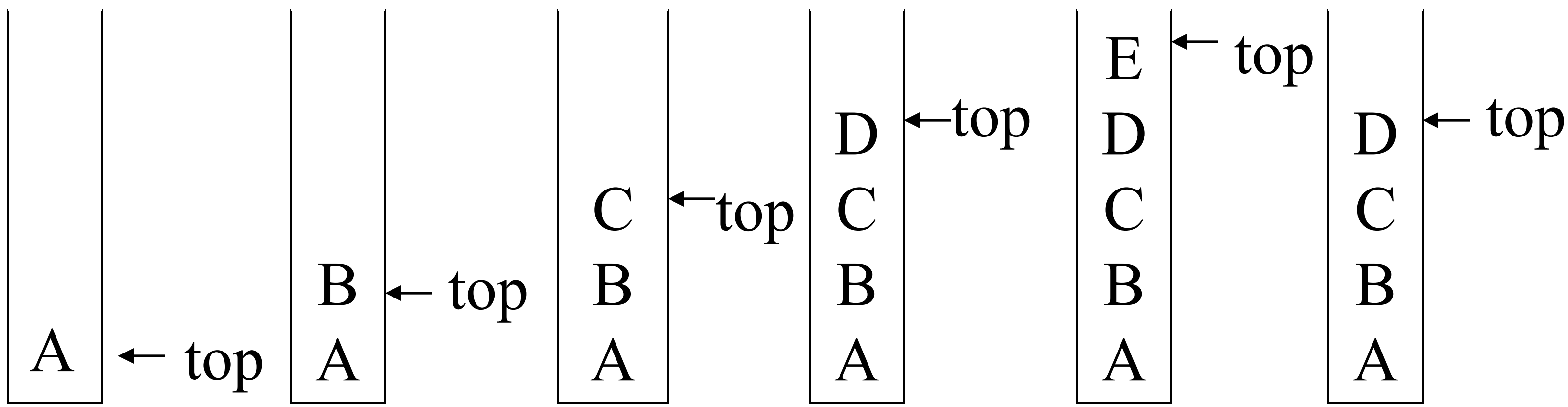
CHAPTER 3

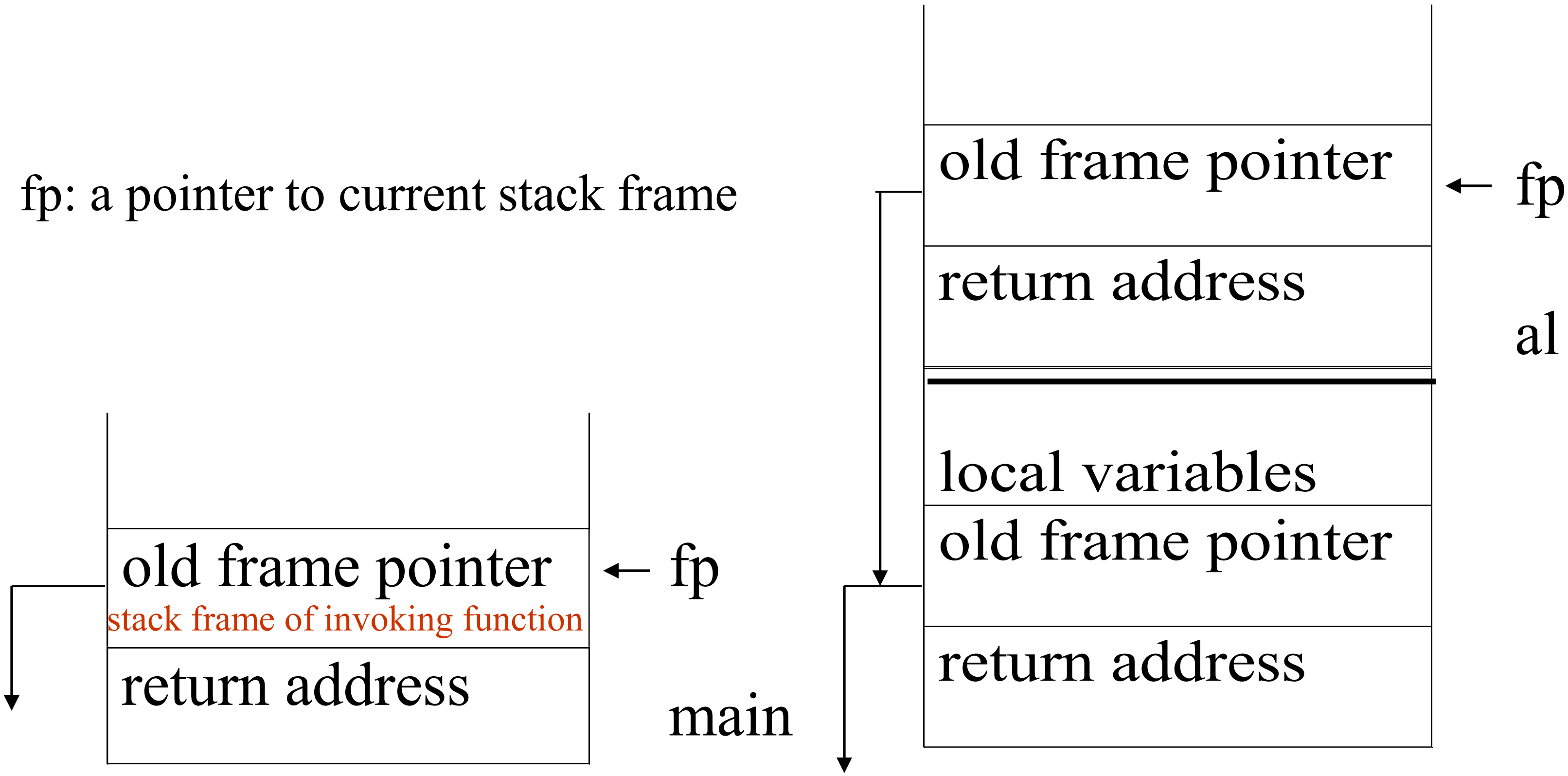
STACKS AND QUEUES

stack: a Last-In-First-Out (LIFO/FILO) list



an application of stack: stack frame of function call (activation record)

fp: a pointer to current stack frame



system stack before a1 is invoked　　system stack after a1 is invoked

(a)　　(b)

System stack after function call a1

abstract data type for stack

structure *Stack* is

objects: a finite ordered list with zero or more elements.

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer

Stack CreateS(*max_stack_size*) ::=

create an empty stack whose maximum size is *max_stack_size*

Boolean IsFull(*stack, max_stack_size*) ::=

if (number of elements in *stack* == *max_stack_size*)

return TRUE

else return FALSE

Stack Push(*stack, item*) ::=

if (IsFull(*stack*)) *stack_full*

else insert *item* into top of *stack* and **return**

Boolean IsEmpty(*stack*) ::=

if(*stack* == CreateS(*max_stack_size*))

return TRUE

else return FALSE

Element Pop(*stack*) ::=

if(IsEmpty(*stack*)) **return**

else remove and return the *item* on the top of the stack.

Abstract data type *Stack*

Catalan Number

假設有n筆資料，依序執行push，中間可穿插pop，試問其合法的排列組合有多少種可能？

$$\frac{\binom{2n}{n}}{n+1}$$

Implementation: using array

```
Stack CreateS(max_stack_size) ::=
    #define MAX_STACK_SIZE [1..n]  /* maximum stack size */
    typedef struct {
            int key;
            /* other fields */
            } element;
    element stack[MAX_STACK_SIZE];
    int top = 0;

    Boolean IsEmpty(Stack) ::= top== 0;

    Boolean IsFull(Stack) ::= top == MAX_STACK_SIZE;
```

Add to a stack (Push)

```
void push(int *top, element item)
{
 /* add an item to the global stack */
    if (*top == MAX_STACK_SIZE)  {
          stack_full( );
          return;
    }
    stack[++*top] = item;
}
```

