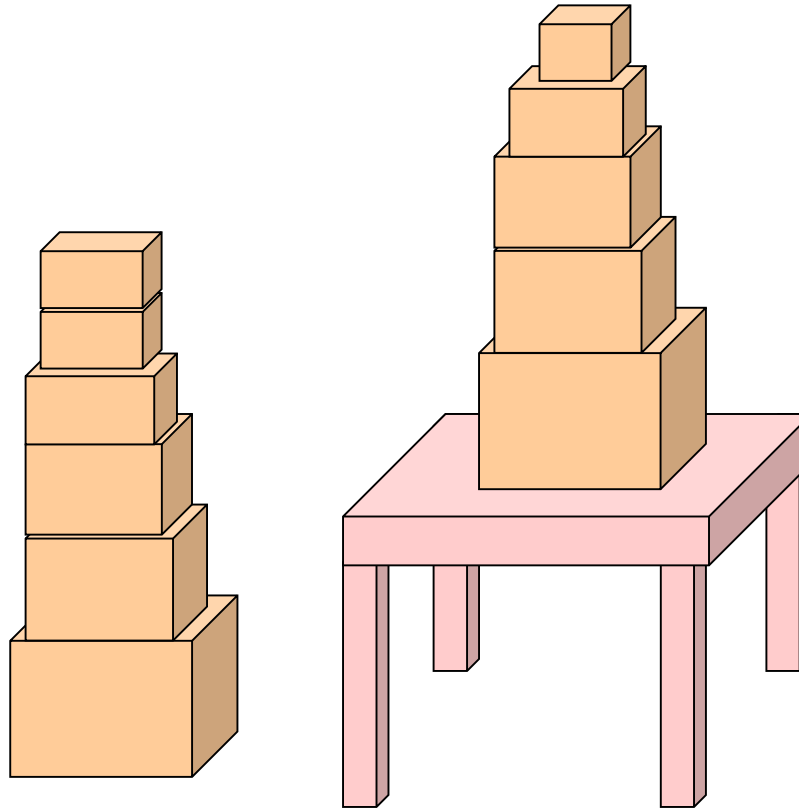


บทที่ 8

Stack

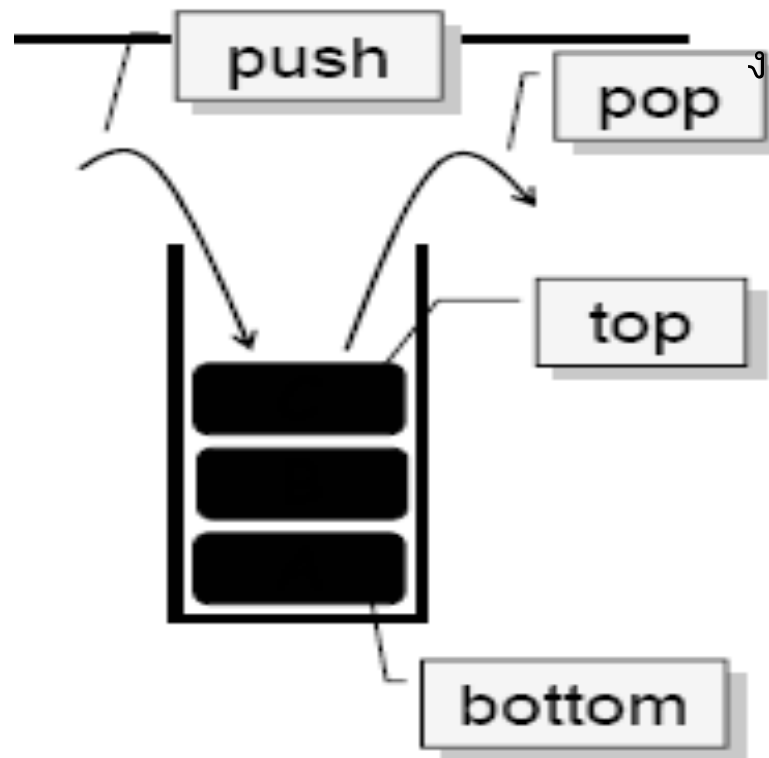
สุนทรี คุ่มไพโรจน์

โครงสร้างข้อมูลแบบกองซ้อน (Stacks)



LIFO : Last In First Out

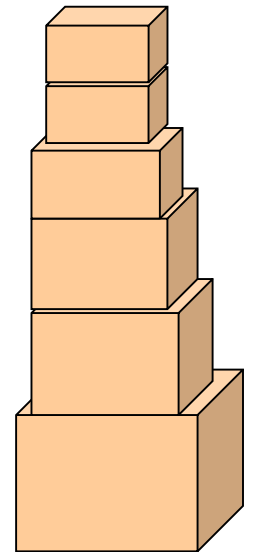
โครงสร้างข้อมูลแบบ Stack



18/07/66

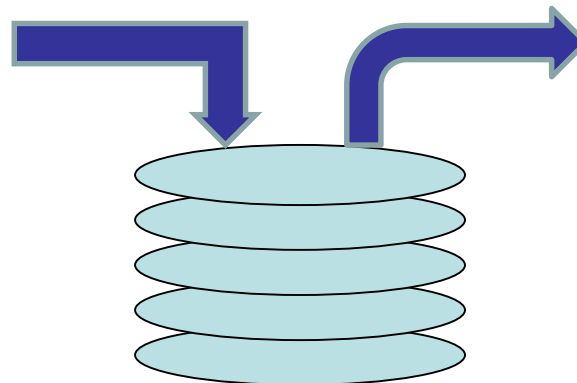
ลักษณะของโครงสร้างข้อมูลแบบ Stack

- ข้อมูลที่เก็บใน Stack จะเก็บในลักษณะวางทับกัน เช่นเดียวกับการวางจานเรียงซ้อนกัน
- ข้อมูลตัวแรกจะเป็นข้อมูลที่อยู่ล่างสุดของ Stack
- ข้อมูลสุดท้ายจะเป็นข้อมูลที่อยู่บนสุดของ Stack



ลักษณะของโครงสร้างข้อมูลแบบ Stack

- เมื่อมีการนำข้อมูลออกจาก Stack ข้อมูลที่อยู่บนสุด ข้อมูลที่นำลงสู่ Stack เป็นข้อมูลสุดท้าย เป็นข้อมูลที่จะต้องนำออกจาก Stack ก่อน
การทำงานลักษณะนี้เรียกว่า LIFO (last-in-first-out)
- การนำข้อมูลเข้าและออกจาก stack จะกระทำที่ปลายข้างเดียวเท่านั้น



ตัวอย่างการใช้งาน stack

- เพื่อแปลงนิพจน์ทางคณิตศาสตร์
- การจัดลำดับการทำงานแบบ recursive
หรือการเรียกใช้ฟังก์ชัน
- เป็นกลไกสำคัญในการทำงานของ compiler เช่น
การตรวจสอบเครื่องหมาย { } ในภาษาซี
หรือการตรวจสอบเครื่องหมายวงเล็บ

