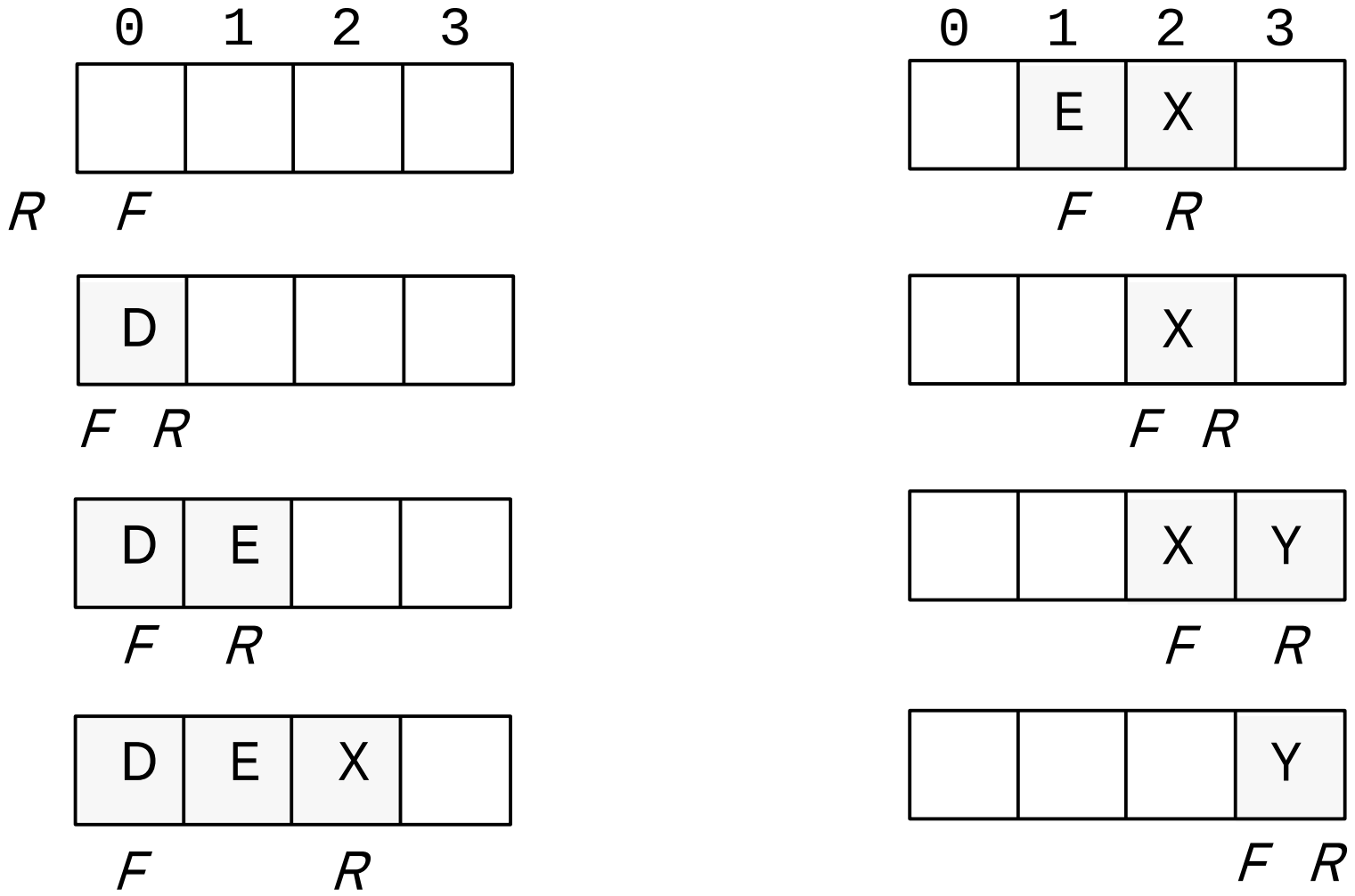

```
void  Pop( ItemType *pItem, StackType *pStack )
{
  if ( StackIsEmpty( pStack ) )
    Error("Stack is empty");
  else
    *pItem = pStack->entry[(pStack->top)--];
}
```