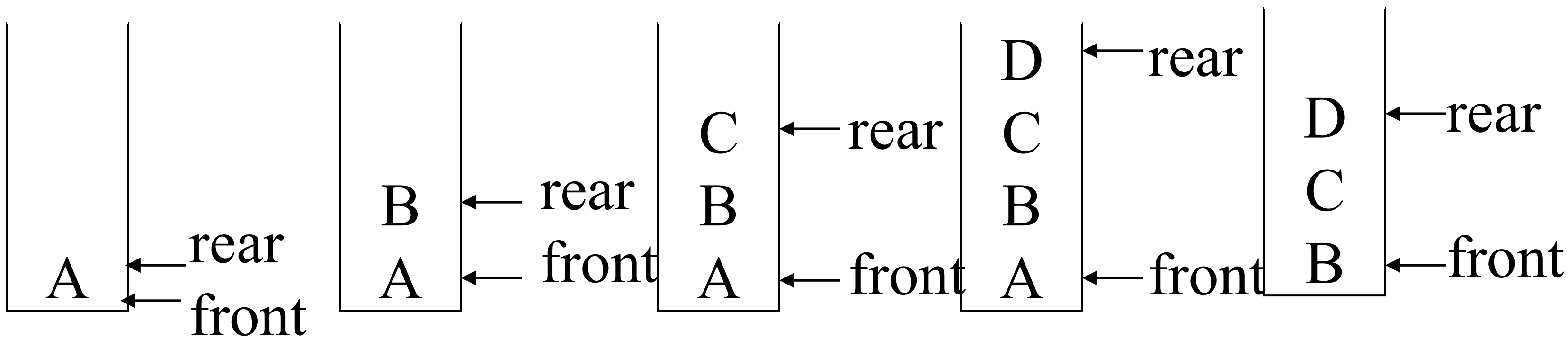
top=top+1
***top=item**

Delete from a stack (Pop)

```
element pop(int *top)
{
 /* return the top element from the stack */
    if (*top == 0)
        return stack_empty( );  /* returns and error key */
    return stack[(*top)--];
 }
```

***top=item**
top=top-1

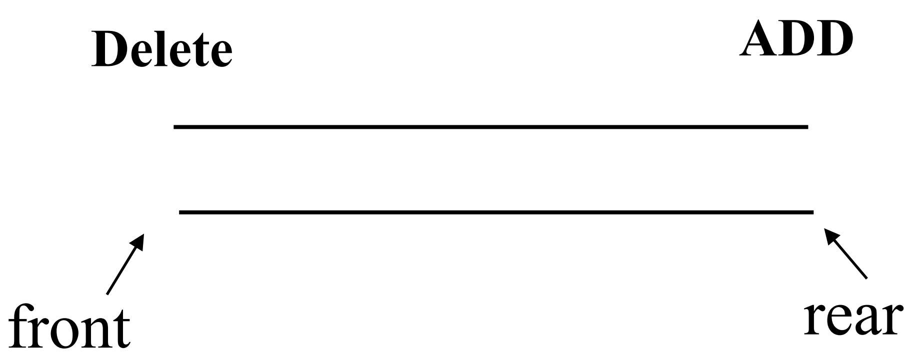
Queue: a First-In-First-Out (FIFO/LILO) list



(1) 插入元素(Add) – Rear 端
(2) 刪除元素(Delete) – Front 端

Different Queue

(1) FIFO Queue
(2) Priority Queue
(3) Double-ended Queue
(4) Double-ended Priority Queue



Application: Job scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

*__Figure 3.5:__ Insertion and deletion from a sequential queue (p.108)

Abstract data type of queue

structure *Queue* is

objects: a finite ordered list with zero or more elements.

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*,

max_ queue_ size ∈ positive integer

Queue CreateQ(*max_queue_size*) ::=

create an empty queue whose maximum size is *max_queue_size*

Boolean IsFullQ(*queue*, *max_queue_size*) ::=

if(number of elements in *queue* == *max_queue_size*)

return *TRUE*

else return *FALSE*

Queue AddQ(*queue*, *item*) ::=

if (IsFullQ(*queue*)) *queue_full*

else insert *item* at rear of *queue* and return *queue*

Boolean IsEmptyQ(*queue*) ::=

if (*queue* ==CreateQ(*max_queue_size*))

return *TRUE*

else return *FALSE*

Element DeleteQ(*queue*) ::=

if (IsEmptyQ(*queue*)) **return**

else remove and return the *item* at front of queue.

Abstract data type (*Queue)*

Implementation 1: using array

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE [1…n]
typedef struct {
            int key;
            /* other fields */
            } element;
element queue[MAX_QUEUE_SIZE];
int rear = 0;
int front = 0;
```

Add to a queue

