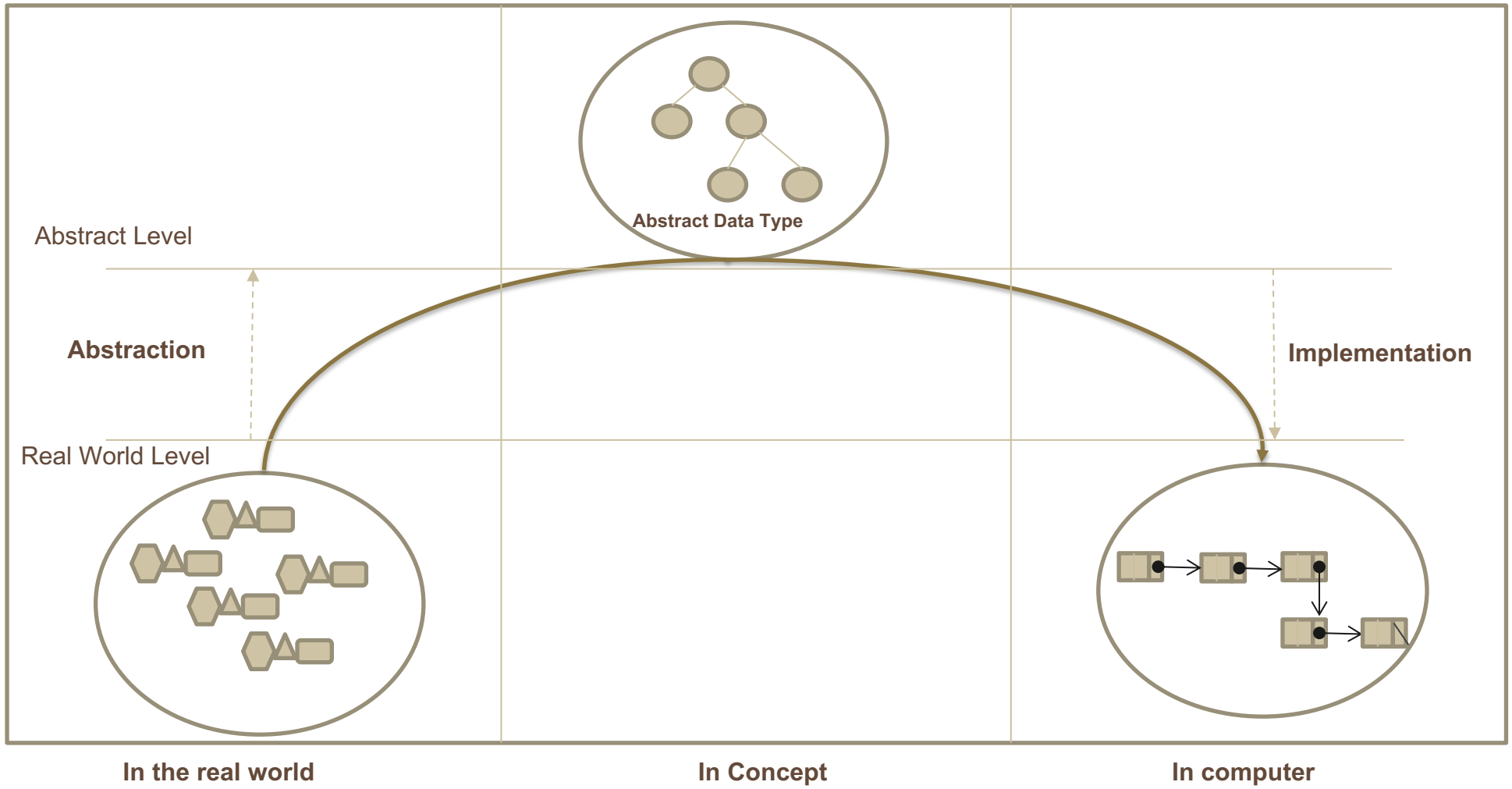
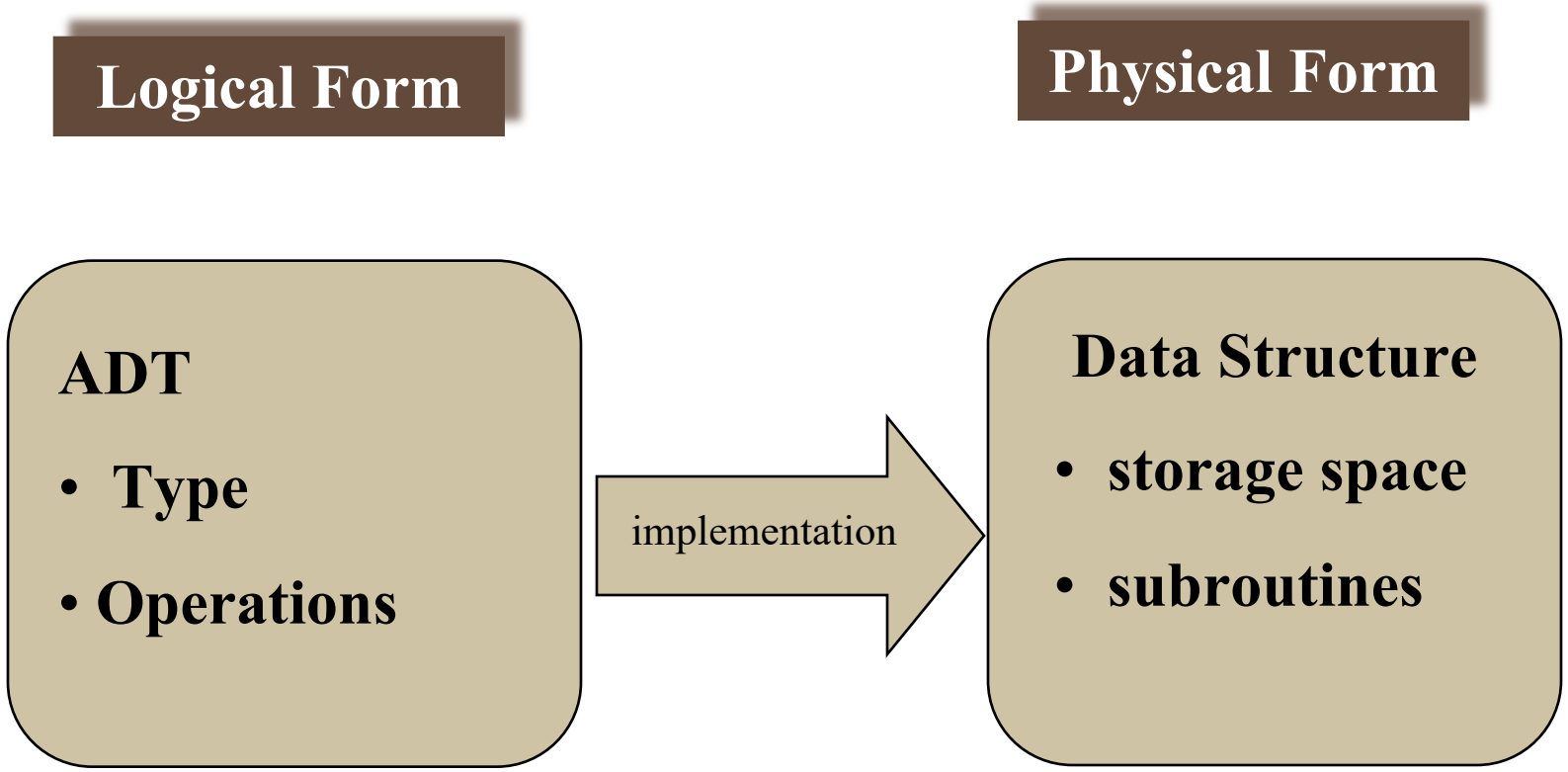
การอบรมคอมพิวเตอร์โอลิมปิกค่าย 2

Linear Data Structures

Review: Abstraction VS Implementation



ความสัมพันธ์ของ ADTและโครงสร้างข้อมูล



Linear Data Structures

- List
- Stack
- Queue

Lists

Linear Data Structures

ลิสต์

- สมมติเรามีข้อมูลเป็นเลขจำนวนเต็ม ชุดหนึ่ง คือ [12 23 45 60] ซึ่งเรียงลำดับแล้ว
- ถ้าต้องการเพิ่มข้อมูล 65 เข้าไป จะไปเพิ่มที่ตำแหน่งใด โดยให้เรียงตามลำดับเหมือนเดิม? และเพิ่มอย่างไร?
- ถ้าต้องการเพิ่มข้อมูล 25 เข้าไป จะไปเพิ่มที่ตำแหน่งใด โดยให้เรียงตามลำดับเหมือนเดิม? และเพิ่มอย่างไร?
- ถ้าต้องการลบข้อมูล 23 ออกจากลิสต์ จะต้องทำอย่างไร ?

นิยามลิสต์ (Lists)

- **ลิสต์** (Lists) เป็นโครงสร้างข้อมูลนามธรรมที่นำข้อมูลมาเรียงต่อกันอย่างเป็นลำดับ
- **ลิสต์**เป็นโครงสร้างข้อมูลแบบพลวัต (dynamic data structure) กล่าวคือ จำนวนข้อมูลในลิสต์จะเปลี่ยนแปลงตลอดเวลาเมื่อมีการเพิ่มหรือลบข้อมูล
- **รูปแบบทั่วไปของลิสต์** :

  $a_1, a_2, a_3, a_4, ..., a_n$

- ขนาดของลิสต์คือ ***n***
- ลิสต์อาจประกอบด้วยข้อมูลแค่ตัวเดียว หรือไม่มีข้อมูลเลย
- ลิสต์ว่าง (ขนาดของลิสต์เป็น 0) ในบางครั้งเราเรียกลิสต์ว่างว่าเป็น *null list*
- หน่วยของข้อมูลแต่ละตัวในลิสต์ว่าโหนด (*node*)

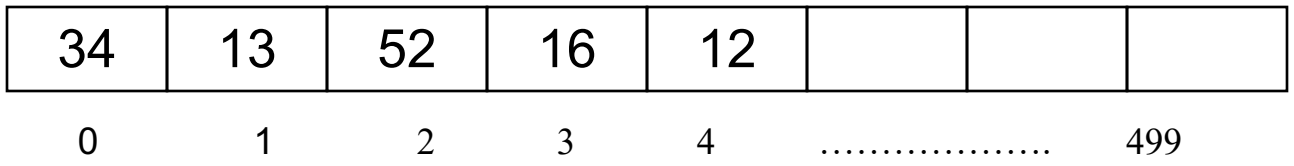
Operations ทั่วไปของลิสต์

print()	แสดงข้อมูลทั้งหมดในลิสต์
empty()	ทำให้ลิสต์กลายเป็นลิสต์ว่างหรือลบข้อมูลทั้งหมดในลิสต์
find(info)	ค้นหาข้อมูล info ในลิสต์และแสดงตำแหน่งที่พบข้อมูลในลิสต์
insert(info, position)	แทรกข้อมูล info ในลิสต์ที่ตำแหน่ง position (ตำแหน่งแรกคือ 0)
delete (position)	ลบข้อมูลที่ตำแหน่ง position ออกจากลิสต์

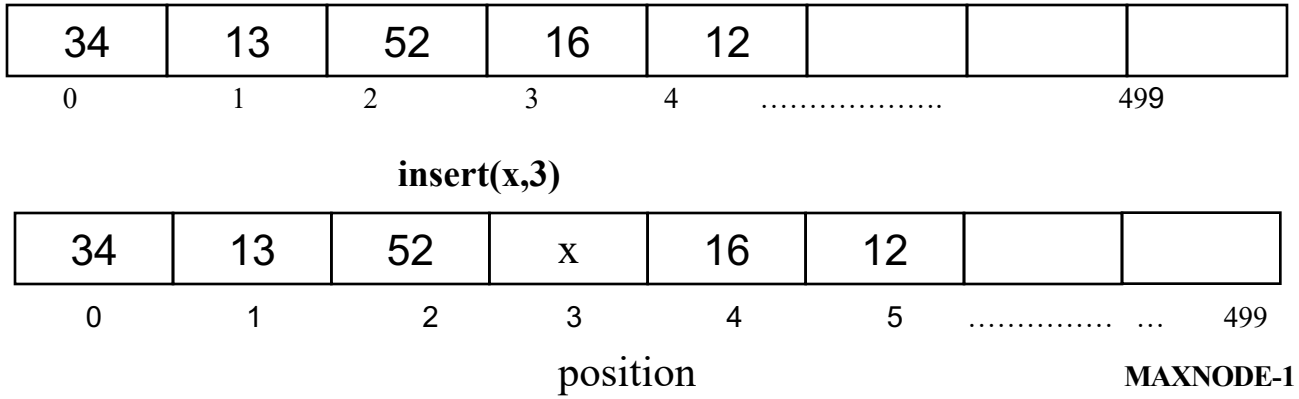
การสร้างลิสต์ด้วยอาเรย์

- สมมติว่าเราจะสร้างลิสต์ของตัวเลขและเราประมาณว่าจำนวนข้อมูลในลิสต์จะมีไม่เกิน 500