ไม่จำเป็นต้องสร้างกองซ้อนด้วยวิธีที่กล่าวมา

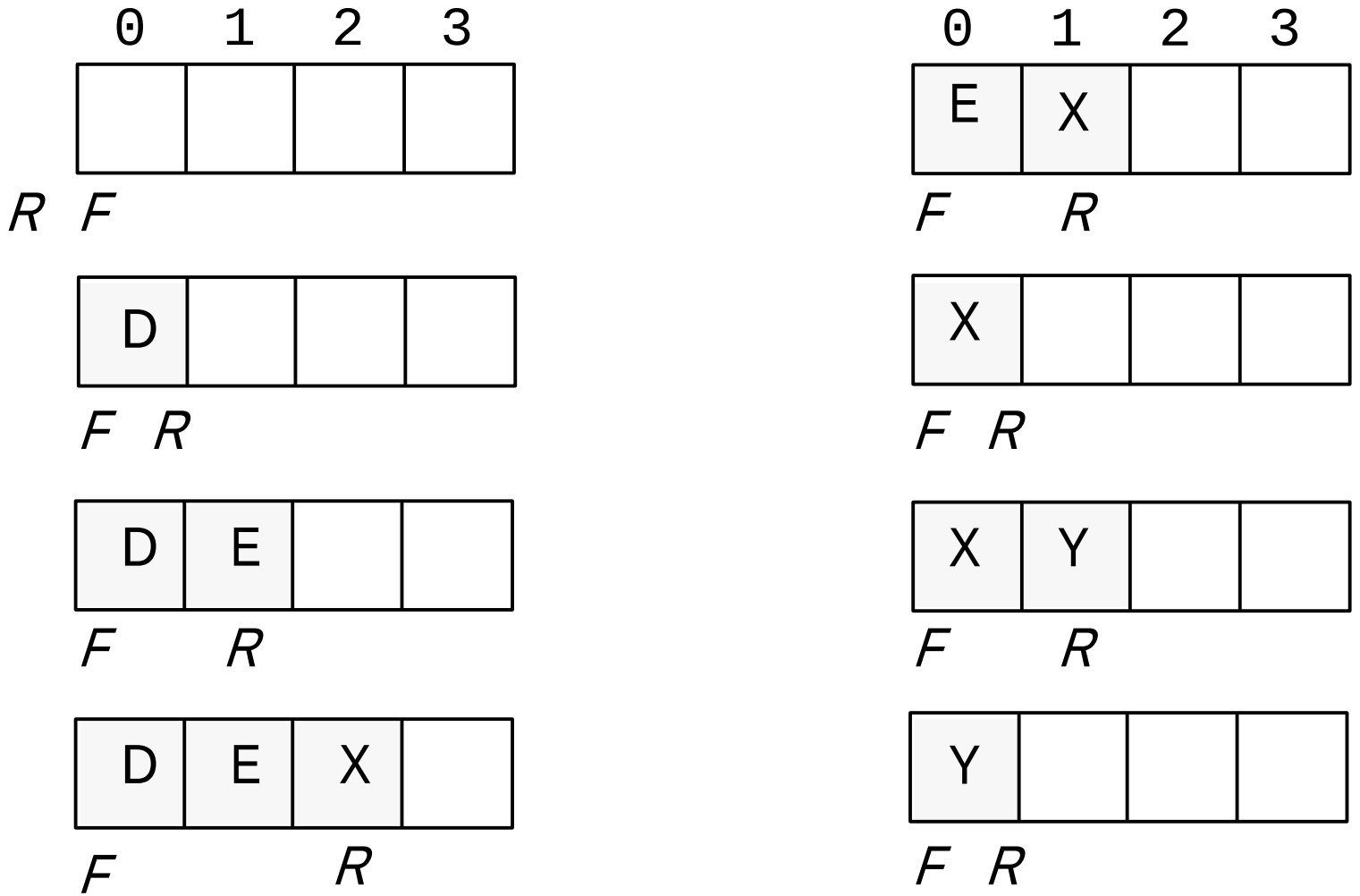


และยังมีวิธีอื่นๆอีก

Array Implementation of Queues



Array Implementation of Queues