```
void addq(int *rear, element item)
{
/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE) {
      queue_full( );
      return;
   }
   queue [++*rear] = item;
}
```

Add to a queue

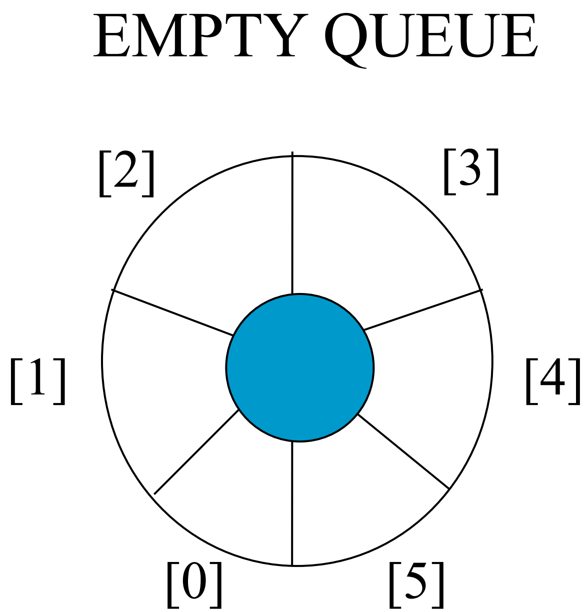
Delete from a queue

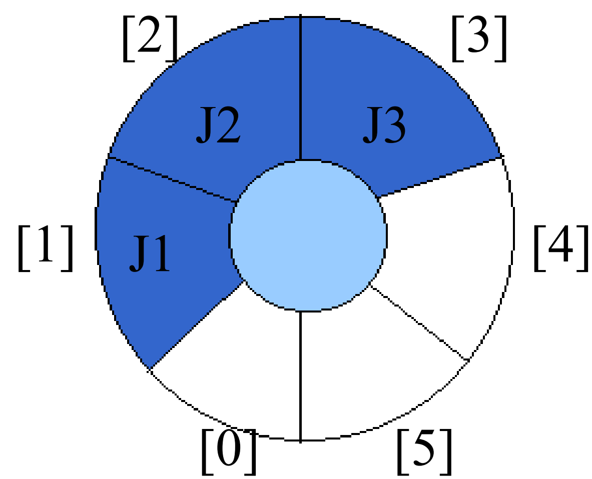
```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
    if ( *front == * rear)
        return queue_empty( );     /* return an error key */
   else
    return queue [++ *front];
}
```

Delete from a queue

problem: there may be available space when IsFullQ is true, i.e., movement is required.

Implementation 2: regard an array as a circular queue





front = 0
rear = 0

front = 0
rear = 3

Empty and nonempty circular queues

Create a circular queue

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE [0…n-1]
typedef struct {
        int key;
        /* other fields */
        } element;
element queue[MAX_QUEUE_SIZE];
int rear = 0;
int front = 0;
```

Add to a circular queue

```
void addq(int front, int *rear, element item)
{
/* add an item to the queue */
    *rear = (*rear +1) % MAX_QUEUE_SIZE;
     if (* front == *rear)  then queue_full( );
     return;
   }
     queue[*rear] = item;
}
```