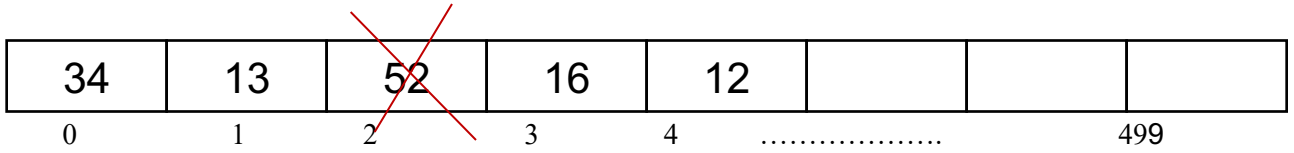
```
#define MAXNODE 500
int node[MAXNODE];
int total;
```



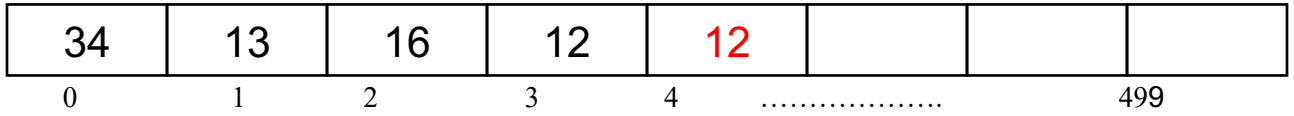
```
/* insert info after the position position */
void insert(int info, int position)  {
    int i;
    if (total !=  MAXNODE) {
        for (i = total-1; i >= position; i-- )
            node[i+1] = node[i];
        node[position] = info;
        total++;
    }
    else
        printf  ("The list is full. Can not insert");
}
```



```
/* delete the position th item */
void delete(int position) {
    int i;
    if (position < total && position >= 0)
        for(i = position; i < total; i++ )
            node[i] = node[i+1];
    total--;
}
```



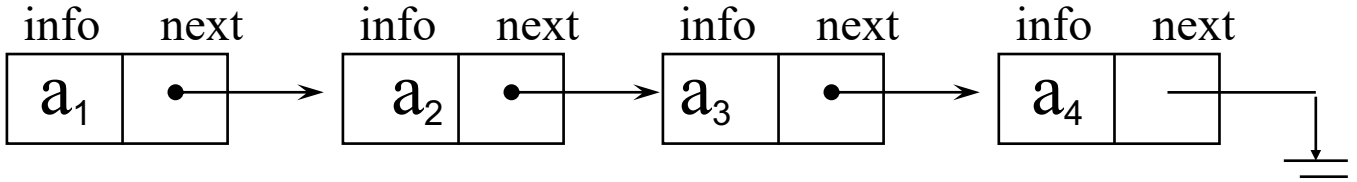
delete(2)



Linked List

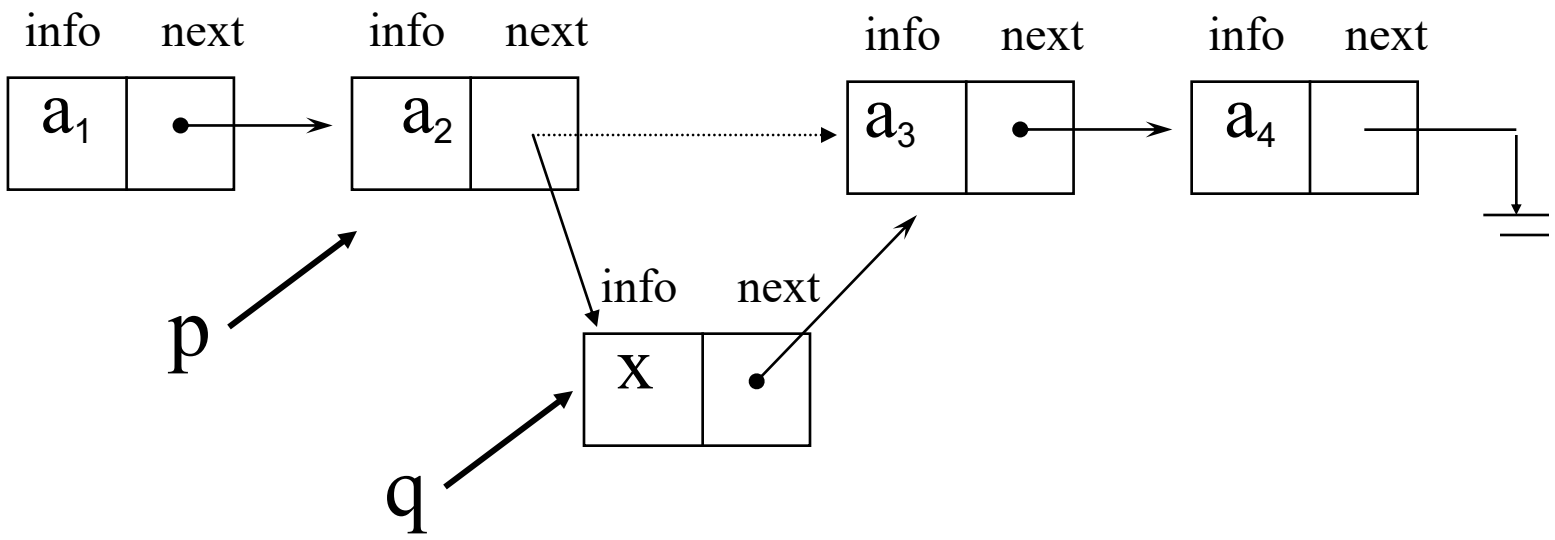
Lists

ลิสต์เชื่อมโยง



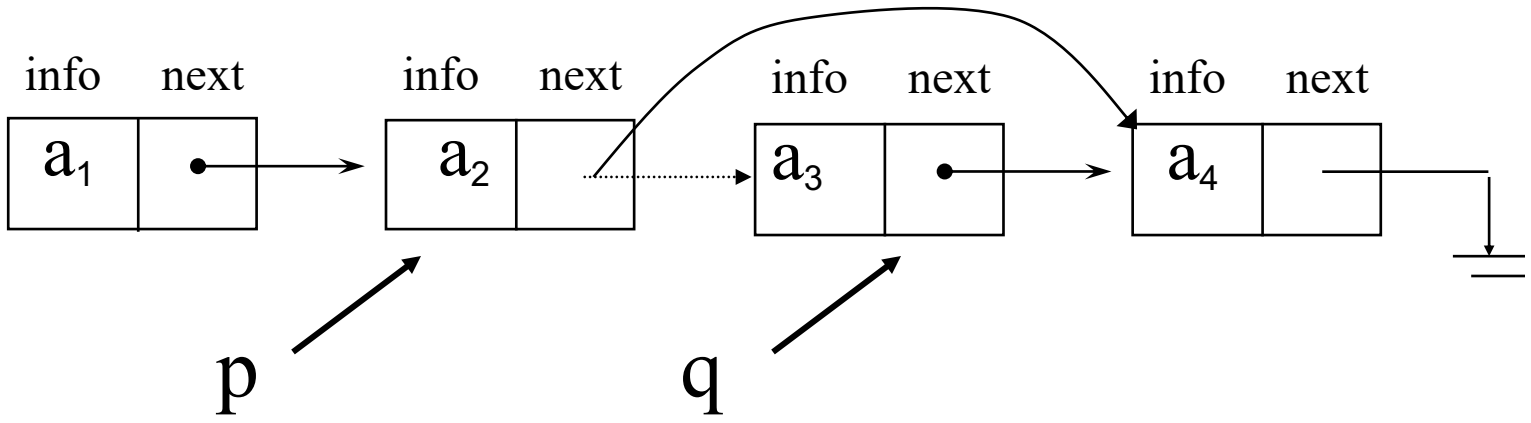
- ลิงค์ลิสต์ประกอบไปด้วยโครงสร้างโหนด ที่เชื่อมโยงกัน โดยที่แต่ละโหนดไม่จำเป็นต้องอยู่ติดกันในหน่วยความจำ
- โหนดจะประกอบไปด้วยส่วนที่เก็บข้อมูล info และส่วนที่เป็นตัวชี้ (Pointer) next ที่จะชี้ไปยังตำแหน่งของโหนดที่เก็บข้อมูลตัวถัดไป
- โหนดของข้อมูลตัวสุดท้ายค่าของ next จะเป็นค่าว่าง (null) หรือค่า 0 ในภาษา C

การแทรกโหนดในลิสต์



แทรกโนด q หลังโหนด p

การลบโหนดจากลิสต์



ลบโนด q ซึ่งอยู่หลังโหนด p จากลิสต์

Linked List Implementation

การสร้างลิงค์ลิสต์ด้วยภาษา C

- เราสามารถจะสร้างลิงค์ลิสต์ด้วยภาษา C ได้ 2 แบบคือ
  - การสร้างด้วยอาเรย์
  - การสร้างด้วยตัวแปรพลวัตร (dynamic variable)

		info	next
	0	26	-1
	1	11	9
	2	5	10
list3 →	3	1	14
	4		
	5	13	1
	6		
list2 →	7	17	0
	8	14	12
	9	4	19
	10	37	18
	11		
	12	6	2
	13		
	14	18	5
	15	7	8
list1 →	16	3	15
	17		
	18	12	-1
	19	15	-1
	:	:	:

ภายในอะเรย์ประกอบด้วย
3 ลิงค์ลิสต์

กำหนด

int list1 = 16;

int list2 = 7;

int list3 = 3;

ข้อมูลของ list1 คือ 3, 7, 14, 6, 5, 37, 12
ข้อมูลของ list2 คือ 17, 26
ข้อมูลของ list3 คือ 1, 18, 13, 11, 4, 15

ข้อดีและข้อเสียของการสร้างด้วยอาเรย์

ข้อดี

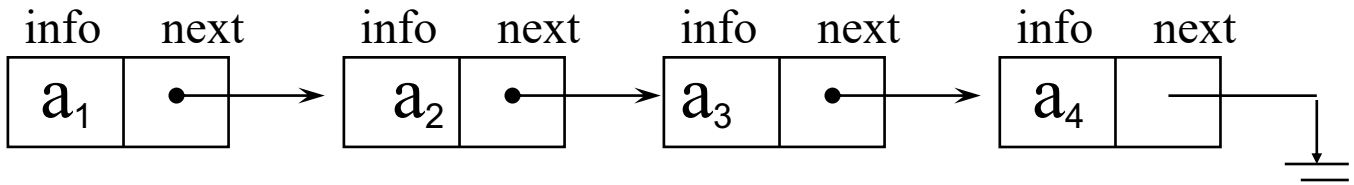
- สามารถทำได้ง่าย เพราะเกือบทุกภาษาโปรแกรมมีโครงสร้างอาเรย์อยู่แล้ว

ข้อเสีย

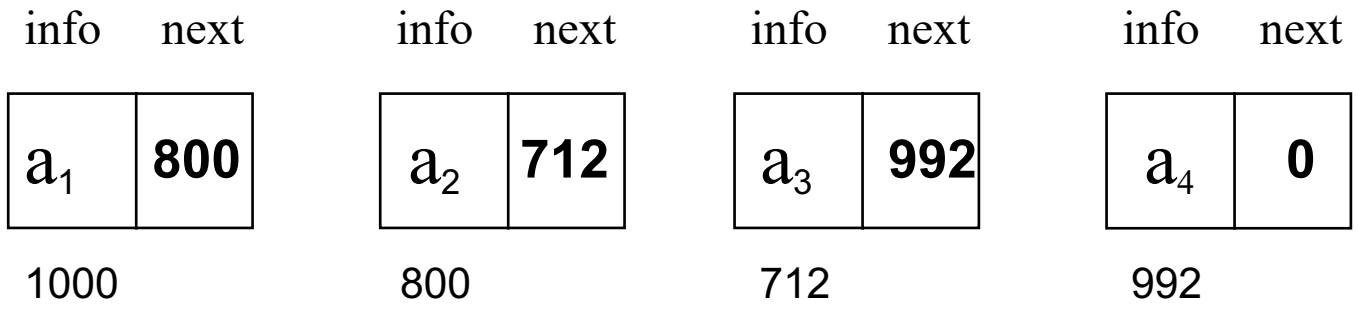
- ต้องกำหนดขนาดของอาเรย์ที่จะใช้ไว้ล่วงหน้า
- การแทรกข้อมูล ก็จะต้องวิ่งหาตำแหน่งที่ว่าง
- ลิสต์อาจจะเต็มได้ เมื่อใช้ตำแหน่งที่ว่างจนหมด
- ถ้ามีการลบ และแทรกข้อมูลมากๆ ตำแหน่งที่ว่างก็กระจัดกระจาย