ตัวอย่างการแปลงนิพจน์ทางคณิตศาสตร์

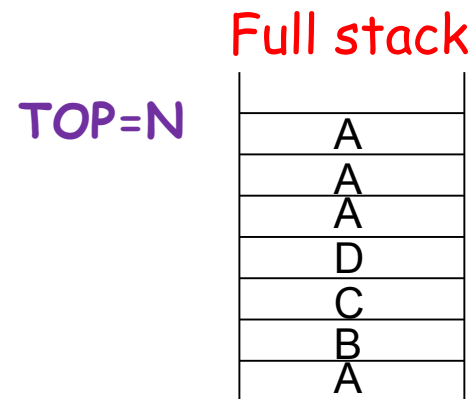
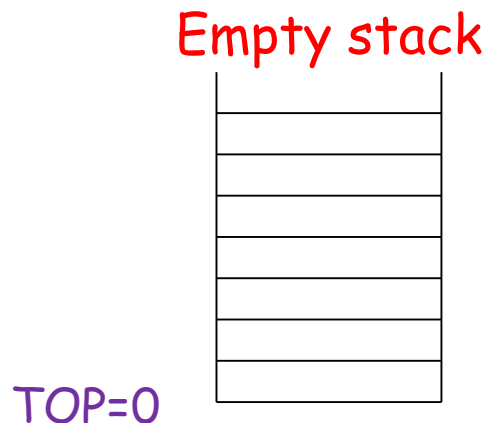
$$A + (B / (C - D)) * E$$

Implementation Stack

- การ implement Stack ทำได้ 2 วิธี คือ
 1. Array Implementation
 2. Linked List Implementation

Array representation of Stacks

- จำนวนข้อมูลสูงสุดของ stack คือ N (ขนาดของ Array)
- ตัวแปร Top แทนจำนวนข้อมูลที่มีอยู่ใน Stack
 - กรณีสแตกว่าง กำหนดให้ค่า $TOP = 0$
 - กรณี $TOP = N$ เมื่อ N คือขนาดของสแตก แสดงว่าสแตกเต็ม