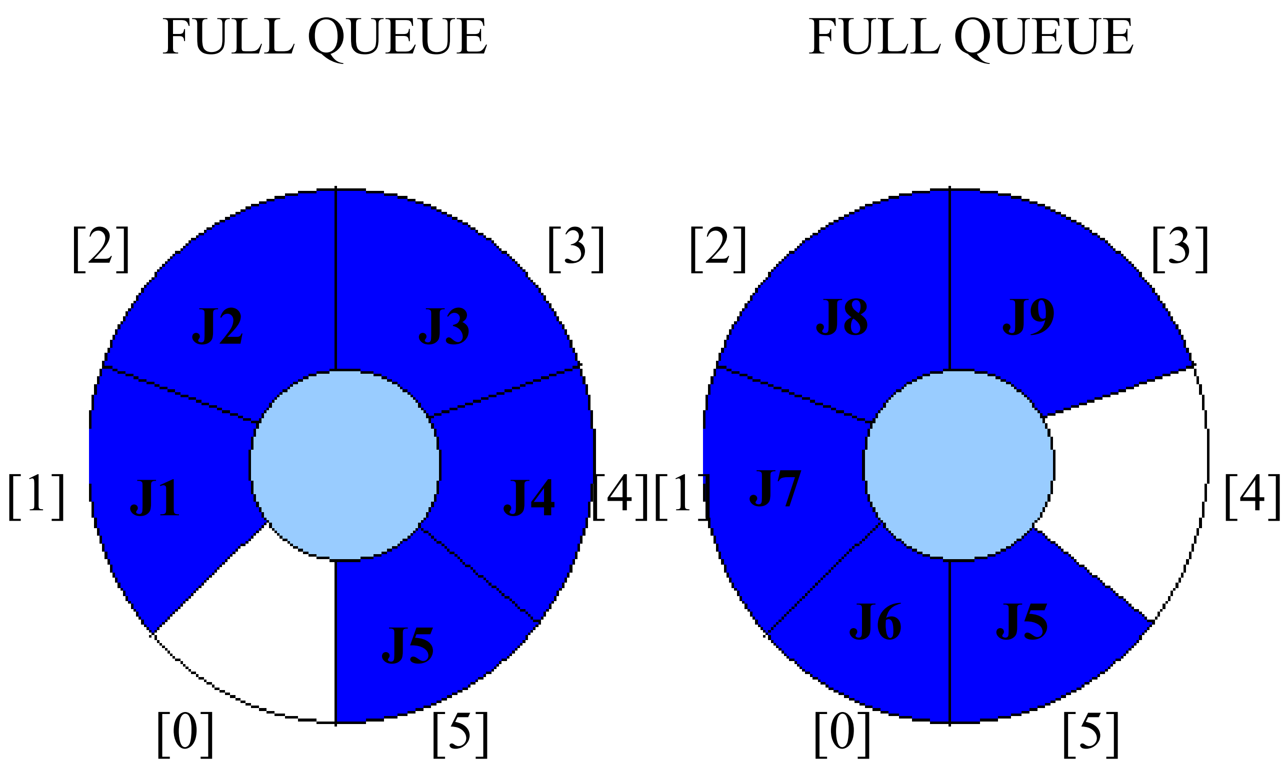
Add to a circular queue

Delete from a circular queue

```
element deleteq(int* front, int rear)
{
  element item;
  /* remove front element from the queue and put it in item */
     if (*front ==* rear) return queue_empty( );
                  /* queue_empty returns an error key */
    else
    *front = (*front+1) % MAX_QUEUE_SIZE;
     return queue[*front];
}
```

Delete from a circular queue

Problem: one space is left when queue is full



front =0
rear = 5

front =4
rear =3

<u>Full circular queues and then we remove the item</u>

Create a circular queue (with n spaces used)

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE [0…n-1]
typedef struct {
            int key;
            /* other fields */
            } element;
element queue[MAX_QUEUE_SIZE];
int rear = 0;
int front = 0;
boolean tag =0
```

Add to a circular queue (with n spaces used)

```
void addq(int front, int *rear, element item)
{
   if (*rear=*front) and tag =1) then queue_full();
   else
    *rear = (*rear +1) % MAX_QUEUE_SIZE;
     queue[*rear]=item
     if (* front == *rear)  tag=1;
}
```

Add to a circular queue

Delete from a circular queue (with n spaces used)

```
element deleteq(int* front, int rear)
{
   element item;
   if ((*front=*rear) and (tag=0)) then queue_empty();
   else
      *front = (*front+1) % MAX_QUEUE_SIZE;
      return queue[*front];
      if (*front=*rear) then tag=0;
}
```

Delete from a circular queue

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:

((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:

(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Token	Operator	Precedence[1]	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! - - + & * sizeof	decrement, increment[3] logical not one’s complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
⊠	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= ⊠	assignment	2	right-to-left
,	comma	1	left-to-right

1.The precedence column is taken from Harbison and Steele.
2.Postfix form
3.prefix form

<u>***Figure 3.12:** Precedence hierarchy for C (p.119)</u>

user **compiler**

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-

***Figure 3.13:** Infix and postfix notation (p.120)

Postfix: no parentheses, no precedence

Infix to Postfix Conversion
(Intuitive Algorithm)

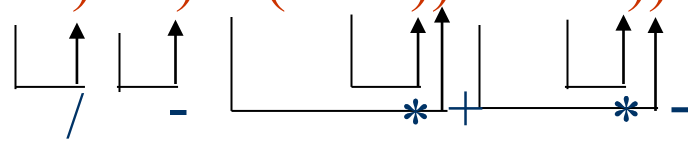
(1) Fully parenthesize expression

a / b - c + d * e - a * c -->

((((a / b) - c) + (d * e)) - a * c))

(2) All operators replace their corresponding right parentheses.

((((a / b) - c) + (d * e)) - a * c))



(3) Delete all parentheses.

ab/c-de*+ac*-

two passes

Infix	Prefix
a*b/c a/b-c+d*e-a*c a*(b+c)/d-g	/*abc -+-/abc*de*ac -/*a+bcdg

(1) evaluation
(2) transformation

***Figure 3.17:** Infix and postfix expressions (p.127)