การสร้างลิงค์ลิสต์โดยใช้ตัวแปรพลวัตร

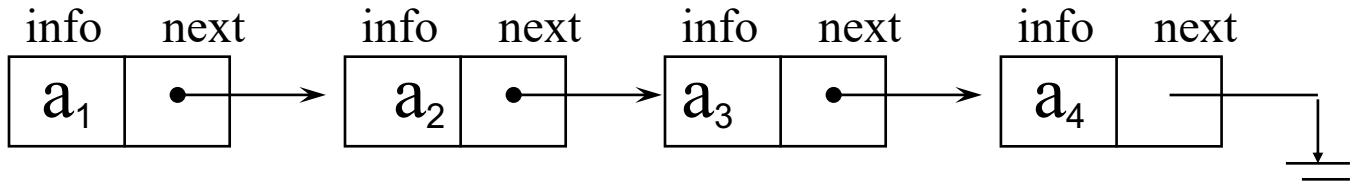
การสร้างลิสต์ด้วยตัวแปรแบบ Dynamic



- ตัวแปร next เป็นตัวแปรประเภท pointer ที่เก็บค่าที่อยู่หรือตำแหน่งของหน่วยความจำ ในลิสต์ประเภทนี้ ค่าของ next จะเก็บตำแหน่งบนของโนดที่เก็บข้อมูลตัวถัดไปนั่นเอง



Linked List



- ลิงค์ลิสต์ประกอบไปด้วยโครงสร้างโหนด ที่เชื่อมโยงกัน โดยที่แต่ละโหนดไม่จำเป็นต้องอยู่ติดกันในหน่วยความจำ
- โหนดจะประกอบไปด้วยส่วนที่เก็บข้อมูล info และส่วนที่เป็นตัวชี้ (Pointer) next ที่จะชี้ไปยังตำแหน่งของโหนดที่เก็บข้อมูลตัวถัดไป
- โหนดของข้อมูลตัวสุดท้ายค่าของ next จะเป็นค่าว่าง (null) หรือค่า 0 ในภาษา C

ข้อดีและข้อเสียของการสร้างลิสต์ด้วยตัวแปรแบบ dynamic

ข้อดี

- ความยาวของลิสต์สามารถยืดหยุ่นได้เมื่อมีการแทรกหรือลบข้อมูล
- สามารถจัดเรียงข้อมูลในลิสต์ได้ดีกว่า ถ้ามีการแทรกหรือลบข้อมูลไม่ต้องเคลื่อนย้ายข้อมูลตัวอื่นๆ

ข้อเสีย

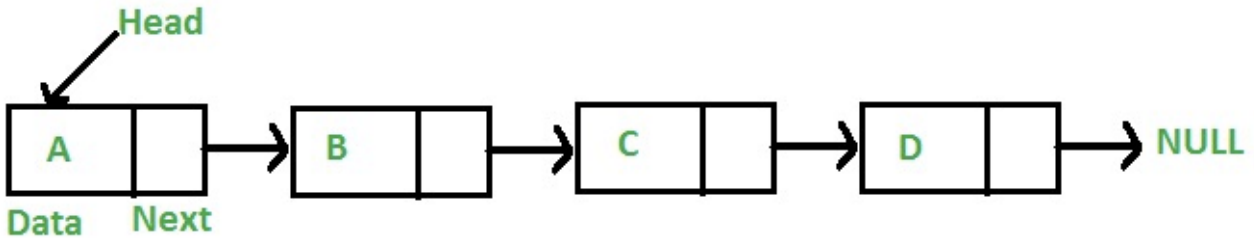
- ต้องเสียเนื้อที่เพิ่มขึ้นสำหรับตัวชี้ next ของแต่ละโนด
- เสียเวลาในการต้องขอเนื้อที่และคืนเนื้อที่ให้กับระบบตลอดการทำงานของโปรแกรม

Operations ทั่วไปของลิสต์

print()	แสดงข้อมูลทั้งหมดในลิสต์
empty()	ทำให้ลิสต์กลายเป็นลิสต์ว่างหรือลบข้อมูลทั้งหมดในลิสต์
find(info)	ค้นหาข้อมูล info ในลิสต์และแสดงตำแหน่งที่พบข้อมูลในลิสต์
insert(position, info)	แทรกข้อมูล info ในลิสต์ที่ตำแหน่ง position
delete (position)	ลบข้อมูลที่ตำแหน่ง position ออกจากลิสต์

การประกาศโหนดสำหรับลิสต์ของตัวเลข

```
struct node {
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```



การสร้าง Linked List

```
// Example 01_CreateList.c
NODEPTR head = NULL;
NODEPTR second = NULL;
NODEPTR third = NULL;

// allocate 3 nodes in the heap
    head = new Node();
    second = new Node();
    third = new Node();
```

การแทรกโหนดเข้าไปใน Linked List

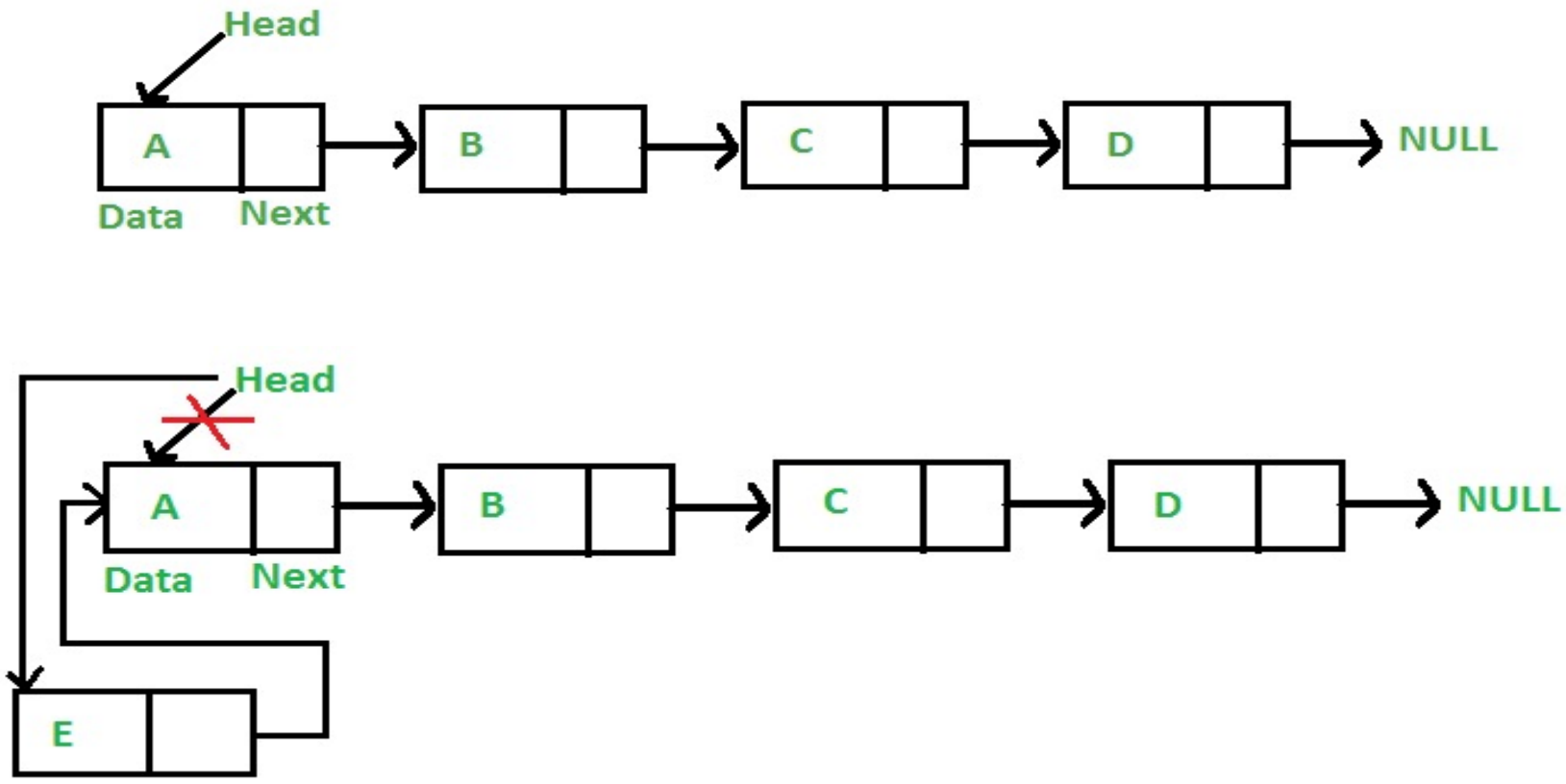
A node can be added in three ways

1) At the front of the linked list

2) After a given node.

3) At the end of the linked list.

Insert a node at the front



4 step to push a node

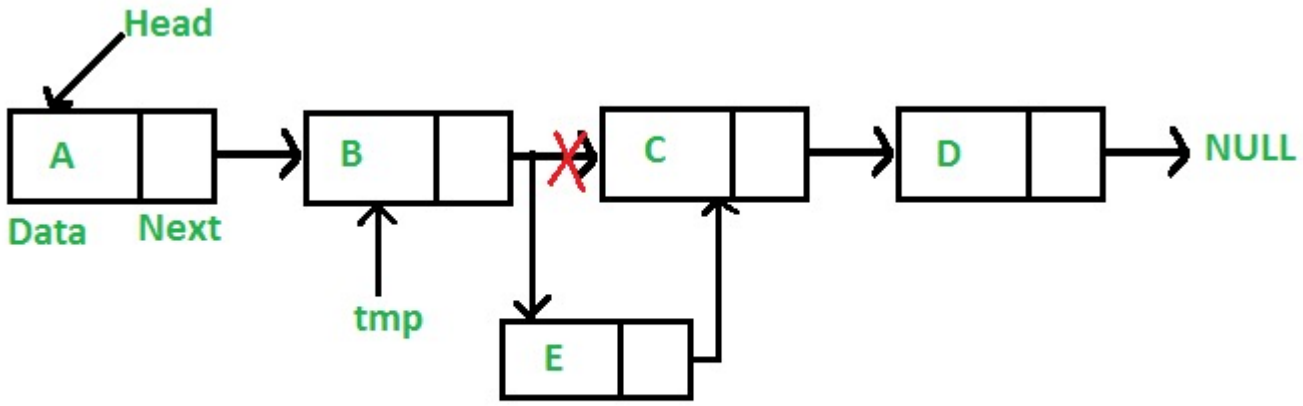
1. allocate node
2. put in the data
3. Make next of new node as head
4. move the head to point to the new node

- Time complexity of push() is O(1) as it does constant amount of work.

push()

```
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));


    /* 2. put in the data */
    new_node->data = new_data;


    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);


    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

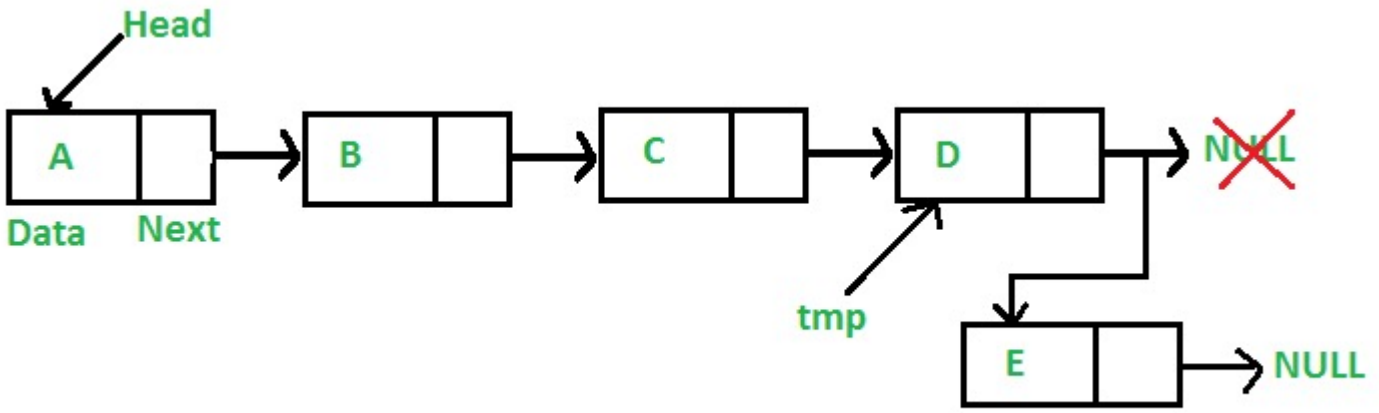
Add a node after a given node



Add a node after a given node (1)

```
void insertAfter(struct Node* prev_node, int new_data)
{
    if (prev_node == NULL){/*1. check if the given prev_node is NULL */
         printf("the given previous node cannot be NULL");
         return;
    }

    /* 2. allocate new node */
    struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

Add a node at the end

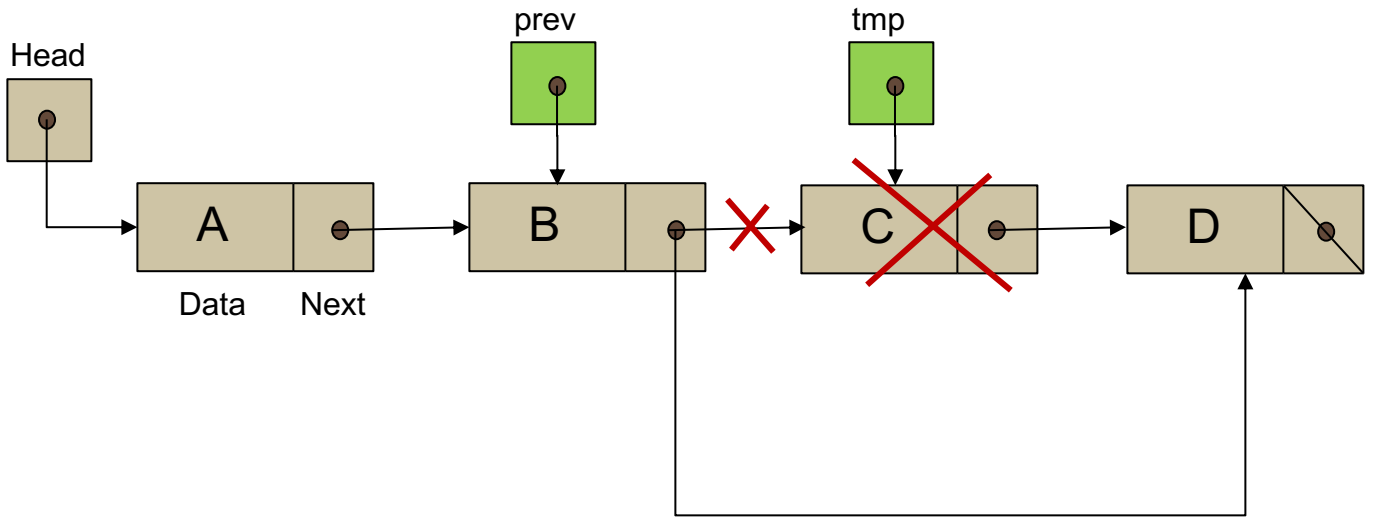