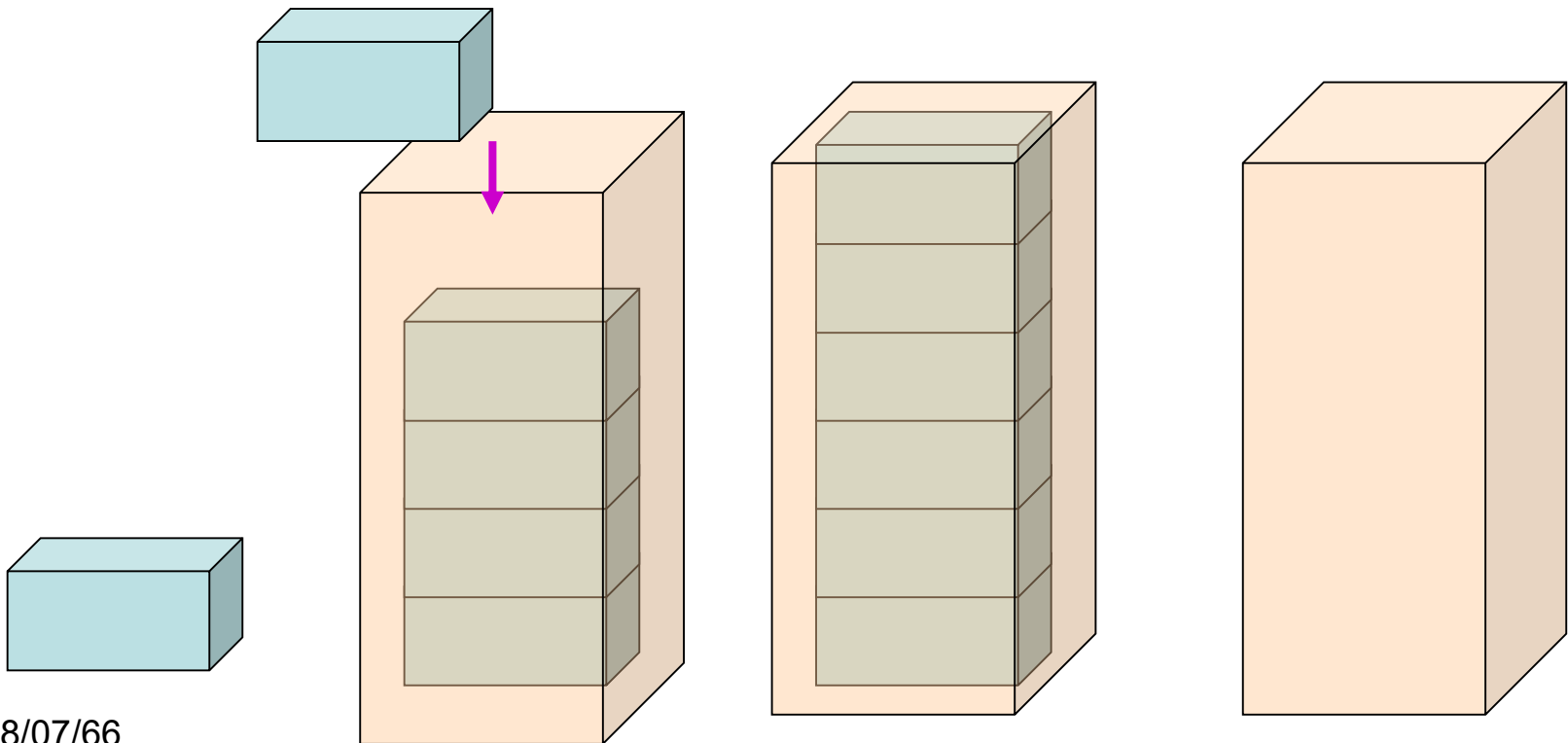
Operation ของ Stack

1. Push Stack เป็น operation สำหรับนำข้อมูลลงใน Stack
2. Pop Stack เป็น operation สำหรับนำข้อมูลออกจาก Stack

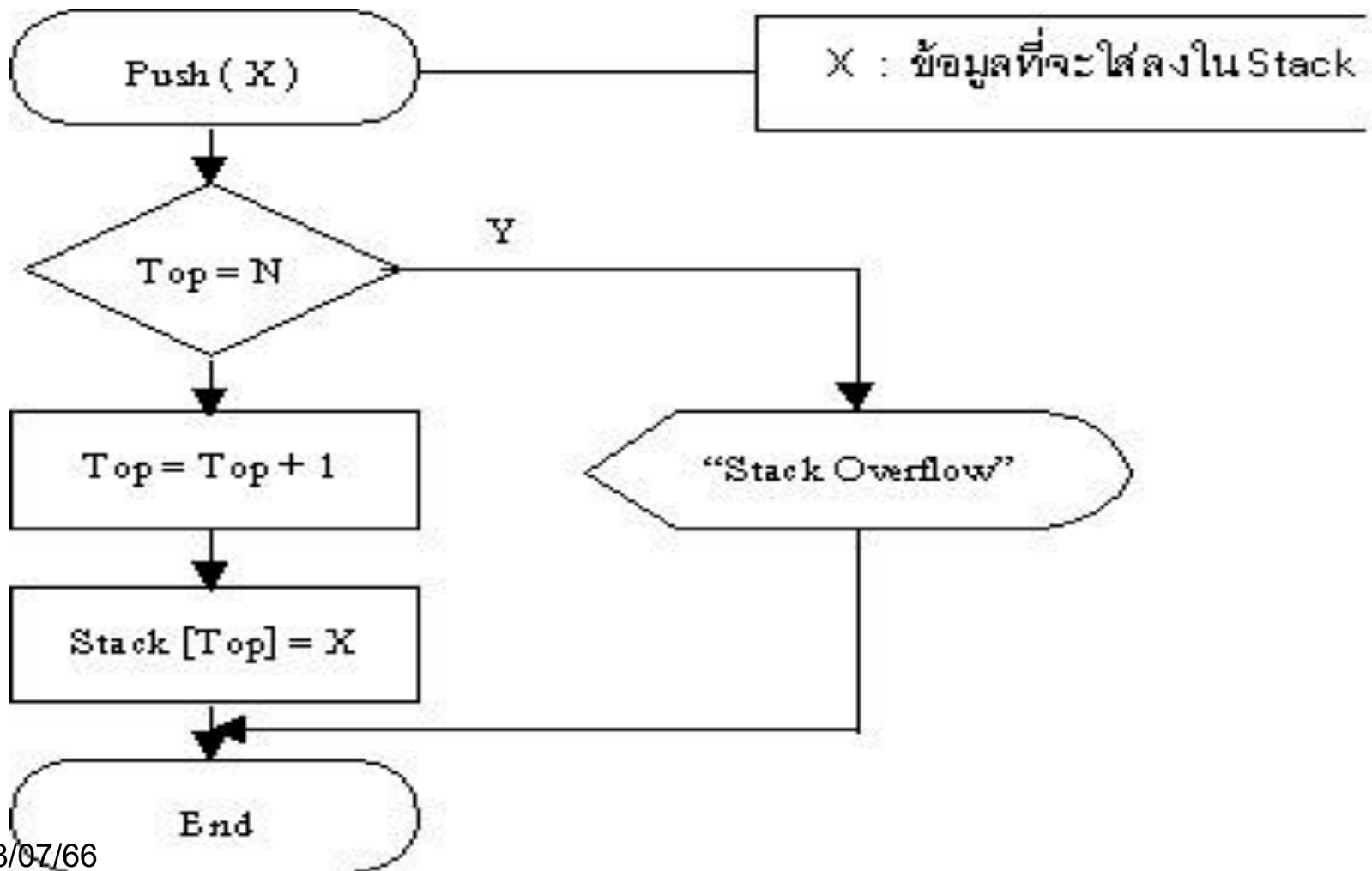
กองซ้อน เป็นโครงสร้างข้อมูลที่ใช้สำหรับ

- แทรกวัตถุลงบนกอง (ถ้าไม่ล้น) **push**
- ลบวัตถุออกจากกอง (ถ้าไม่ว่าง) **pop**

โดยจะต้องกระทำที่ด้านบนสุดของกองซ้อนเสมอ

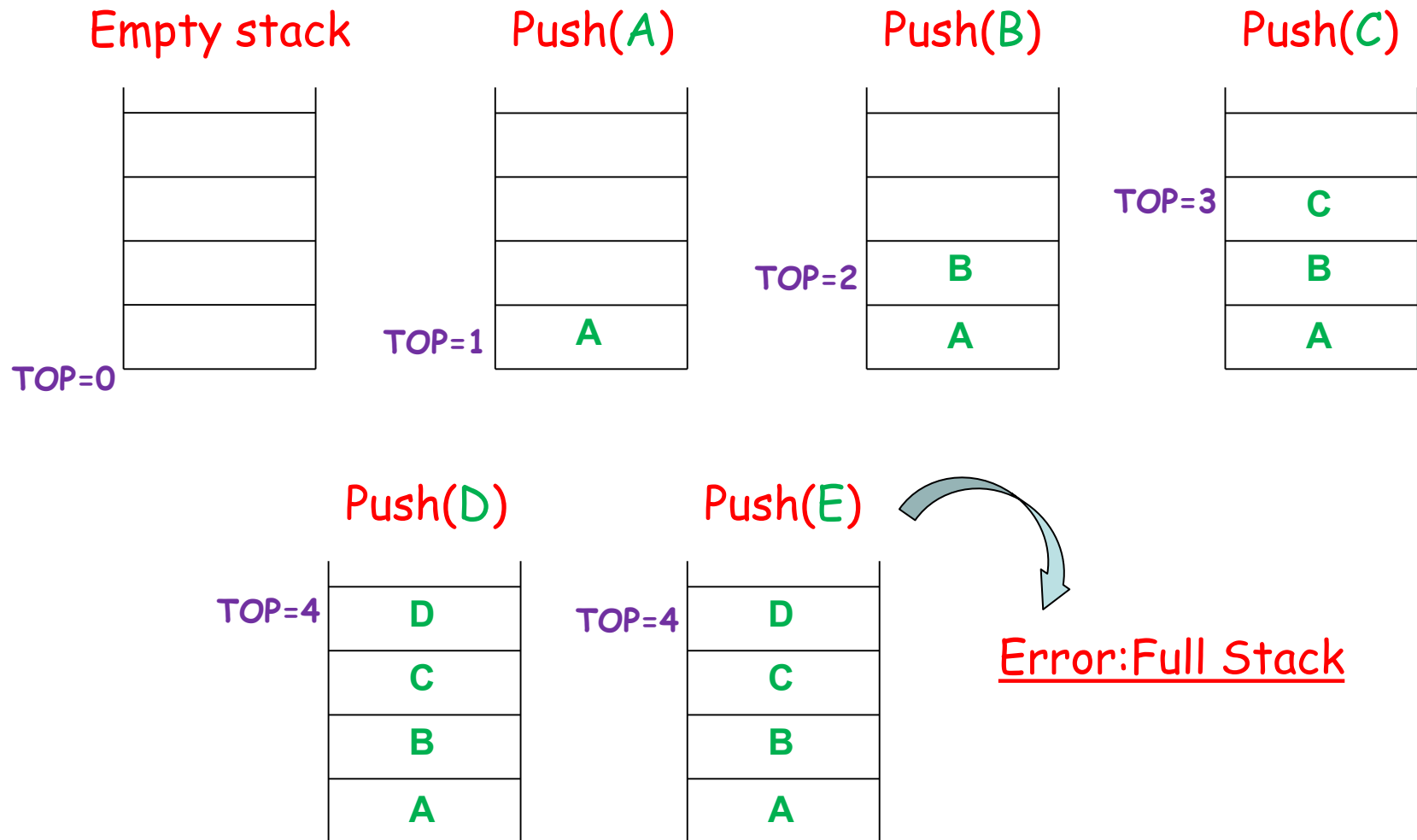


Flowchart Push()

