

# 전산 SMP 11주차

2014. 12. 06

김범수

[bskim45@gmail.com](mailto:bskim45@gmail.com)

Special thanks to 박기석 (kisuk0521@gmail.com)

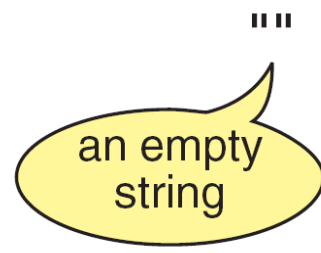
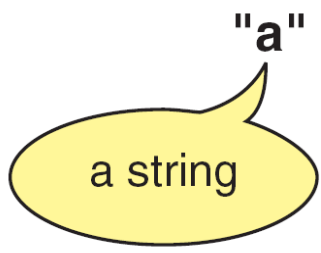
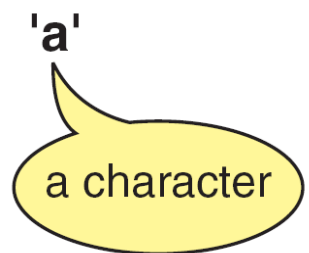
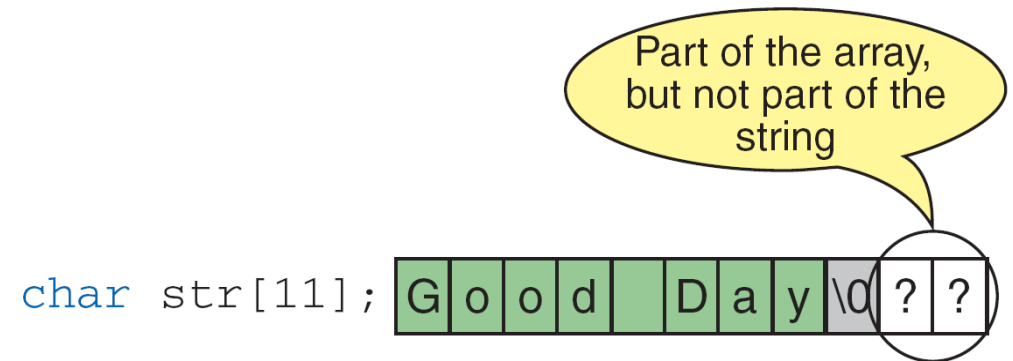
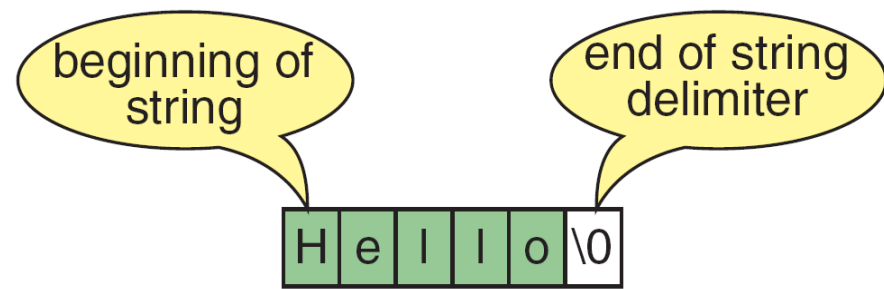
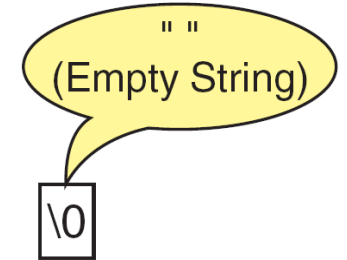
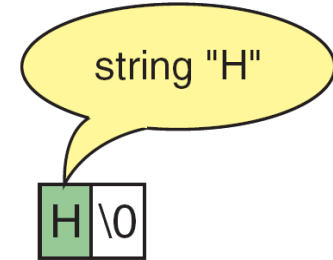
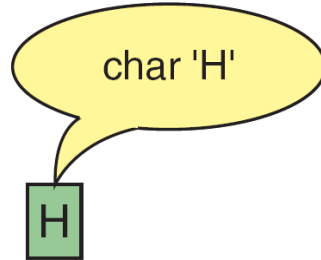
# 지난 내용 복습