```c
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    struct Node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next of
        it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```

Deleting a Node

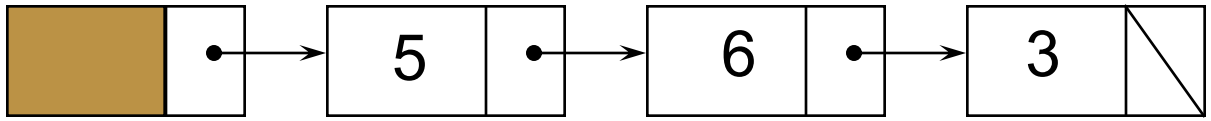
To delete a node from the linked list, we need to do the following steps.

1) Find the previous node of the node to be deleted.
2) Change the next of the previous node.
3) Free memory for the node to be deleted.

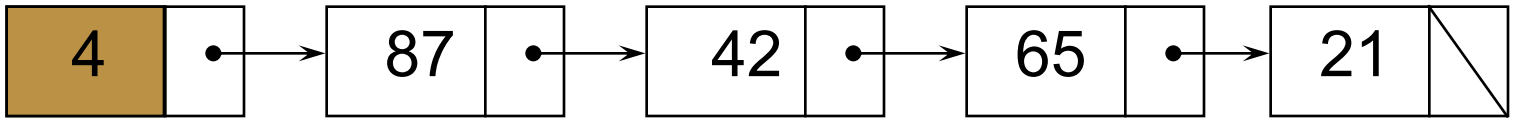


Header Nodes

- Header Nodes คือโนดพิเศษที่อยู่ที่หัวของลิสต์ โนดที่เป็น header จะไม่เก็บข้อมูล ฉะนั้นข้อมูลตัวแรกจะอยู่ที่โนดถัดไป ในบางครั้งเราอาจจะใช้ header nodes เก็บข้อมูลทั่วไปเกี่ยวกับลิสต์



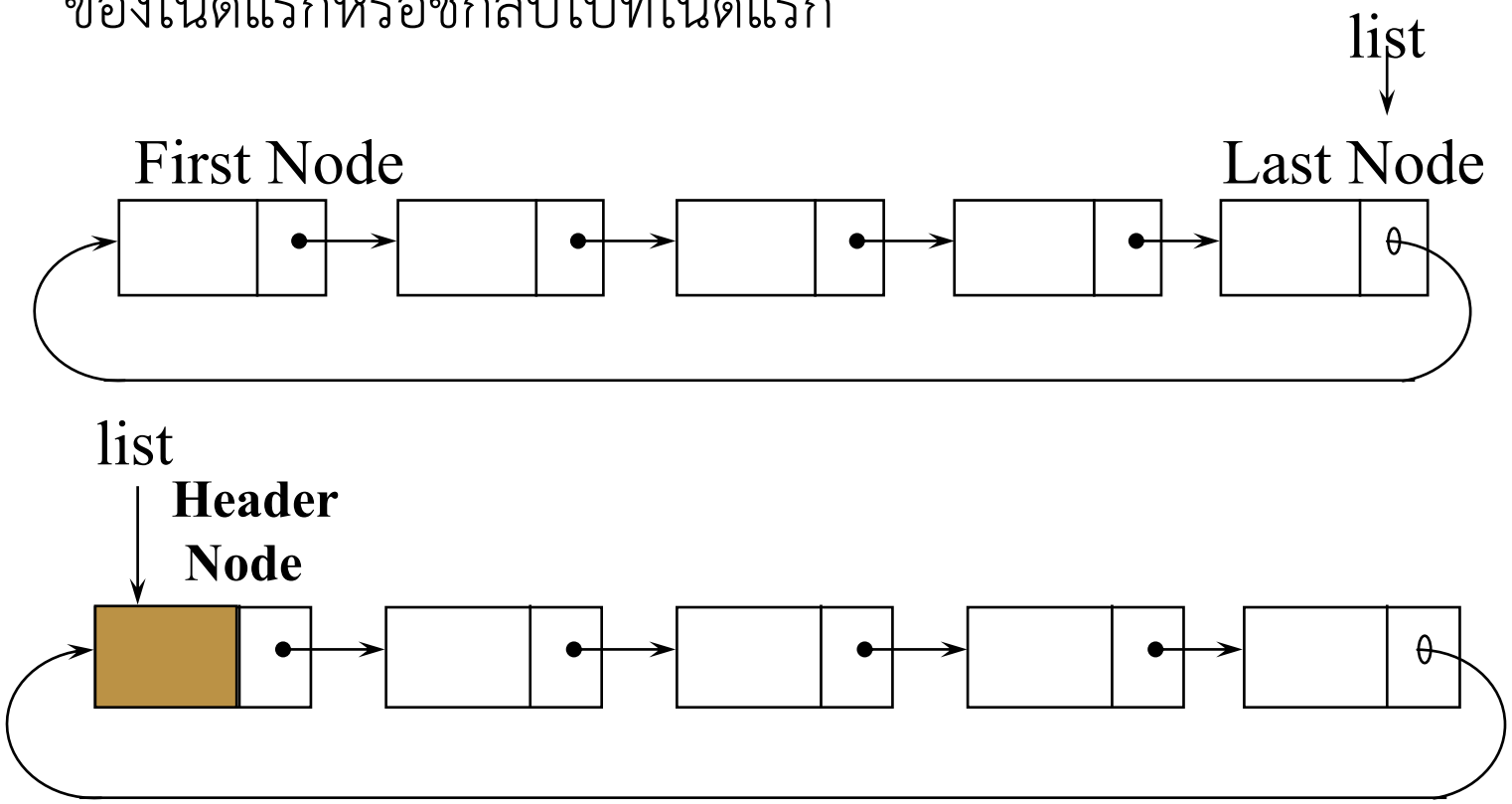
ลิสต์ที่มี header node แต่ไม่ได้เก็บข้อมูล



ลิสต์ที่มี header node และเก็บจำนวนโนดทั้งหมดของลิสต์

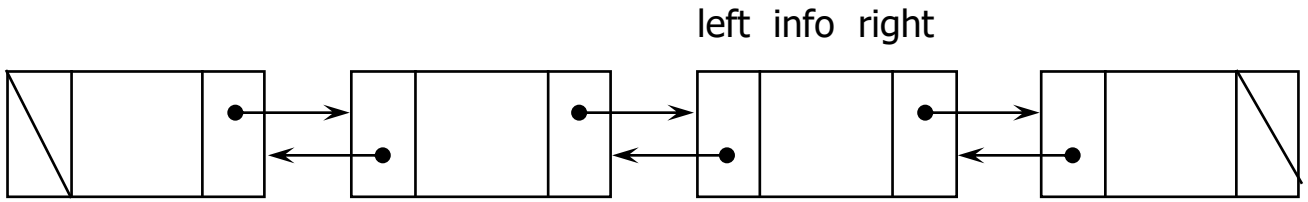
Circular Lists

คือ ลิสต์ที่ค่าของ *next* ที่โนดสุดท้ายไม่เป็น *null* แต่จะเก็บค่าที่อยู่ของโนดแรกหรือชี้กลับไปที่โนดแรก



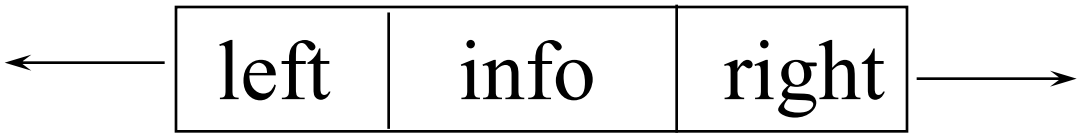
Double Linked Lists

คือ ลิสต์ที่แต่ละโนดจะประกอบไปด้วย ส่วนที่เก็บข้อมูล และตัวชี้ที่เก็บตำแหน่งของโนดก่อนหน้าและตัวชี้ที่เก็บตำแหน่งของโนดถัดไป ดังรูป



โครงสร้างโนดของ Double Linked List

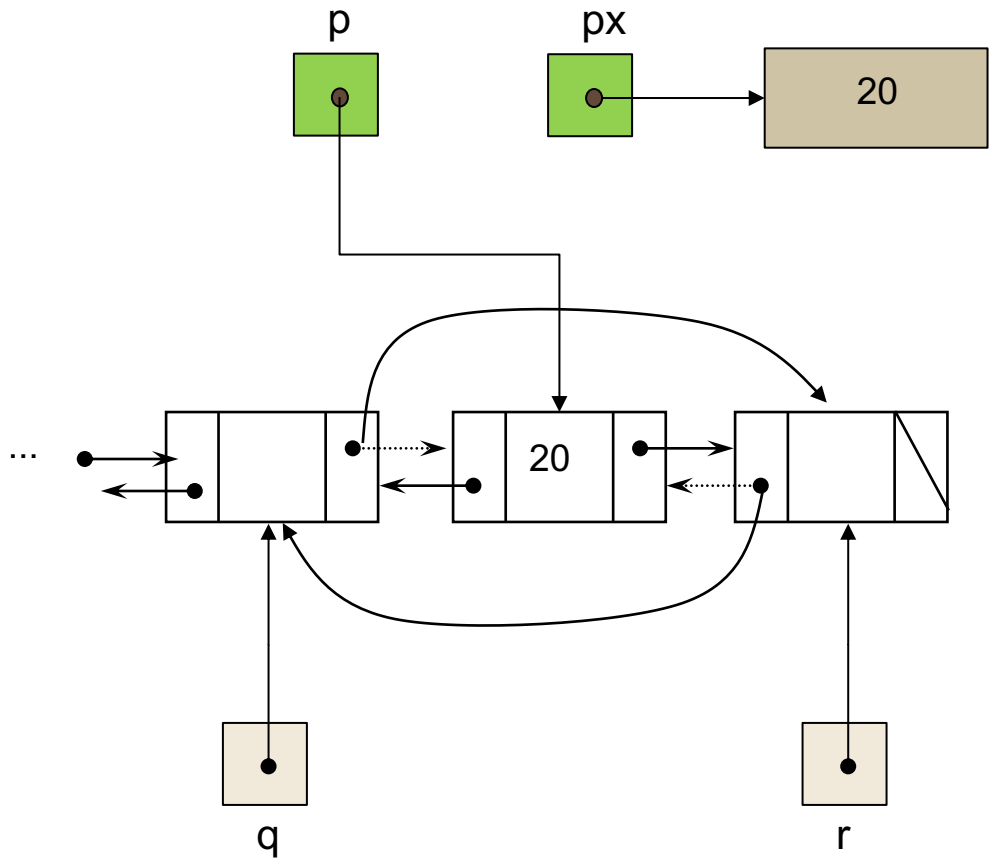
```
struct node {
        int info;
        struct node *left;
        struct node *right;
};
typedef struct node *NODEPTR;
```



```
/* delete node p and store deleted value in *px
*/
void delete(NODEPTR p, int *px)
{
        NODEPTR q, r;
        if (p = = NULL) {
                printf("void deletion\n");
                return;
        }  /* end if */
        *px = p->info;
        q = p->left;
        r = p->right;
        q->right = r;
        r->left = q;
        freenode(p);
        return;
}  /* end delete  */
```