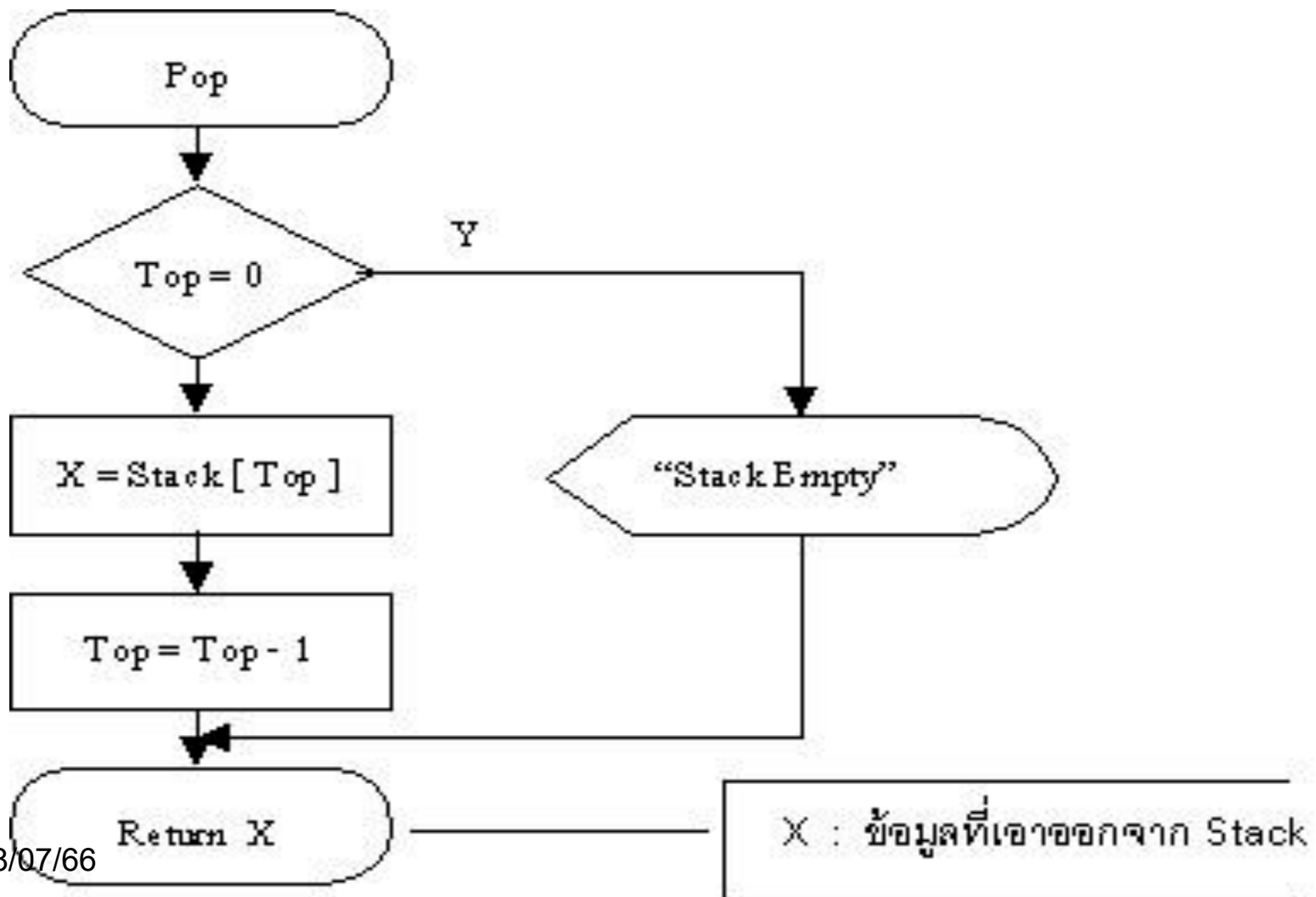


Stack

ตัวอย่าง PUSH โดยสมมติให้ ขนาดของ stack = 4



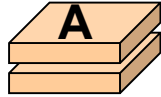
Flowchart Pop()



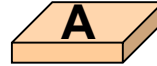
Push Q onto empty stack



Push A onto stack



Pop a box from stack



Pop a box from stack

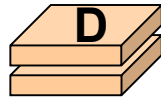
empty



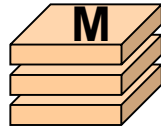
Push R onto stack



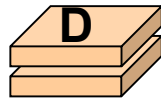
Push D onto stack



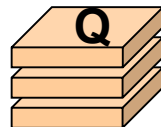
Push M onto stack



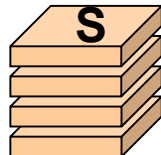
Pop a box from stack



Push Q onto stack



Push S onto stack



Push

Pop

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 1

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

(Stack1.c)

```
#define MAX 10
```

```
int stackdata[MAX];
```

```
int top = -1; stack Empty เพราะ Arr เริ่มต้น ที่ 0
```

```
int Push(int what) {
```

```
    if (top < MAX-1) {
```

```
        top++;
```

```
        stackdata[top] = what;
```

```
        return 1;
```

```
    }
```

```
    return -1;
```

```
}
```


ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 1 (ต่อ)

```
int Pop() {  
    int r;  
  
    if (top>-1) {  
        r=stackdata[top];  
        stackdata[top]=0;  
        top--;  
        return r;  
    }  
    return -1;  
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 1 (ต่อ)

```
void main() {  
    Push(7);  
    Push(12);  
    Push(23);  
    Push(2);  
    Push(33);  
    Push(10);  
    printf("Pop1=%d\n",Pop());  
    printf("Pop2=%d\n",Pop());  
    printf("Pop3=%d\n",Pop());  
    printf("Pop4=%d\n",Pop());  
    getch();  
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 2 (ต่อ)

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

(Stack2.c)

```
#define MAX 10
```

```
int stackdata[MAX];
int top = -1;
```

```
int Push(int what) {
    if (top < MAX-1) {
        top++;
        stackdata[top] = what;
        return 1;
    }
    return -1;
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 2 (ต่อ)

```
int Pop() {  
    int r;  
  
    if (top>-1) {  
        r=stackdata[top];  
        stackdata[top]=0;  
        top--;  
        return r;  
    }  
    return -1;  
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 2 (ต่อ)

```
void main() {  
    Push(7);  
    Push(12);  
    Push(23);  
    Push(2);  
    Push(33);  
    Push(10);  
    for (int i=0;i<8;i++) {  
        printf("Pop=%d\n",Pop());  
    }  
    getch();  
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 3 (ต่อ)

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

(Stack3.c)

```
#define MAX 10
```

```
int stackdata[MAX];
int top = -1;
```

```
int Push(int what) {
    if (top < MAX-1) {
        top++;
        stackdata[top] = what;
        return 1;
    }
    return -1;
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 3 (ต่อ)

```
int Pop() {  
    int r;  
    if (top>-1) {  
        r=stackdata[top];  
        stackdata[top]=0;  
        top--;  
        return r;  
    }  
    return -1;  
}
```