String Advanced

**String**

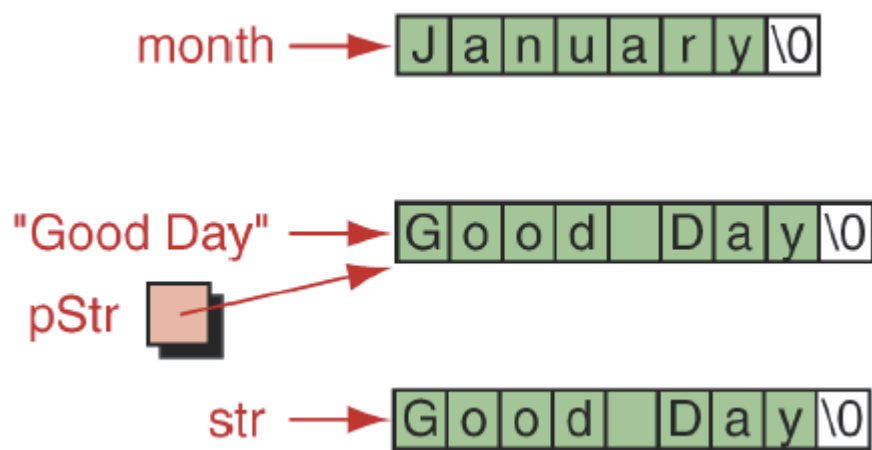
# 문자열(string)도 결국은 배열이다

- char 형의 배열
- 끝에 널문자 '\0'



# 문자열의 선언

- `char str[15] = "Good Day";`
  - 할당된 용량을 모두 사용할 필요는 없습니다.
- `char month[] = "January";`
  - 배열크기가 문자열의 길이에 맞춰 자동으로 설정
- `char *pStr = "Good Day";`
- `char str[9] = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y', '\0'};`



```
// Local Declarations
char str[9];
```

(a) String Declaration

```
// Local Declarations
char* pStr;
```

(b) String Pointer Declaration



# 문자열의 포인터

- 배열의 이름은 첫번째 element의 주소와 같다.
- 그래서 scanf로 %s 받을 때 & 안 붙인다!



# 문자열 포인터 사용 주의점

```
// Local Declarations  
char str[9];
```

(a) String Declaration



```
// Local Declarations  
char* pStr;
```

(b) String Pointer Declaration



- 반드시 동적 할당 해주거나 어떤 문자열의 주소를 넣어주어야 한다.

# String Copy?

- `char str1[6] = "Hello";`  
`char str2[6];`  
`str2 = str1; // ?`

# String Input/Output Functions

formatted input/output functions

- scanf/fscanf
- printf/fprintf

special set of string-only functions

- get string (gets/fgets)
- put string ( puts/fputs ).

# FLUSH

- 스트림 버퍼를 비우는 역할을 한다.
- Scanf 등 입력받는 함수 사용 시  
→ The string conversion code(s) skips whitespace.

Test)

```
int a; char b;  
scanf("%d", &a);  
scanf("%c",&b);  
printf("%d %c\n", a, b);
```

```
1 { // Read Month  
2   #define FLUSH while (getchar() != '\n')  
3   char month[10];  
4  
5   printf("Please enter a month. ");  
6   scanf("%9s", month);  
7   FLUSH;  
8 }
```

```
fflush();
```

```
#define FLUSH while(getchar != '\n')
```

# Formatted String Input

- `char str[10];`
- `scanf("%9s", str);`
- `str` - 배열 포인터 → `&`가 붙지 않음
  
- 입력 문자 개수를 반드시 정의해 주자

# String Manipulation Functions

- `#include <string.h>`
- String Length and String Copy
- String Compare and String Concatenate
- Character in String
- Search for a Substring and Search for Character in Set
- String Span and String Token
- String to Number

# More at

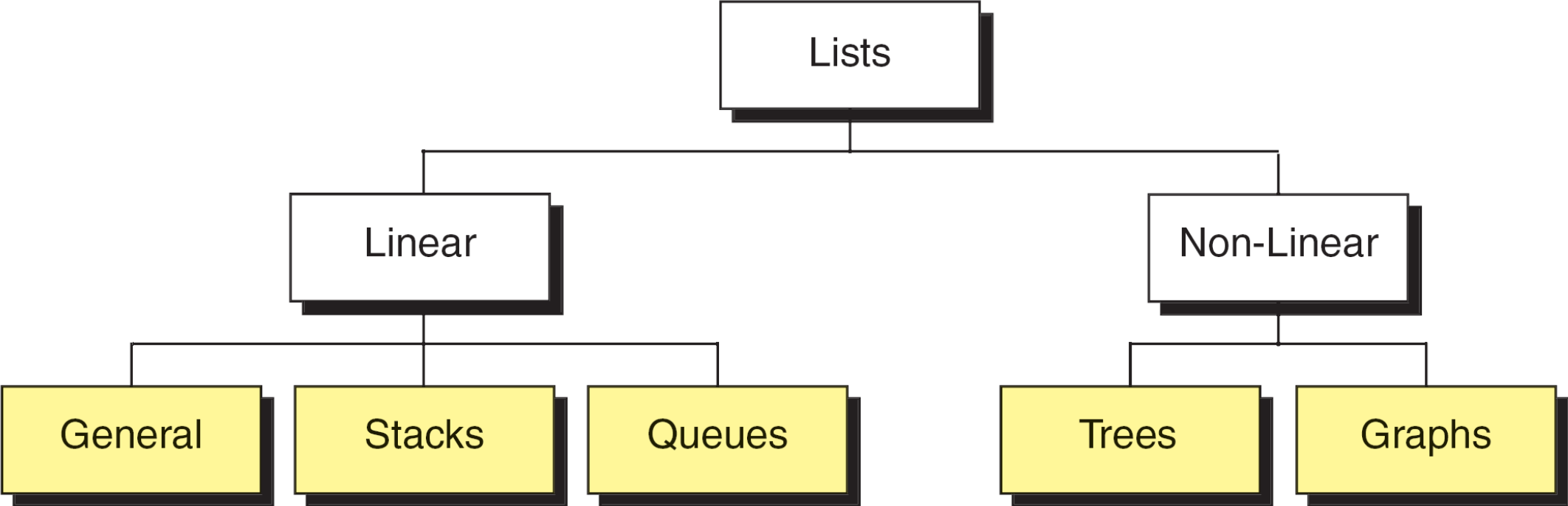
- <http://www.cplusplus.com/reference/string/>
- 수업시간에 한 것들은 한 번씩 찾아 보고 사용법 익히기