p->left->right = p->right

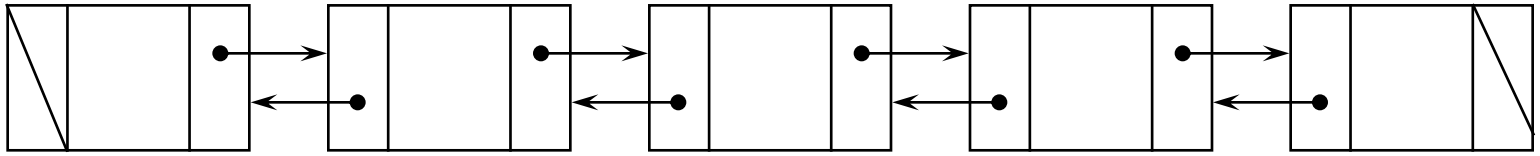
p->right->left = p->left



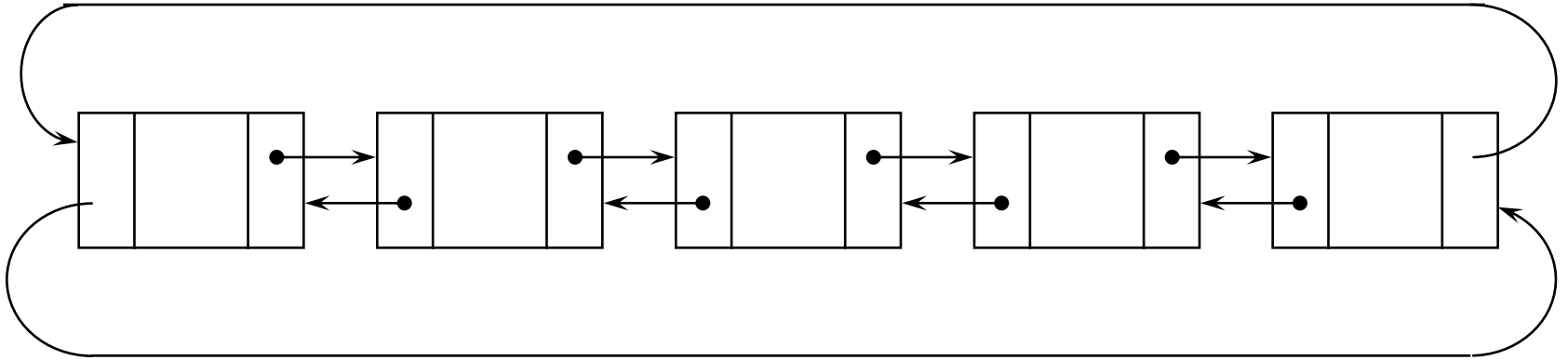
```
void  insertafter(NODEPTR p, int x)
{
        NODEPTR q, r;
        if (p = = NULL) {
                printf("void insertion\n");
                return;
        }  /* end if */
        q = getnode( );
        q->info = x;
        r = p->right;
        r->left = q;
        q->right = r;
        q->left = p;
        p->right = q;
        return;
} /* end insertafter */
```

p->right->left = q

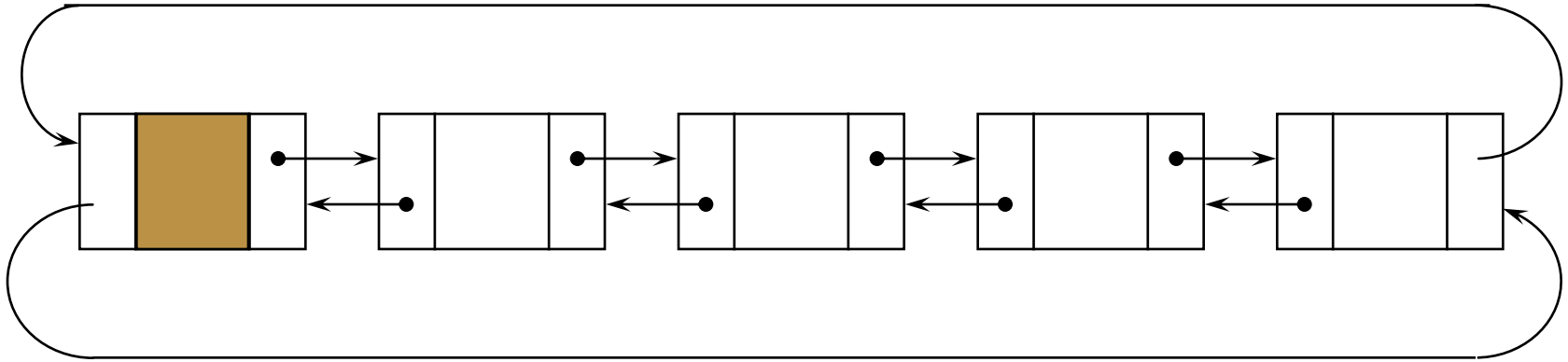
q->right = p->right



(a) A linear double linked list.



(b) A circular double linked list without a header.



(c) A circular double linked list with a header.

Lab1: Count Pairs whose sum is equal to X

Given two linked list of size **N1** and **N2** respectively of distinct elements, your task is to complete the function **countPairs()**, which returns the count of all pairs from both lists whose sum is equal to the given value **X**.
Note: The 2 numbers of a pair should be parts of different lists.

Example 2:
Input:
L1 = 7->5->1->3
L2 = 3->5->2->8
X = 10
Output: 2
Explanation: There are 2 pairs that add up to 10 : (7,3) and (5,5)

Example 1:
Input:
L1 = 1->2->3->4->5->6
L2 = 11->12->13
X = 15
Output: 3
Explanation: There are 3 pairs that add up to 15 : (4,11) , (3,12) and (2,13)

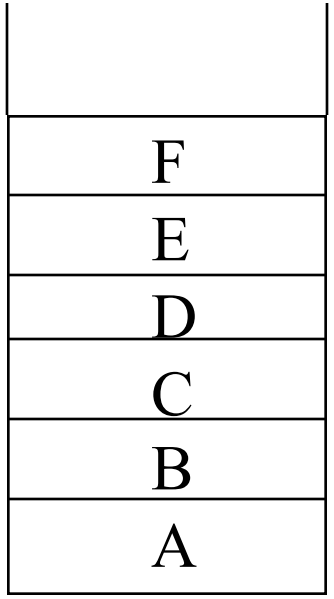
Your Task:
You only need to implement the given function **countPairs()** and return the count.
Expected Time Complexity: O(N+M)
Expected Auxiliary Space: O(N+M)

สแตก (Stack)

Linear Data Structures

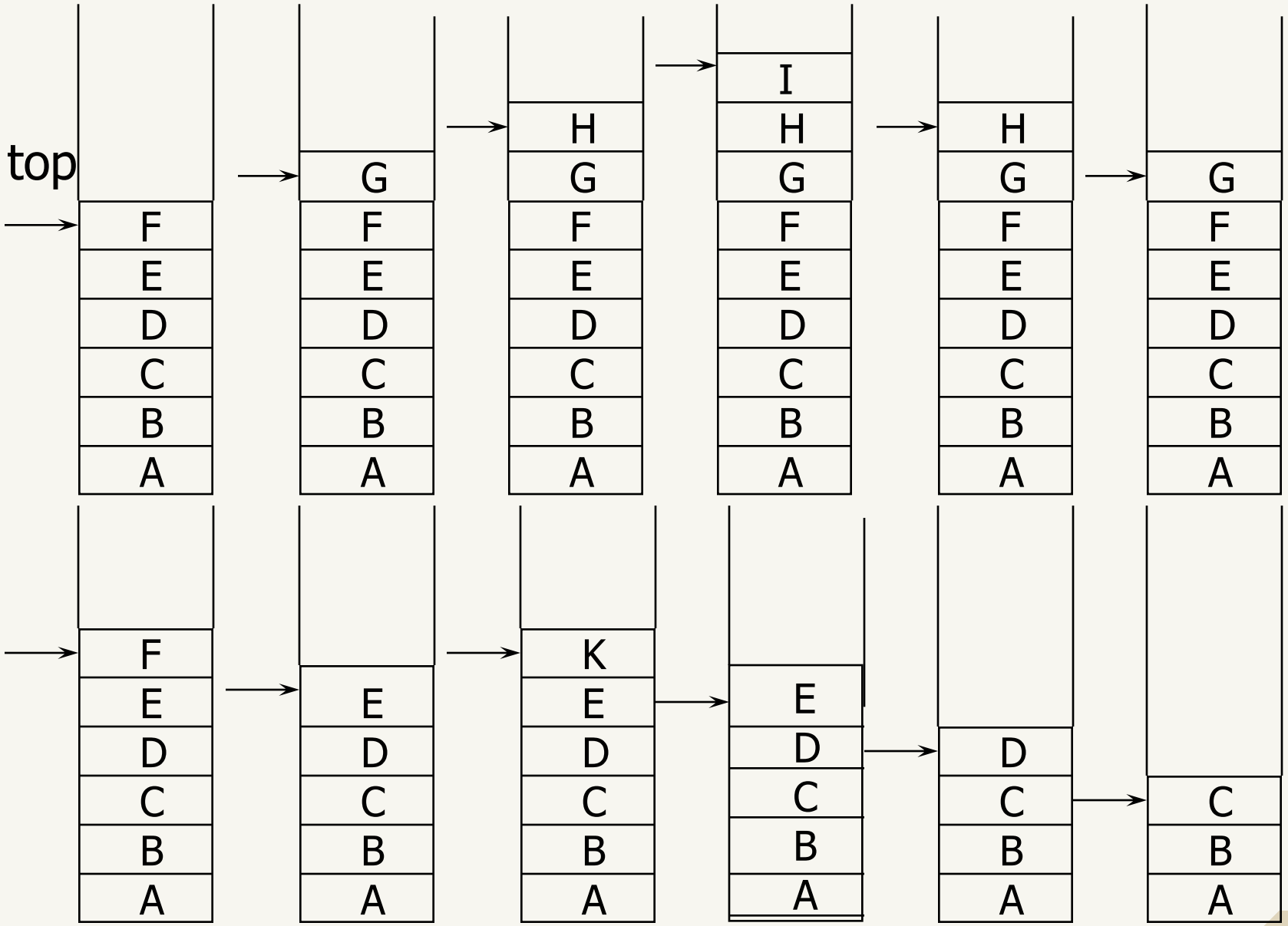
สแตก (Stack)

สแตก คือ กลุ่มของข้อมูลที่จัดเรียงกันอย่างมีลำดับ โดยการเพิ่มหรือลบข้อมูลจะกระทำที่จุดจุดเดียวกันคือที่ด้านบนของสแตกที่เรียกว่า *top* ของสแตก



← top of stack

ข้อมูลที่ถูกเพิ่มเข้าไปในสแตกตัวสุดท้ายจะเป็นข้อมูลตัวแรกที่ถูกลบออกจากสแตก ทำให้ในบางครั้งเราจึงเรียกสแตกว่าเป็นลิสต์แบบ *last-in, first-out* (LIFO)


top
F
E
D
C
B
A
G
F
E
D
C
B
A
H
G
F
E
D
C
B
A
I
H
G
F
E
D
C
B
A
H
G
F
E
D
C
B
A
G
F
E
D
C
B
A
F
E
D
C
B
A
E
D
C
B
A
K
E
D
C
B
A
E
D
C
B
A
D
C
B
A
C
B
A

Operations ทั่วไปของสแตก

กำหนดให้ *s* แทนสแตกและ *i* แทนข้อมูล

push(s, i) : เพิ่มข้อมูล i ที่ตำแหน่ง top ในสแตก s

pop(s) : ลบข้อมูลที่ตำแหน่ง top ของสแตกและส่งกลับข้อมูลนั้น

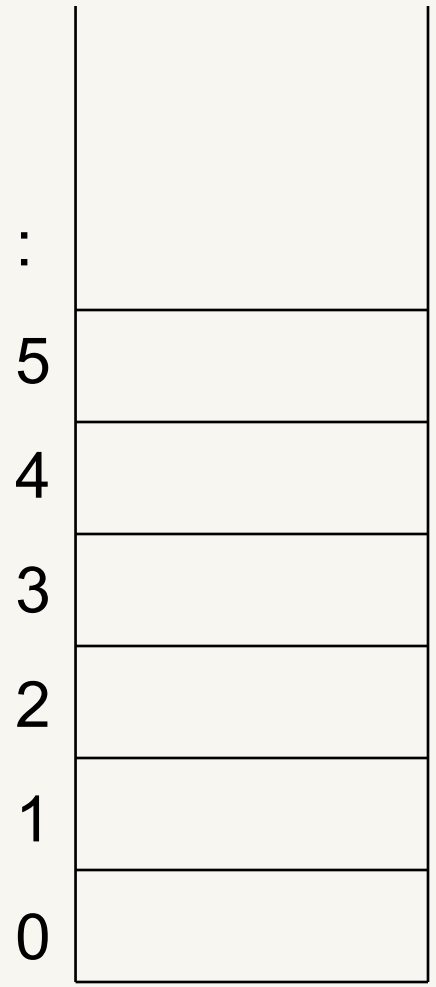
empty(s) : ตรวจสอบว่าสแตก s ว่างหรือไม่

stacktop(s) : ดูข้อมูลที่ตำแหน่ง top ของสแตกโดยก็อบปี้ข้อมูลออกมา และไม่ลบข้อมูลออก