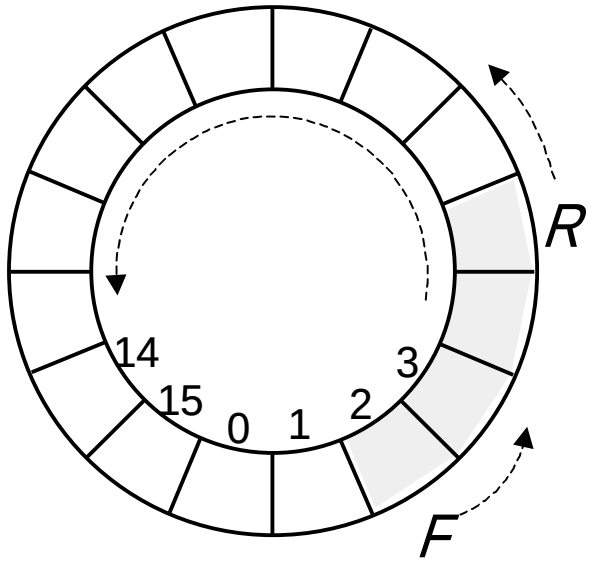
Circular Arrays

AddQueue

$R = (R + 1) \% \text{QUEUESIZE}$

DeleteQueue

$F = (F + 1) \% \text{QUEUESIZE}$

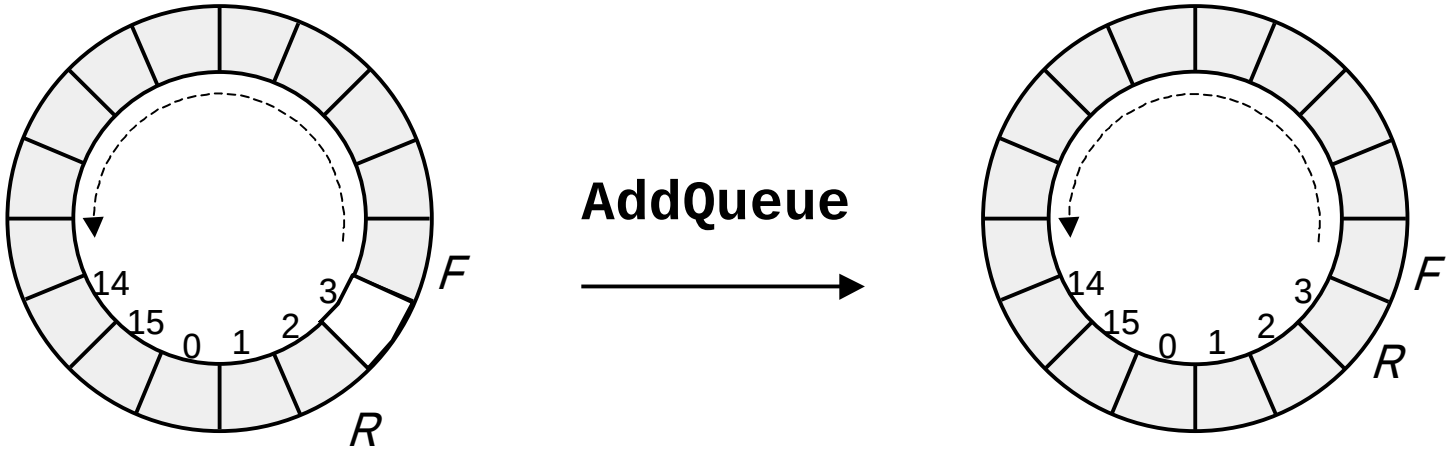


Modular Arithmetic :