ตัวอย่างโปรแกรมแสดงการทำงานของ Stack ตัวอย่างที่ 3 (ต่อ)

```
void main() {  
    Push(7);  
    Push(12);  
    Push(23);  
    Push(2);  
    Push(33);  
    Push(10);  
    for (int i=0;i<50;i++) {  
        int result=Pop();  
  
        if (result==-1) {  
            printf("Stack is empty\n");  
            getch();  
            break;  
        }  
        printf("Pop=%d\n", result);  
        getch();  
    }  
}
```



```

1 #define MAX_STACK_SIZE 100
2 #define TRUE 1
3 #define FALSE 0
4
5 typedef struct int_stack_type {
6     int item[MAX_STACK_SIZE];
7     int top;
8 } Stack;
9
10 void initStack(Stack *s){
11     s->top = -1;
12 }
13
14 int pushStack(Stack *s, int x){
15     if (s->top >= MAX_STACK_SIZE -1)
16         return FALSE;
17     else {
18         s->top++;
19         s->item[s->top] = x;
20         return TRUE;
21     }
22 }
23
24 int popStack(Stack *s, int *x) {
25     if (s->top < 0)
26         return FALSE;
27     else {
28         *x = s->item[s->top];
29         s->top--;
30         return TRUE;
31     }
32 }
33
34 void printStack(Stack s){
35     int i;
36
37     printf("Stack:");
38     for (i=0; i<=s.top; i++)
39         printf("%d", s.item[i]);
40     printf("\n");
41 }

```

ตัวอย่างโปรแกรม stack4.c

```

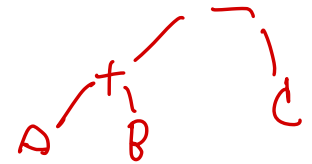
43 void main(){
44     Stack s;
45     int choice;
46     int cont = TRUE;
47     int x;
48
49     initStack(&s);
50     while (cont == TRUE){
51         printf("Please select [1:push 2:pop 3:print 0:exit]");
52         scanf("%d", &choice);
53         switch(choice){
54             case 1:
55                 printf("Please enter a number to be pushed: ");
56                 scanf("%d", &x);
57                 if (!pushStack(&s,x))
58                     printf(" Error pushing into the stack\n");
59                 break;
60             case 2:
61                 if (popStack(&s,&x))
62                     printf("The number %d is popped off\n",x);
63                 else
64                     printf(" Error popping from the stack\n");
65                 break;
66             case 3:
67                 printStack(s);
68                 break;
69             case 0:
70                 cont = FALSE;
71                 break;
72         }
73     }
74 }

```

การใช้ สแตค เพื่อแปลงรูปนิพจน์ทางคณิตศาสตร์

รูปแบบนิพจน์ทางคณิตศาสตร์

- นิพจน์ **Infix** คือ นิพจน์ที่เครื่องหมายดำเนินการ (Operator) อยู่ระหว่างตัวดำเนินการ (Operands) เช่น $A+B-C$



- นิพจน์ **Prefix** คือ นิพจน์ที่เครื่องหมายดำเนินการ (Operator) อยู่หน้าตัวดำเนินการ (Operands) เช่น $-+ABC$

- นิพจน์ **Postfix** คือ นิพจน์ที่เครื่องหมายดำเนินการ (Operator) อยู่หลังตัวดำเนินการ (Operands) เช่น $AB+C-$

$$A+(B/(C-D))*E$$

ตัวอย่างนิพจน์คณิตศาสตร์ในรูปแบบต่าง ๆ

นิพจน์ Infix

- $A+B-C$
- $A+B*C-D/E$
- $A*B+C-D/E$

นิพจน์ Postfix

$AB+C-$

$ABC*+DE/-$

$AB*C+DE/-$

นิพจน์ Prefix

$- +ABC$

$- +A*BC/DE$

$- +*ABC/DE$

$$A+(B/(C-D))*E$$

การแปลงนิพจน์ Infix ให้เป็น Postfix

- ข้อเสียของนิพจน์ infix ที่ทำให้คอมพิวเตอร์ยุ่งยาก
- ลำดับความสำคัญของโอเปอเรเตอร์ (**Precedence**) มีความต่างกัน เช่น
 - เครื่องหมายยกกำลัง มีความสำคัญมากกว่าเครื่องหมายคูณ และหาร
 - เครื่องหมายคูณและหารมีความสำคัญมากกว่าเครื่องหมายบวกและลบ

การแปลงนิพจน์ Infix ให้เป็น Postfix

- เมื่อการประมวลนิพจน์ infix เป็นไปด้วยความยากที่การคำนวณไม่เป็นไปตามลำดับของเครื่องหมายโอเปอเรเตอร์(operator) ที่มีก่อนหลัง คอมไพเลอร์จึงแปลงนิพจน์ infix ให้เป็น postfix เสียก่อน
- นิพจน์ Postfix ก็คือนิพจน์ที่มีโอเปอเรเตอร์อยู่หลังโอเปอเรนด์(operand)ทั้งสองของมัน เช่น

$AB+$	หมายถึง	$A+B$
$AB-$	หมายถึง	$A-B$
$AB*$	หมายถึง	$A*B$

การแปลงนิพจน์ Infix ให้เป็น Postfix

- Operator คือเครื่องหมายกระทำ $+$ $-$ $*$ $/$ $^$
- Operand คือตัวแปรต่าง ๆ A, B, C, D, E เช่น $A+B*C, (A*B)-C$

ลำดับความสำคัญ Operator

1. เครื่องหมายยกกำลัง $^$
2. เครื่องหมายคูณกับหาร $*, /$
3. เครื่องหมายบวกกับลบ $+, -$
4. เครื่องหมายวงเล็บ $()$ กระทำก่อนเช่น $A+(B*C)$
5. เครื่องหมายระดับเดียวกันไม่มีวงเล็บให้ทำจากซ้ายไปขวา
เช่น $A+B+C$

ตัวอย่างนิพจน์ทางคณิตศาสตร์และลำดับการคำนวณ

$$\begin{array}{c} A - B * C \\ \quad \underbrace{\hspace{1.5cm}} \\ \quad \quad 1 \\ \underbrace{\hspace{2.5cm}} \\ \quad 2 \end{array}$$

$$\begin{array}{c} (A + B) * C \\ \underbrace{\hspace{1.5cm}} \\ \quad 1 \\ \underbrace{\hspace{2.5cm}} \\ \quad \quad 2 \end{array}$$

$$\begin{array}{c} (A - B) * (C / D) \\ \underbrace{\hspace{1.5cm}} \quad \underbrace{\hspace{1.5cm}} \\ \quad 1 \quad \quad 2 \\ \underbrace{\hspace{3.5cm}} \\ \quad \quad 3 \end{array}$$

$$\begin{array}{c} A * B / (C / D) \\ \quad \quad \underbrace{\hspace{1.5cm}} \\ \quad \quad \quad 1 \\ \quad \quad \underbrace{\hspace{1.5cm}} \\ \quad \quad \quad 2 \\ \underbrace{\hspace{2.5cm}} \\ \quad 3 \end{array}$$

อัลกอริทึมการแปลงนิพจน์ Infix เป็น นิพจน์ Postfix

- เราสามารถแปลงนิพจน์ Infix ให้เป็น Postfix ได้โดยอาศัย **stack** ที่มีคุณสมบัติการเข้าหลังออกก่อนหรือ LIFO โดยมีอัลกอริทึมในการแปลงนิพจน์ ดังนี้
 1. ถ้าข้อมูลเข้า (input) เป็นตัวถูกดำเนินการ (operand) ให้นำออกไปเป็นผลลัพธ์ (output)
 2. ถ้าข้อมูลเข้าเป็นตัวดำเนินการ (operator) ให้ดำเนินการดังนี้
 - 2.1 ถ้า stack ว่าง
ให้ push operator ลงใน stack