การสร้างสแตกด้วยอะเรย์

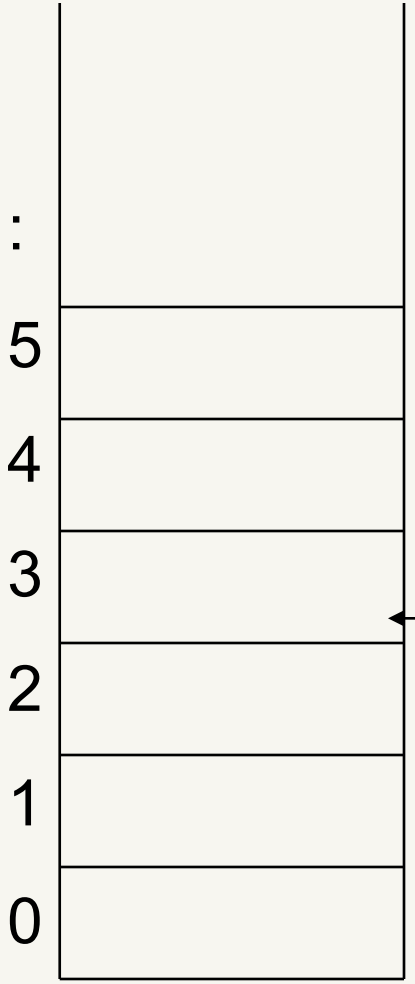
การสร้าง stack ในภาษา C สามารถกำหนดเป็นโครงสร้างที่ประกอบไปด้วยข้อมูลสองส่วนคือ ตำแหน่งบนสุดของแสตก (top) และอะเรย์ที่ใช้เก็บข้อมูล (item)

```
#define STACKSIZE 100
struct stack {
        int top;
        int items[STACKSIZE];
};
struct stack s;
```



Top = -1

(a) แสตกว่าง



Top = 3

(b) แสตกที่มีข้อมูล 4 ตัว

ฟังก์ชัน empty()

การให้ค่าเริ่มต้นของ stack เป็น stack ว่าง สามารถทำได้โดยการเซตค่า *s.top* = -1;

```
int empty(struct stack *ps)
{
  if (ps->top = = -1)
          return (1);
  else
          return (0);
} /* end empty */
```

ฟังก์ชัน push()

```
void push(struct stack *ps, int x)
{
        if (ps->top = = STACKSIZE-1) {
            printf("%s", "stack overflow");
            exit(1);
}
else
        ps->item[++(ps->top)] = x;

return;
}  /* end push */
```

ฟังก์ชัน pop()

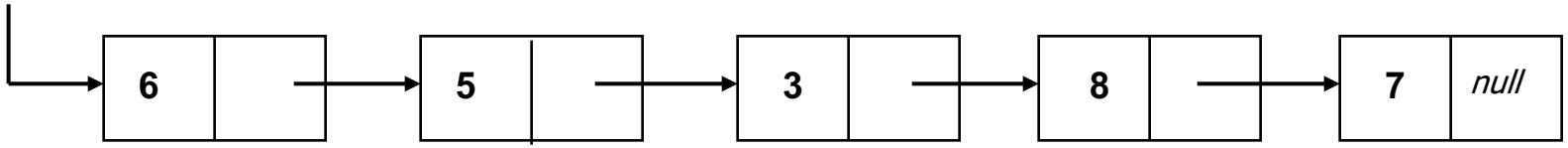
```
int pop(struct stack *ps)
{
    if (empty(ps))  {
        printf("%", "stack underflow");
        exit(1);
    }  /* end if */

    return (ps->item[ps->top--]);
}  /* end pop */
```

ฟังก์ชัน stacktop()

```
int stacktop(struct stack *ps)
{
        if (empty(ps))  {
            printf("%", "stack underflow");
            exit(1);
        }  /* end if */

        return (ps->item[ps->top]);
}  /* end pop */
```

ส่งคืนค่าที่ตำแหน่งบนสุดของแสตก โดยไม่ลบออก

การสร้างสแตกด้วยลิงค์ลิสต์



push(s,x)	**x = pop(s)**

p = getnode();
info(p) = x;
next(p) = s;
s = p;
	if (empty(s)) {
	  printf(”Stack underflow”);
	  exit(1);
	}
	else {
	  p = s;
	  s = next(p);
	  x = info(p);
	  freenode(p);
	}

การใช้สแตกในการแก้ปัญหา

ตัวอย่าง : พิจารณานิพจน์ทางคณิตศาสตร์ ดังต่อไปนี้ :

$$\{[A+B] - [C * (D - E)]\}$$

เราต้องตรวจสอบว่า

1. จำนวนวงเล็บเปิดเท่ากับจำนวนวงเล็บปิดหรือไม่
2. ทุกๆ วงเล็บปิดจะต้องมีวงเล็บเปิดที่คู่กันอยู่ก่อนหน้าเสมอหรือไม่
3. วงเล็บเปิดและวงเล็บปิดเป็นวงเล็บชนิดเดียวกันหรือไม่

วิธีการแก้ปัญหาโดยไม่ใช้สแตก เราอาจใช้วิธีการนับวงเล็บ โดยนำจำนวนวงเล็บเปิดที่นับได้ลบด้วยจำนวนวงเล็บปิดที่นับได้

- ถ้าผลลัพธ์เป็นศูนย์ แสดงว่าจำนวนวงเล็บเปิดเท่ากับจำนวนวงเล็บปิด
- ค่าที่ได้จากการนับ ณ ขณะใดๆ ต้องไม่เป็นค่าลบ เนื่องจากเราเอาจำนวนวงเล็บเปิดลบด้วยจำนวนวงเล็บปิด ถ้าค่าเป็นลบแสดงว่าวงเล็บปิดมาก่อนวงเล็บเปิด

ตัวอย่าง :

7 - ((x * ((x + y) / (j - 3)) + y) / (4 - 2.5))

0 0 **1 2** 2 2 **3 4** 4 4 4 **3** 3 **4** 4 4 4 **3 2** 2 2 **1** 1 **2** 2 2 2 **1 0**

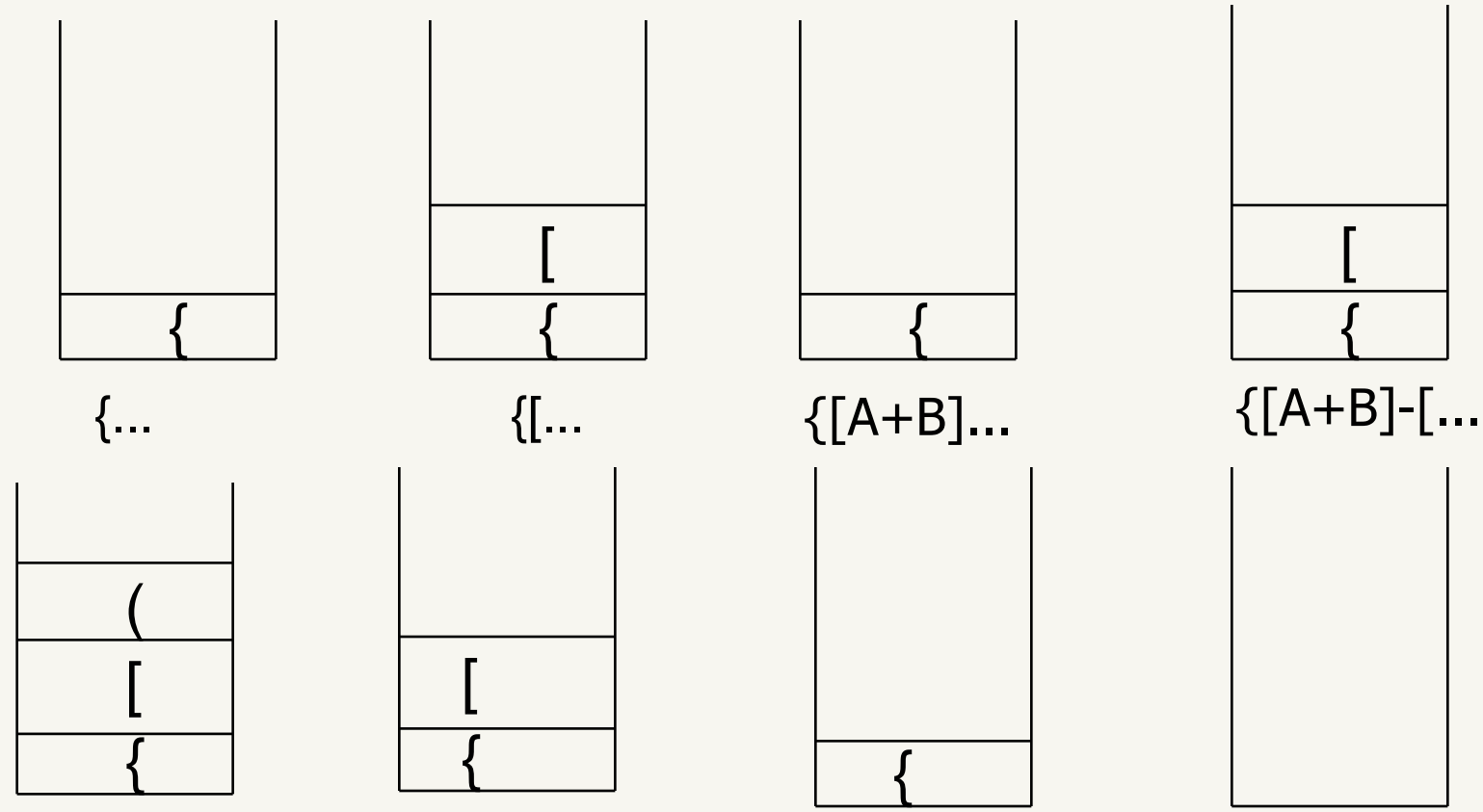
((A + B)

1 2 2 2 2 **1**

(A + B)) - (C + D

1 1 1 1 **0 -1** -1 **0** 0 0 0

- ถ้ามีการใช้วงเล็บหลายแบบ เช่น **{[A+B] - [C*(D-E)]}**

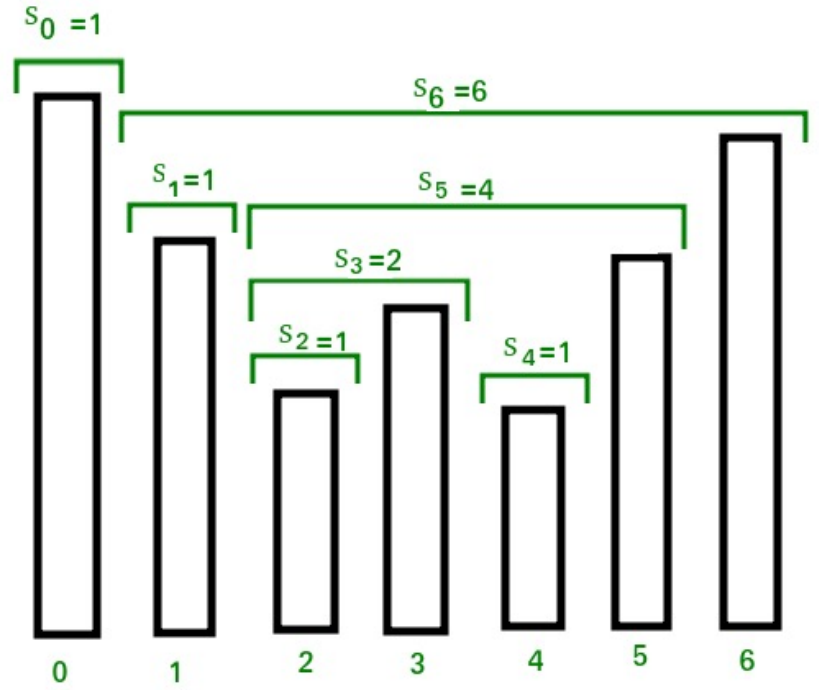


```
valid = true;
s = the empty stack;
while (we have not read the entire string) {
        read the next symbol (symb ) of the string ;
        if  (symb == '(' ) || (symb == '[' ) || (symb == '{' )
            push(s, symb);
        if  (symb == ')' ) || (symb == ']' ) || (symb == '}' )
             if  (empty(s))
                 valid = false;
            else {
                      i = pop(s);
            if ( i  is not the matching opener of symb)
                 valid = false;
            }
}
if  (empty(s))
    valid = false;
```

Lab2: The Stock Span Problem

ปัญหาช่วงเวลาของราคาหุ้นที่ขึ้นสูงเป็นปัญหาทางการเงินที่เราจะคำนวณช่วงราคาหุ้นตลอด n วัน โดยจะคำนวณหาว่าราคาหุ้นในวันที่ i มีจำนวนวันก่อนหน้าที่ราคาหุ้นน้อยกว่าหรือเท่ากับติดต่อกันสูงสุดกี่วัน ซึ่งก็คือค่าสแปนของวันที่ i (S_i)

ตัวอย่างเช่นหากกำหนดราคา 7 วันเป็น {100, 80, 60, 70, 60, 75, 85} ค่าสแปนสำหรับ 7 วันที่สอดคล้องกันคือ {1, 1, 1, 2, 1, 4 , 6}



Hint

1. In this approach, I have used the data structure stack to implement this task.
2. Here, two stacks are used. One stack stores the actual stock prices whereas, the other stack is a temporary stack.
3. The stock span problem is solved using only the Push and Pop functions of Stack.
4. Just to take input values, I have taken array ‘price’ and to store output, used array ‘span’.