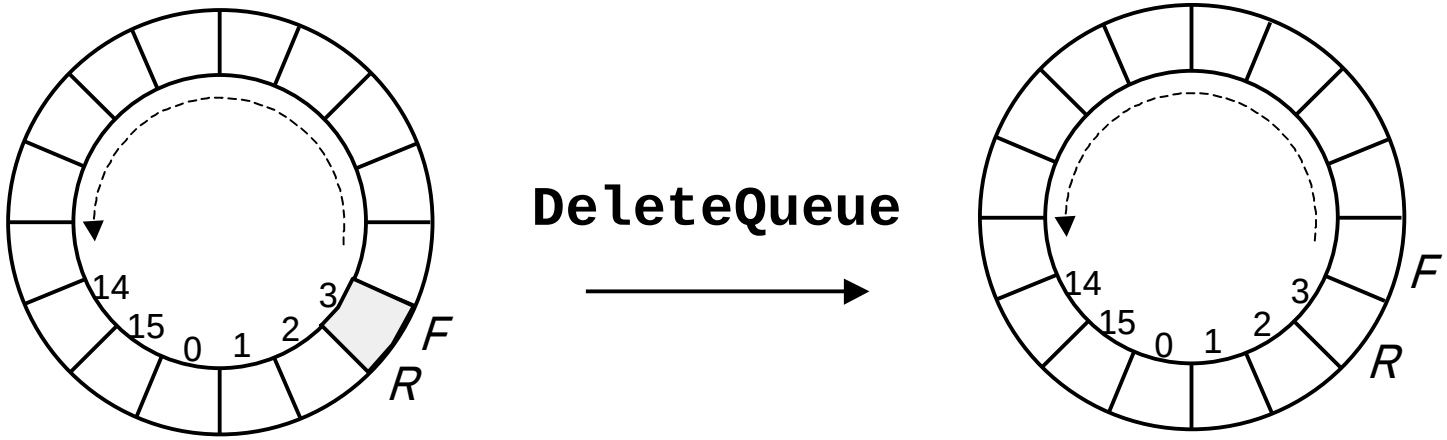
```
11 % 16  =  11 (0*16 + 11 = 11)
18 % 7   =  4  (2*7  + 4  = 18)
22 % 11  =  0  (2*11 + 0  = 22)
```

Circular Array

Is queue full or empty ?



F == (*R*+1)%QSIZE



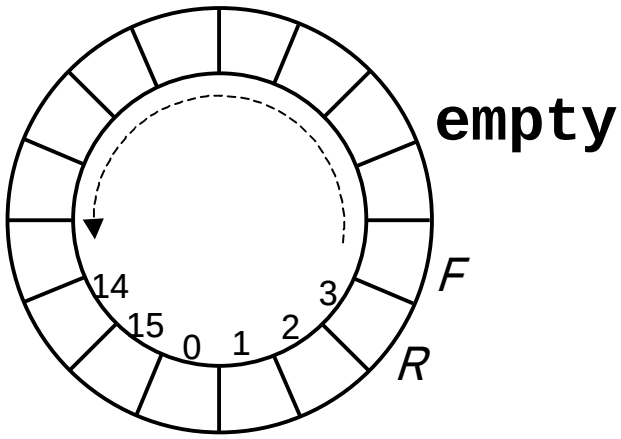
Possible Solutions

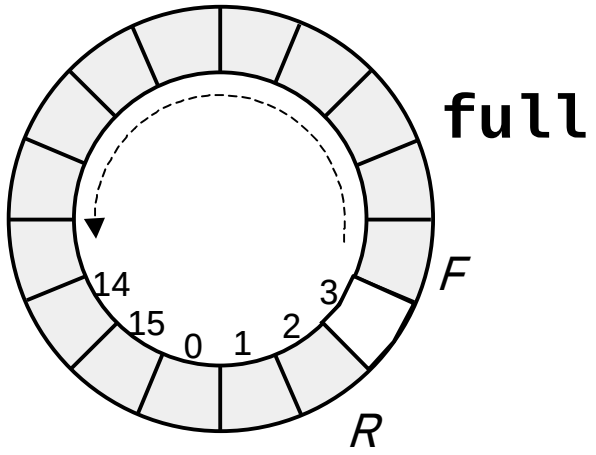
Counter

```
empty  : count = 0
full : count = QUEUESIZE
```