อัลกอริทึมการแปลงนิพจน์ Infix เป็น นิพจน์ Postfix

2.2 ถ้า stack ไม่ว่าง

ให้เปรียบเทียบ operator ที่เข้ามากับ operator ที่อยู่ในตำแหน่ง TOP ของ stack

2.2.1 ถ้า operator ที่เข้ามามีความสำคัญ $>$ operator ที่ตำแหน่ง TOP ของ stack
ให้ push ลง stack

2.2.2 ถ้า operator ที่เข้ามามีความสำคัญ \leq operator ที่ตำแหน่ง TOP ของ stack
ให้ pop stack ออกไปเป็นผลลัพธ์

แล้วทำการเปรียบเทียบ operator ที่เข้ามากับ operator ที่ตำแหน่ง TOP ต่อไป

จนกว่า operator ที่เข้ามามีความสำคัญ $>$ operator ที่ตำแหน่ง TOP ของ stack

แล้วจึง push operator ที่เข้ามานั้นลง stack

อัลกอริทึมการแปลงนิพจน์ Infix เป็น นิพจน์ Postfix

3. ถ้าข้อมูลเข้าเป็นวงเล็บเปิด

ให้ push ลง stack

4. ถ้าข้อมูลเข้าเป็นวงเล็บปิด

ให้ pop ข้อมูลออกจาก stack ไปเป็นผลลัพธ์

จนกว่าจะถึงวงเล็บ เปิด

จากนั้นตัดวงเล็บเปิดและปิดทิ้งไป


5. ถ้าข้อมูลเข้าหมด

ให้ pop ข้อมูลออกจาก stack ไปเป็นผลลัพธ์

จนกว่า stack จะว่าง

ตัวอย่างการแปลงนิพจน์ Infix เป็นนิพจน์ Postfix

- นิพจน์ $A + B * C$

นิพจน์ Infix ข้อมูลเข้า (Input)	Stack เก็บตัวดำเนินการ	นิพจน์ Postfix ข้อมูลออก (Output)
A	stay empty 	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
18/07/66		ABC*+ 36

นิพจน์ $(A * B) + (C / D)$

นิพจน์ Infix ข้อมูลเข้า (Input)	Stack เก็บตัวดำเนินการ	นิพจน์ Postfix ข้อมูลออก (Output)
((
A	(A
*	(*	A
B	(*	AB
)		AB*
+	+	AB*
(+(AB*
C	+(AB*C
/	+(/	AB*C
D	+(/	AB*CD
)	+	AB*CD/
		AB*CD/+

นิพจน์ $A / B + (C - D)$

นิพจน์ Infix ข้อมูลเข้า (Input)	Stack เก็บตัวดำเนินการ	นิพจน์ Postfix ข้อมูลออก (Output)
A		A
/	/	A
B	/	AB
+	+	AB/
(+(AB/
C	+(AB/C
-	+(-	AB/C
D	+(-	AB/CD
)	+	AB/CD-
		AB/CD-+

นิพจน์ $(A + B) * (C ^ D - E) * F$

นิพจน์ Infix ข้อมูลเข้า (Input)	Stack เก็บตัวดำเนินการ	นิพจน์ Postfix ข้อมูลออก (Output)
((
A	(A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
(*(AB+
C	*(AB+C
^	*(^	AB+C
D	*(^	AB+CD
-	*(-	AB+CD^
E	*(-	AB+CD^E
)	*	AB+CD^E-
*	*	AB+CD^E-*
F	*	AB+CD^E-*F
		AB+CD^E-*F*

ขั้นตอนการคำนวณจากนิพจน์ Postfix

ในการคำนวณค่า Postfix ที่แปลงมาแล้ว

Compiler จะทำการคำนวณโดยใช้โครงสร้าง **stack** ช่วยอีกเช่นกัน
ขั้นตอนในการคำนวณ

1. อ่านตัวอักษรในนิพจน์ Postfix จากซ้ายไปขวาทีละตัวอักษร
2. ถ้าเป็นตัวถูกดำเนินการ(ตัวเลข)

ให้ทำการ push ตัวถูกดำเนินการ(Operand)นั้นลงใน stack
แล้วกลับไปอ่านอักษรตัวใหม่เข้ามา

3. ถ้าเป็นตัวดำเนินการ(Operator/เครื่องหมาย)

ให้ทำการ pop ค่าจาก stack 2 ค่า

โดยตัวแรกเป็นตัวถูกดำเนินการตัวที่ 2 และตัวที่ 1 ตามลำดับ

ขั้นตอนการคำนวณจากนิพจน์ Postfix

4. ทำการคำนวณ ตัวถูกดำเนินการ(Operand)ตัวที่ 1
ด้วยตัวถูกดำเนินการ(Operand)ตัวที่ 2
โดยใช้ตัวดำเนินการ(Operator)ในข้อ 3
5. ทำการ push ผลลัพธ์ที่ได้จากการคำนวณในข้อ 4 ลง stack
6. ถ้าตัวอักษรในนิพจน์ Postfix ยังอ่านไม่หมด
ให้กลับไปทำข้อ 1 ใหม่