คิว (Queue)

Linear Data Structures

คิว (QUEUES)

คิว เป็นกลุ่มหรือลิสต์ของข้อมูลที่เรียงกันอย่างมีลำดับ การเพิ่มข้อมูลในคิวจะเพิ่มที่ด้านท้ายหรือที่เรียกว่า rear ของคิว ส่วนการลบข้อมูลจะลบที่ด้านหน้าหรือ front ของคิว

จากคุณลักษณะเฉพาะดังกล่าว ในบางครั้งเราจึงเรียกคิวว่าเป็น first-in, first-out หรือ FIFO list

Operations ทั่วไปของคิว

enqueue(*q*,*x*)

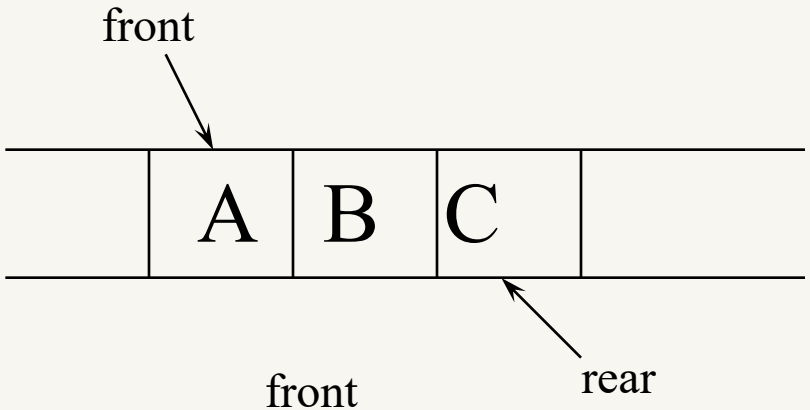
เพิ่มข้อมูล x ที่ตำแหน่งท้ายของคิว *q*

x = dequeue(*q*)

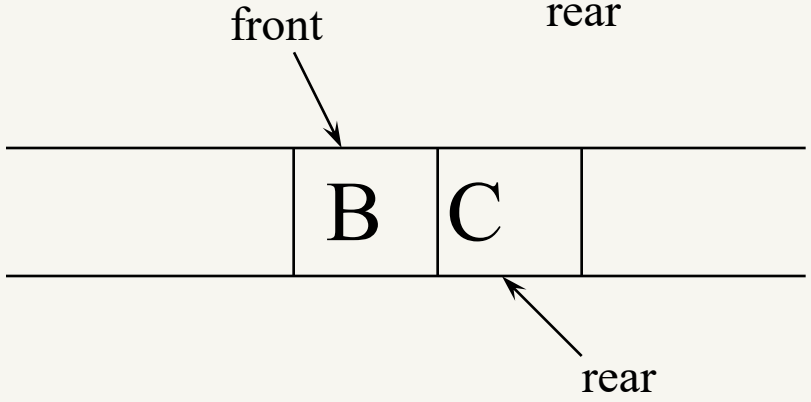
ลบข้อมูลที่ตำแหน่งแรก (front) ของคิว q แล้วส่งกลับค่าให้ตัวแปร x

empty(*q*)

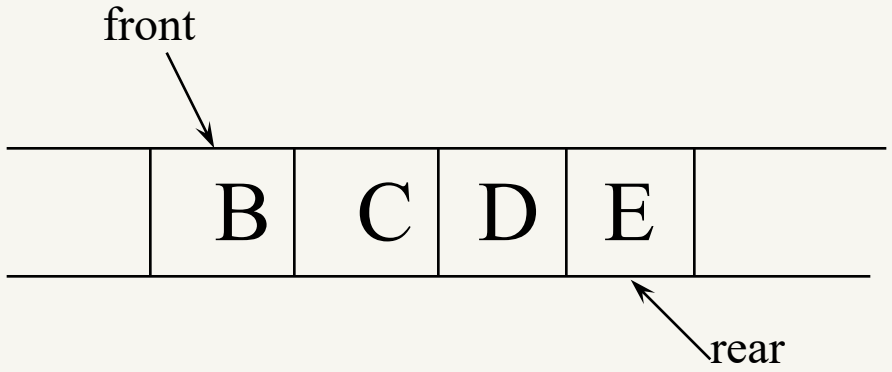
ตรวจสอบว่าคิว q เป็นคิวว่างหรือไม่ (คิวว่างคือคิวที่ไม่มีข้อมูลเลย)



enqueue(q, A);
enqueue(q, B);
enqueue(q, C);



x = dequeue(q);



enqueue(q, D);
enqueue(q, E);

การสร้างคิวด้วยอะเรย์

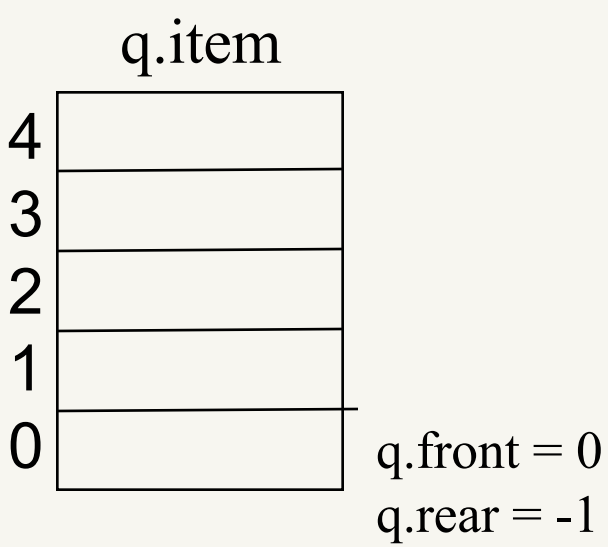
```
#define  MAXQUEUE  5
struct queue {
        int items[MAXQUEUE];
        int front;
        int rear;
}q;
```

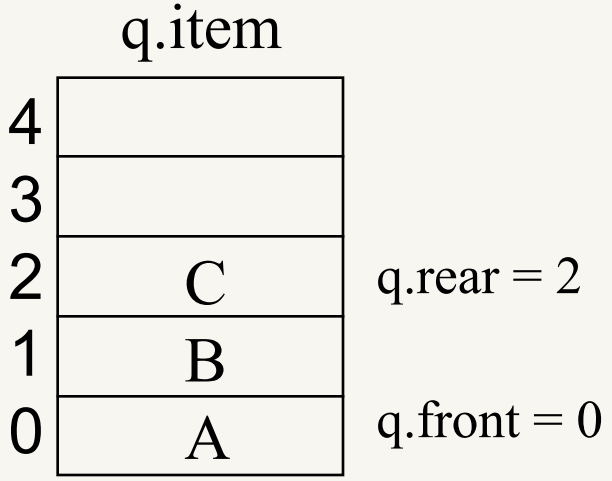
enqueue(q, x) สามารถเขียนได้ดังนี้

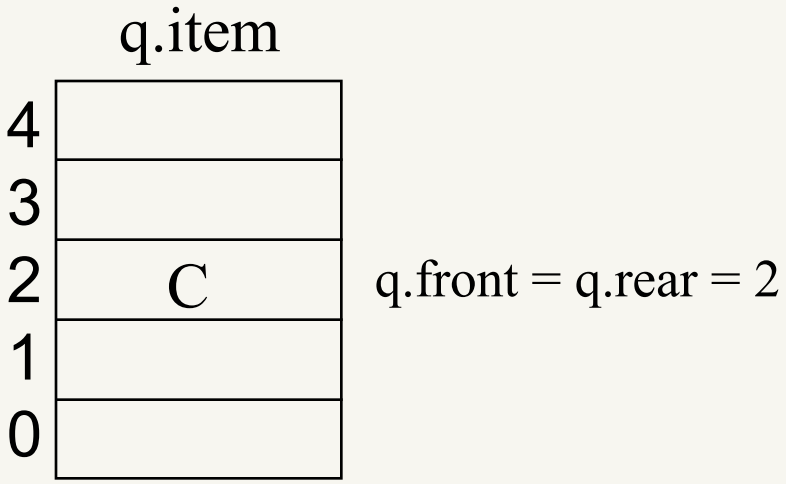
```
q.items [++q.rear] = x;
```

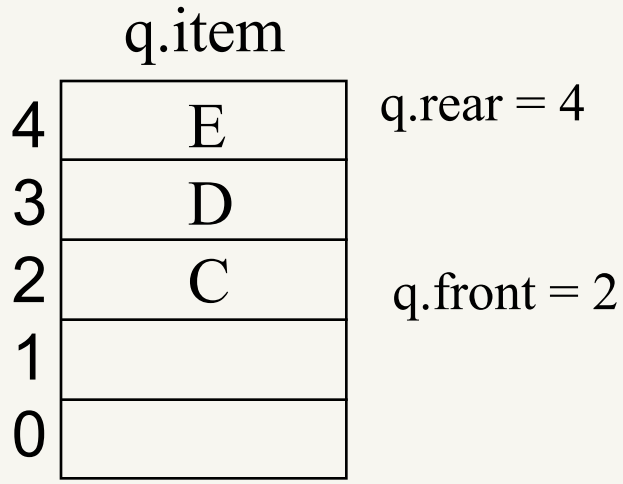
x = dequeue(q) สามารถเขียนได้ดังนี้

```
x = q.item [q.front++];
```


q.item
4
3
2
1
0
q.front = 0
q.rear = -1


q.item
4
3
2
C
1
B
0
A
q.rear = 2
q.front = 0


q.item
4
3
2
C
1
0
q.front = q.rear = 2


q.item
4
E
3
D
2
C
1
0
q.rear = 4
q.front = 2

จากรูป:

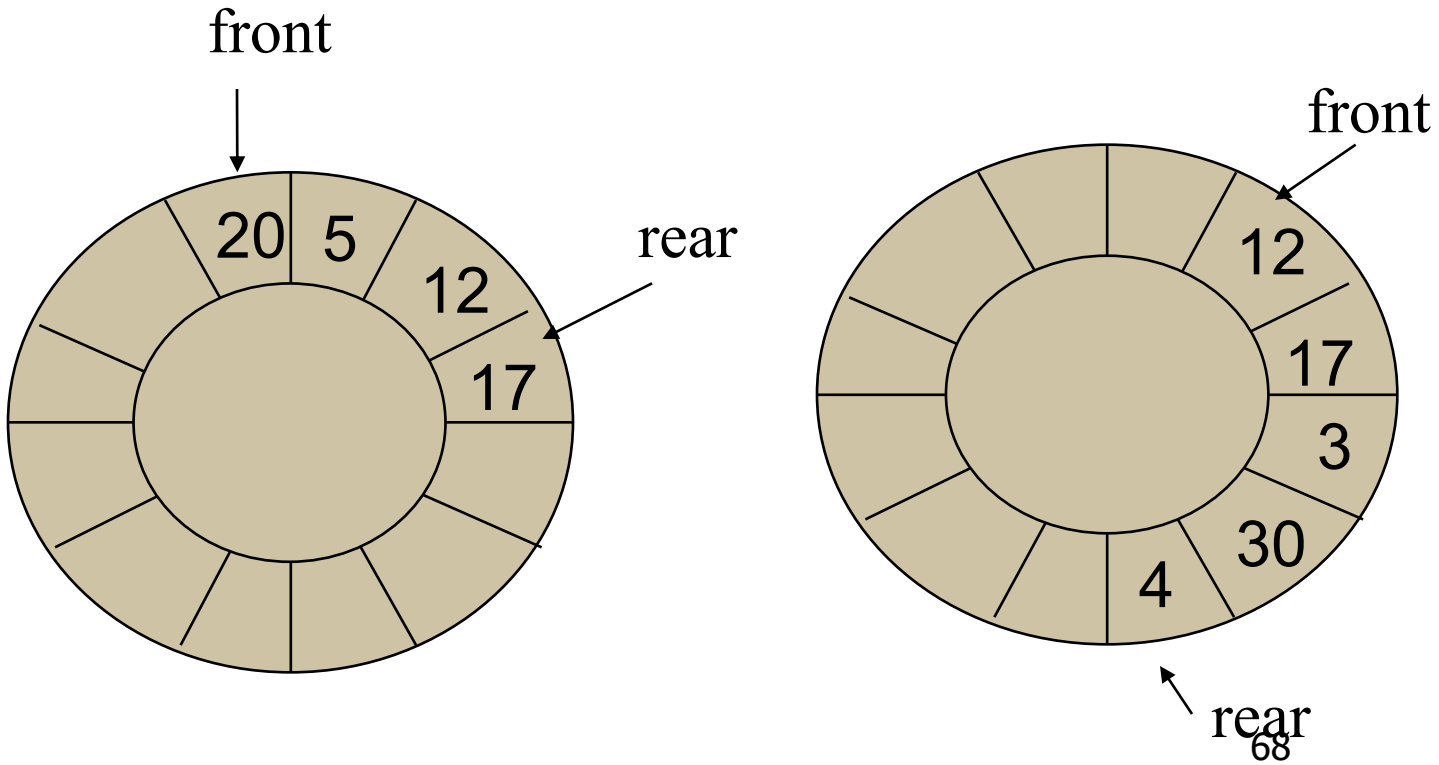
เริ่มต้น	:	q.front = 0, q.rear = -1
คิวว่าง	:	q.rear < q.front
จำนวนข้อมูล	:	q.rear - q.front + 1