Empty position

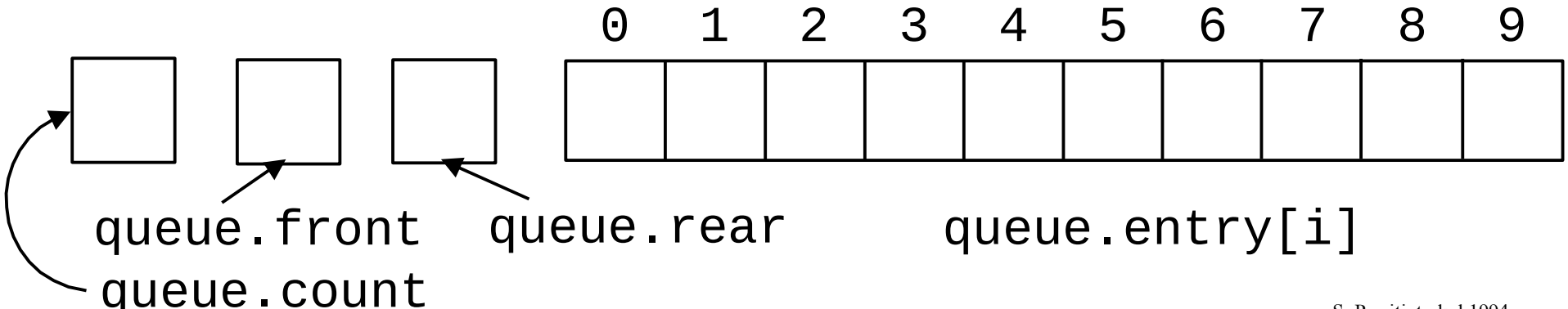
```
empty  : front == (rear+1) % QUEUESIZE
full : front == (rear+2) % QUEUESIZE
```





Queue: Circular Array with a Counter

```
#define    QUEUESIZE    10
typedef    char    ItemType;
typedef    struct  QueueTag  {
     int         count;
     int         front, rear;
     ItemType    entry[ QUEUESIZE ];
} QueueType;

QueueType    queue;
QueueType    *pQueue = &queue;
```



Queue: Circular Array with a Counter

```
void  InitializeQueue( QueueType *pQueue )
{
  pQueue->count = 0;
  pQueue->front = 1; pQueue->rear = 0;
}

Boolean  QueueIsEmpty( QueueType *pQueue )
{
  return(  pQueue->count == 0  );
}
```

Queue: Circular Array with a Counter

```
Boolean  QueueIsFull( QueueType *pQueue )
{
  return( pQueue->count == QUEUESIZE );
}
void  AddQueue( ItemType item, QueueType *pQueue )
{
  if ( QueueIsFull( pQueue ) )
    Error("Queue is full");
  else {
    (pQueue->count)++;
    pQueue->rear = (pQueue->rear + 1) % QUEUESIZE;
    pQueue->entry[pQueue->rear] = item;
  }
}
```

Queue: Circular Array with a Counter

```
void  DeleteQueue( ItemType *pItem, QueueType *pQueue
{
  if ( QueueIsEmpty( pQueue ) )
    Error("Queue is empty");
  else {
    *pItem = pQueue->entry[pQueue->front];
    (pQueue->count)--;
    pQueue->front = (pQueue->front + 1) % QUEUESIZE;
  }
}
```