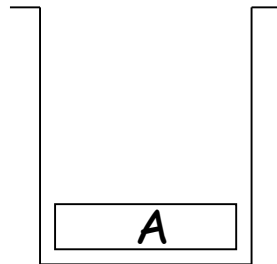
ขั้นตอนการคำนวณจากนิพจน์ Postfix

ตัวอย่าง ขั้นตอนการคำนวณจากนิพจน์ Postfix $ABC+D-*E/$

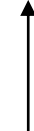
1. $ABC+D-*E/$



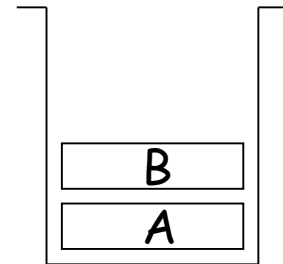
Push A



2. $ABC+D-*E/$



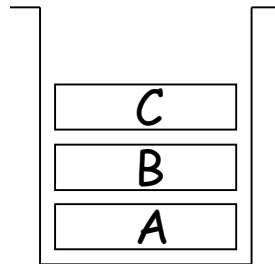
Push B



ขั้นตอนการคำนวณจากนิพจน์ Postfix

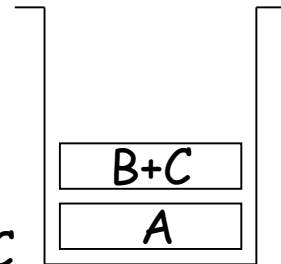
3. $ABC+D-*E/$

Push C



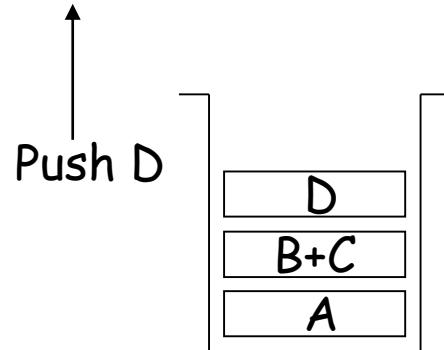
4. $ABC+D-*E/$

1. Pop C
2. Pop B
3. $B+C$
4. Push $B+C$

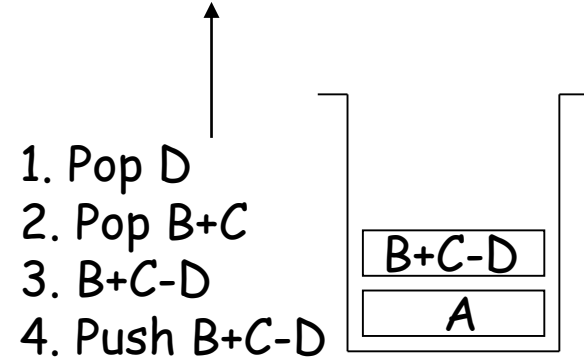


ขั้นตอนการคำนวณจากนิพจน์ Postfix

5. $ABC+D-*E/$



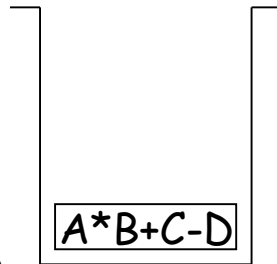
6. $ABC+D-*E/$



ขั้นตอนการคำนวณจากนิพจน์ Postfix

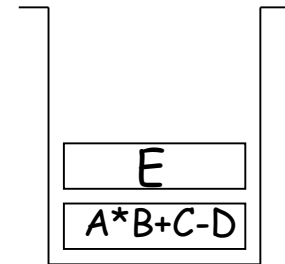
7. $ABC+D-*E/$

1. Pop $B+C-D$
2. Pop A
3. $A*B+C-D$
4. Push $A*B+C-D$



8. $ABC+D-*E/$

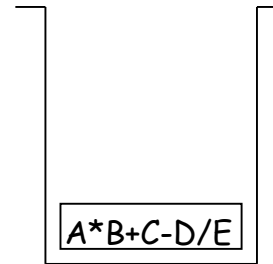
Push E



ขั้นตอนการคำนวณจากนิพจน์ Postfix

9. $ABC+D-*E/$

1. Pop E
2. Pop $A*B+C-D$
3. $A*B+C-D/E$
4. Push $A*B+C-D/E$



10. ค่าสุดท้ายที่อยู่ในสแตกคือคำตอบที่ต้องการ

Postfix Expression

ใช้เพื่อกำหนดหาผลลัพธ์จาก postfix

1. ถ้าเป็น Operand ให้ push ลง stack
2. ถ้าเป็น Operator ให้ pop operand มา 2 ตัว
โดยให้ตัวที่ pop มาทีหลังเป็นตัวตั้ง
คำนวณแล้ว push ลง stack
3. ตัวที่อยู่ใน stack จะเป็นผลลัพธ์

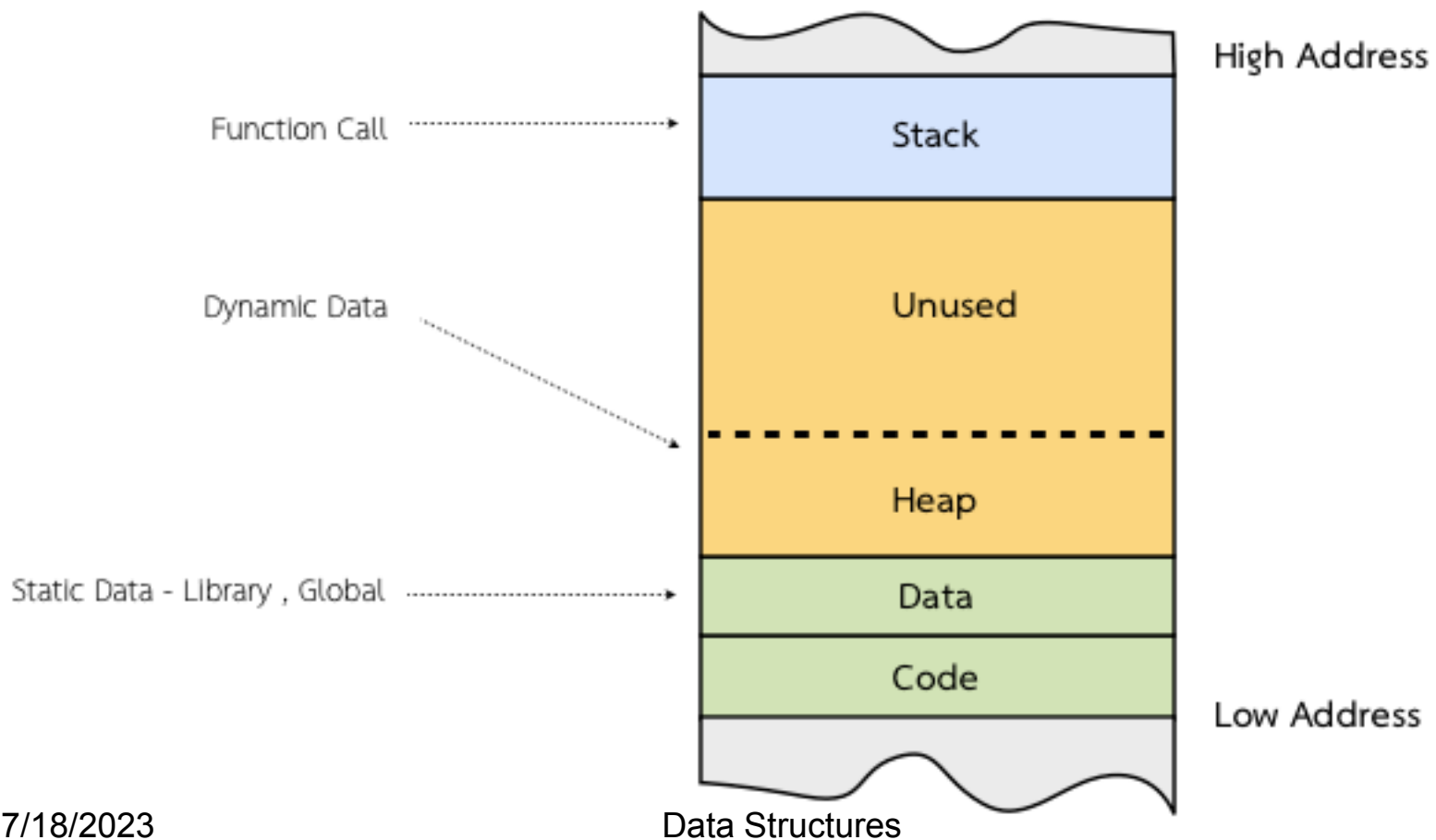
Postfix : 12 7 3 - / 2 1 5 + * *

Input	Action	Stack
12	NON	12
7	NON	12 7
3	NON	12 7 3
-	$7-3 = 4$ ← action ใน stack นี้	12 4
/	$12/4 = 3$	3
2	NON	3 2
1	NON	3 2 1
5	NON	3 2 1 5
+	$1+5 = 6$	3 2 6
*	$2*6 = 12$	3 12
*	$3 * 12 = 36$	36