หลังจากแทรก “E” จะทำให้ q.rear = MAXQUEUE -1

แล้วเราจะ INSERT ข้อมูล *F* เข้าไปใน Q ได้อย่างไร

A Circular Queue

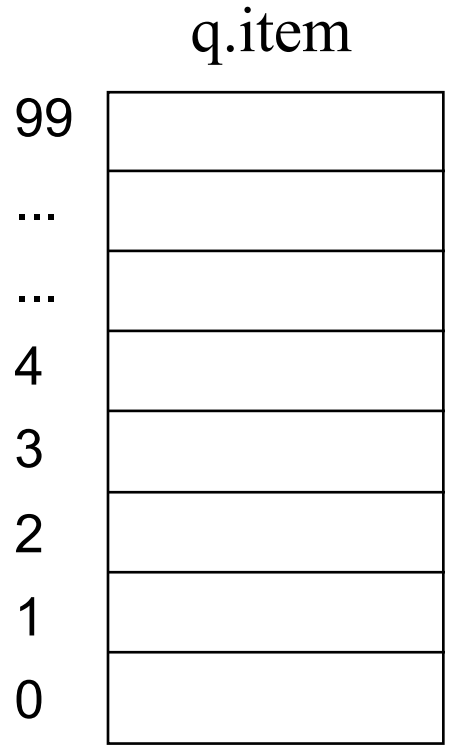
มองอะเรย์ที่เก็บคิวเป็นวงกลม



โครงสร้างของ Circular queue

```
#define  MAXQUEUE 100
struct queue {
        int item[MAXQUEUE];
        int front;
        int rear;
        int count;
};
struct queue q;
```

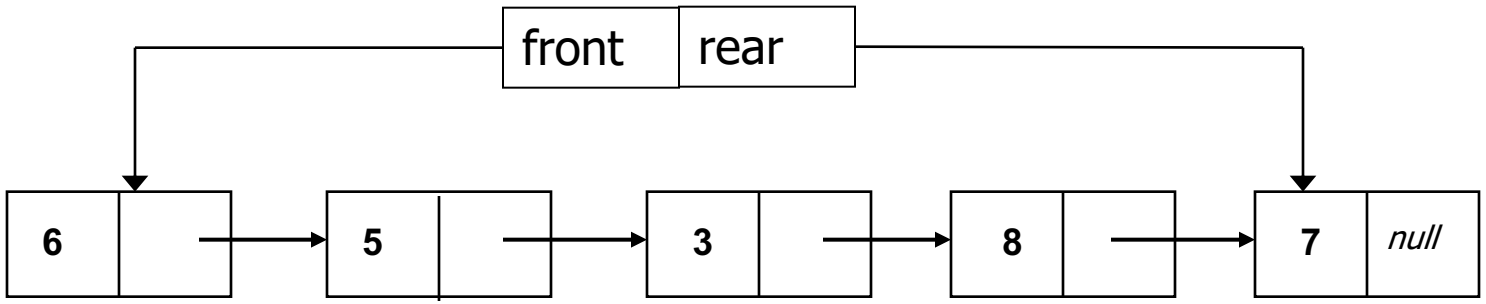
q.rear = -1 q.front = -1 q.count = 0



```
int dequeue(struct queue *pq)
{     int frontItem;

      if (empty(pq))  {
            printf("queue underflow");
            exit(1);
      }  /* endif */
      frontItem = pq->item[pq->front];
      (pq->front)++;

      if (pq->front = = MAXQUEUE-1)
            pq->front = 0;
      if (pq->count == 1)
            pq->front = pq->rear = -1
      (pq->count)--;
      return (frontItem);
}
```

```
void enqueue(int x , struct queue *pq)
{
      if ( pq->count = MAXQUEUE )    /*queue is full */
          return;
      (pq->rear)++;
      if (pq->rear = = MAXQUEUE)
            pq->rear = 0;
      pq->item[pq->rear] = x;

      if (pq->count = = 0)  {
            pq->front = 0
      pq->count++;
      return;
}
```

การสร้างคิวด้วยลิสต์




```
struct queue {
        NODEPTR front, rear;
};
struct queue q;
```

```
void insert(int x, struct queue *pq)
{
        NODEPTR p;

        p = getnode( );
        p->info = x;
        p->next = NULL;
        if (pq->rear = = NULL)
                (pq->front = p;
        else
                (pq->rear)->next = p;
                pq->rear = p;
        pq->rear = p;
}  /* end insert  */
```

```
NODEPTR getnode(void)
{
        NODEPTR p;
        p = (NODEPTR) malloc(sizeof(struct node));
        return(p);
}
```

```
int remove(struct queue *pq)
{
        NODEPTR p;
         int x;
        if (empty(pq)) {
                printf("queue underflow\n");
                exit(1);
        }
        p = pq->front;
        x = p->info;
        pq->front = p->next;
        if (pq-front = = NULL)
                pq->rear = NULL;
        freenode(p);
        return(x);
}  /*  end remove  */
```

```
void freenode( NODEPTR  p)
{
        free(p);
}
```

Priority Queue

คือโครงสร้างของคิวที่ตำแหน่งของข้อมูลในคิวไม่ได้บอกตำแหน่งของการเพิ่มหรือลบข้อมูล ข้อมูลแต่ละตัวจะมีค่า priority ที่ต่างกันและค่านี้จะเป็นตัวบอกลำดับการออกจากคิว เราสามารถแบ่งประเภทของ Priority Queue ได้เป็น 2 แบบ คือ

- Ascending priority queue คิวของข้อมูลที่ข้อมูลที่มีค่าน้อยที่สุดหรือมี priority น้อยที่สุด จะถูกลบออกจากคิวก่อน
- Descending priority queue คิวของข้อมูลที่ข้อมูลที่มีค่ามากที่สุดหรือมี priority มากที่สุด จะถูกลบออกจากคิวก่อน

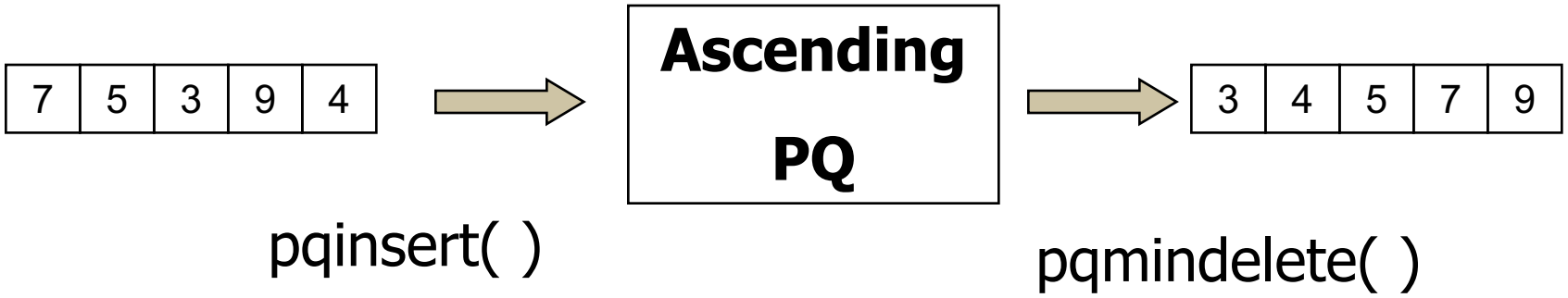
Operations ของ Priority Queue

กำหนดให้ x เป็นข้อมูล pq เป็น priority queue, apq เป็น ascending priority queue และ dpq เป็น descending priority queue

pqinsert(pq, x)	เพิ่มข้อมูล x ใน pq โดยการเพิ่มเราไม่ได้สนใจว่า ข้อมูลในคิวจะเรียงลำดับอย่างไร
dpqmaxdelete(dpq)	เป็น operation สำหรับ descending priority queue โดยทำการลบข้อมูลที่มีค่ามากที่สุดหรือมีค่า priority สูงสุดออกจากคิว
apqmindelete(apq)	เป็น operation สำหรับ ascending priority queue โดยทำการลบข้อมูลที่มีค่าน้อยที่สุดหรือมีค่า priority ต่ำสุดออกจากคิว

การเรียงลำดับข้อมูลด้วย Priority Q

- การใช้ operation ของ priority queue ในการเลือกหยิบข้อมูลน้อยที่สุดหรือมากที่สุดในคิวออกทีละตัว และเลือกวางข้อมูลที่ออกจากคิวในอะเรย์ ซึ่งถ้าวางในอะเรย์จากตำแหน่งแรกไปตำแหน่งท้าย หรือเลือกวางข้อมูลจากตำแหน่งท้ายมาตำแหน่งแรก ก็จะได้ลำดับข้อมูลที่แตกต่างกันทั้งจากน้อยไปมากหรือมากไปน้อย



ตัวอย่าง : การเรียงลำดับข้อมูลด้วย dpq

กำหนดให้ dpq เป็น descending priority queue ว่าง

x เป็นอะเรย์ของข้อมูลแบบไม่มีลำดับ ที่มีข้อมูล n จำนวน

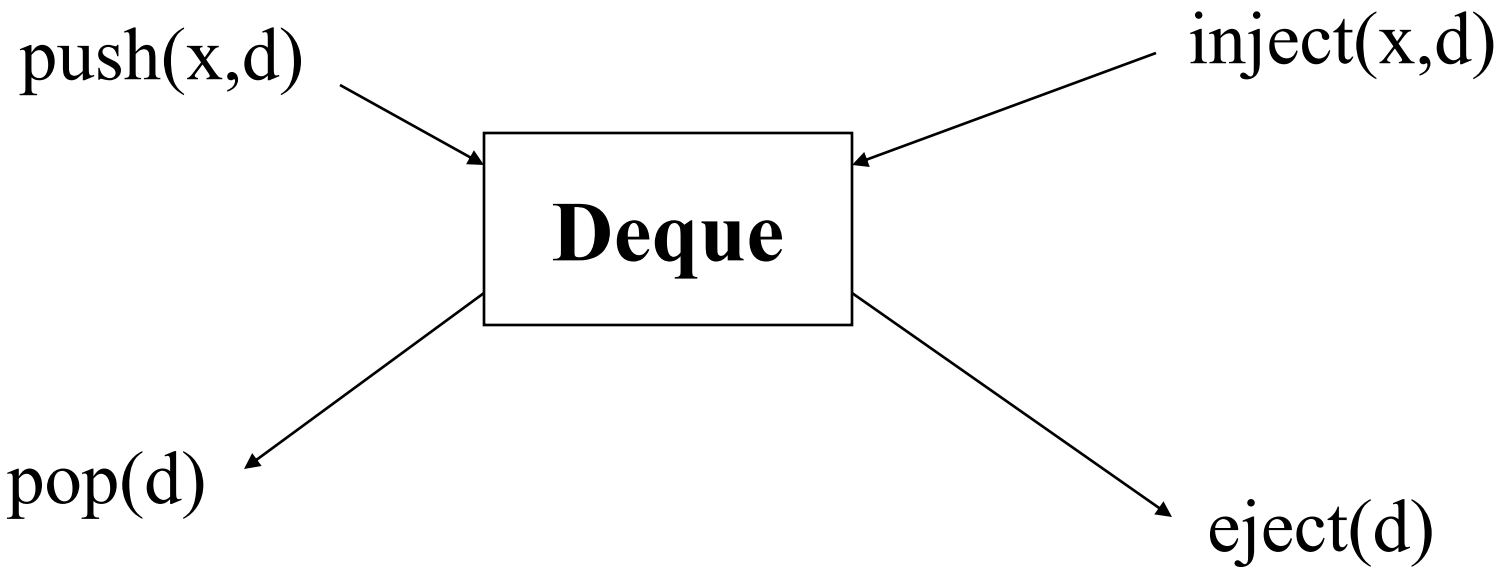
ใส่ข้อมูลในอะเรย์ x ลงใน dpq จนหมด ดังนี้

```
for (i = 0; i < n; i++)
    pqinsert(x[i]);
```

ลบข้อมูลจาก dpq แล้วใส่กลับคืนที่อะเรย์โดยเลือกวางจากตำแหน่งท้ายไปตำแหน่งแรก

```
for (i = n-1; i >= 0; i--)
    x[i] = dpqmaxdelete( );
```

Deque (Double - Ended Queue)



deque คือโครงสร้างข้อมูลที่ประกอบไปด้วยลิสต์ของข้อมูลและ operations ดังต่อไปนี้

push(x)	: การใส่ข้อมูลใหม่ไปที่ด้านหน้าของคิว
pop()	: การลบข้อมูลตัวแรกของคิว
inject(x)	: การใส่ข้อมูลใหม่ไปที่ท้ายของคิว.
eject()	: การลบข้อมูลที่ด้านท้ายของคิว

ลักษณะของ deque ก็เป็นการผสมระหว่างคิวกับสแตกนั่นเอง