Function Call

- การเรียกใช้ Function หรือ Procedure
หรือโปรแกรมย่อยในภาษาที่ไม่มีการ Recursive
- เมื่อมีการเรียกใช้ Function
จะทำการ Push Function ไปยัง Stack
- เมื่อมีการ Return หรือจบการทำงานของ Function
จะต้อง Pop Function ออกจาก Stack

การจัดสรร memory



การเรียกใช้โปรแกรมย่อย

- การเรียกโปรแกรมย่อยมีความแตกต่างกับการกระโดดทั่วไป
- ภายหลังที่โปรแกรมย่อยทำงานเสร็จ
หน่วยประมวลผลจะต้องสามารถกระโดดกลับมาทำงานใน
โปรแกรมหลักต่อไปได้
- ดังนั้นการเรียกใช้โปรแกรมย่อยนั้น
ต้องเก็บตำแหน่งของคำสั่งที่ทำงานอยู่เดิมด้วย
และเมื่อจบโปรแกรมย่อยโปรแกรมจะต้องกระโดด
กลับมาทำงานที่เดิม โดยใช้ข้อมูลที่เก็บไว้

PROGRAM MAIN

.....

CALL Sub¹

PRINT Q

....

END MAIN

PROCEDURE Sub¹

....

CALL Sub²

A:=A+B

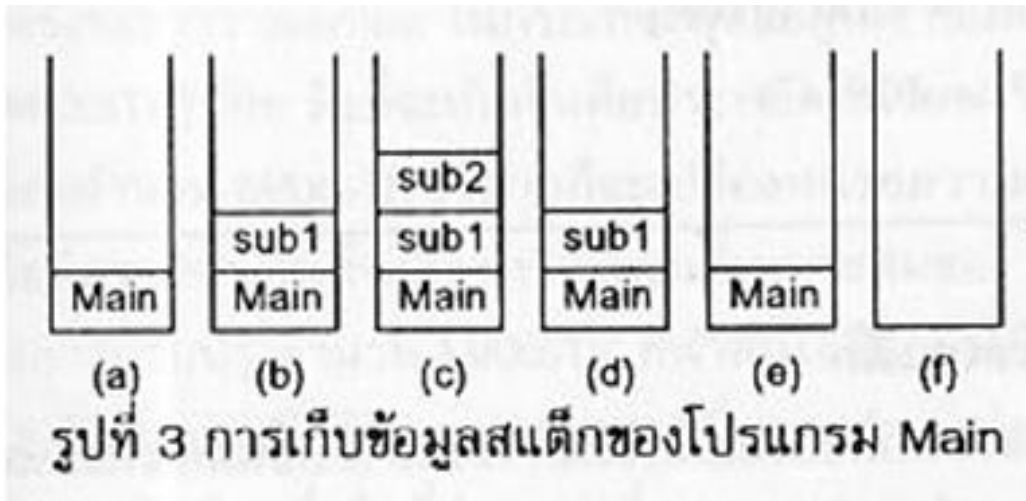
...

END Sub¹

PROCEDURE Sub²

...

END Sub²



Balancing Symbols

Example :

$$\{x+(y-[a+b])^*c-[(d+e)]\}/(h-(j-(k-[l-n])))$$

การตรวจสอบอักขระสมดุล (Balancing Symbol)

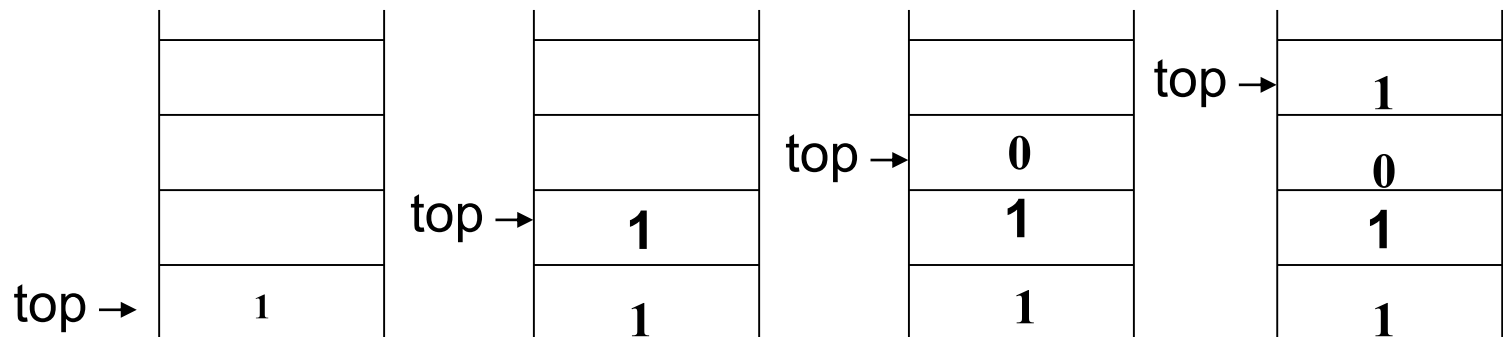
- คอมไพเลอร์ได้นำแนวคิด `Stack` มาประยุกต์ โดยมีวิธีการดังนี้
- 1.ให้อ่านอักขระทีละตัว
 - - ถ้าอักขระเป็นอักขระเปิด เช่น `{, [,` เป็นต้น ให้ **PUSH** ลง `stack`
 - - ถ้าอักขระเป็นอักขระปิด เช่น `},],` เป็นต้น ให้ตรวจสอบว่าอักขระบน `TOP` ของ `stack` เป็นอักขระเปิดที่คู่กันหรือไม่
 - - ถ้าใช่ ให้ **POP** อักขระนั้นออกจาก `stack`
 - - ถ้าไม่ใช่ ให้แสดงผล **error**
- 2. เมื่ออ่านอักขระหมดแล้ว แต่ `stack` ไม่เป็น `stack` ว่างให้แสดงผล **error**

การใช้ สแตก เพื่อแปลงเลขฐาน10 เป็นเลขฐาน 2

Algorithm

1. loop (number > 0)
 - 1.1 $b = \text{number modulo } 2$
 - 1.2 push (stack,b)
 - 1.3 $\text{number} = \text{number} / 2$
2. loop (not empty(stack))
 - 2.1 pop(stack)

ตัวอย่าง: 11 --> 1011



ตัวอย่างโปรแกรมภาษา C++

```
1  // stack::push/pop
2  #include <iostream>           // std::cout
3  #include <stack>              // std::stack
4  using namespace std;
5
6  int main ()
7  {
8      stack<int> mystack;
9
10     for (int i=0; i<5; ++i) mystack.push(i);
11
12     cout << "Popping out elements...";
13     while (!mystack.empty())
14     {
15         cout << ' ' << mystack.top();
16         mystack.pop();
17     }
18     cout << '\n';
19
20     return 0;
21 }
```