# 오늘 할 것

- Lists – various Data Structures implemented with List
- Assn #5 질문받기

**List**

# What to learn...



# Linked List

- 각각의 Element가 다음 Element의 위치를 저장하여 이어진 데이터 스트럭처
- 연속으로 쭉 이어졌다면 왜 배열(Array)를 안쓸까??
  - Array의 장점? 단점?
- Linked List의 장점? 단점?

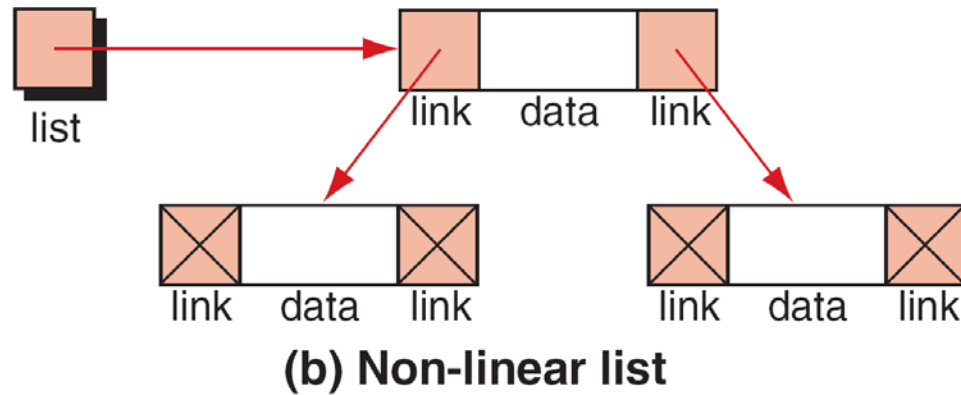
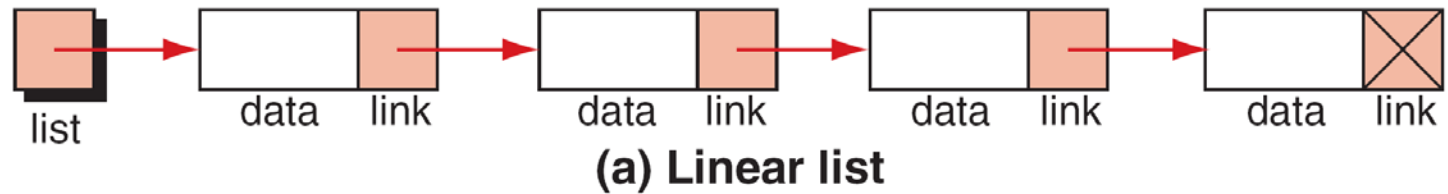
	List	Linked List
Space Complexity	$O(n)$	
At	$O(1)$	$O(n)$
Insert	$O(n)$	$O(1)$
Remove	$O(n)$	$O(1)$

# Array – List Pros and Cons

항목	연결 리스트	배열
임의의 위치 접근	어려움	한번에 가능
임의의 위치 삽입/삭제	쉬움	어려움
가변 개수 데이터 지원	쉬움	최대 수용 가능 개수 이하로만 쉬움
데이터 정렬	어려움	쉬움
기억 공간 차지	(데이터의 크기 + 포인터의 크기 + $\alpha$ ) * 연결 개수	최대 수용 가능 개수 * (데이터 당 크기 + $\beta$ )
사용하지 않는 기억 공간으로 인한 낭비	없음	최대 수용 가능 개수 - 실제로 사용하는 데이터 수
초기화 후 유지/관리	포인터 사용으로 인해 까다로움	배열의 인덱스 값만 조절하면 되므로 쉬움
특정 값을 가지는 데이터 찾기	정렬 여부에 관계없이 어려움	정렬된 배열에서는 쉬움

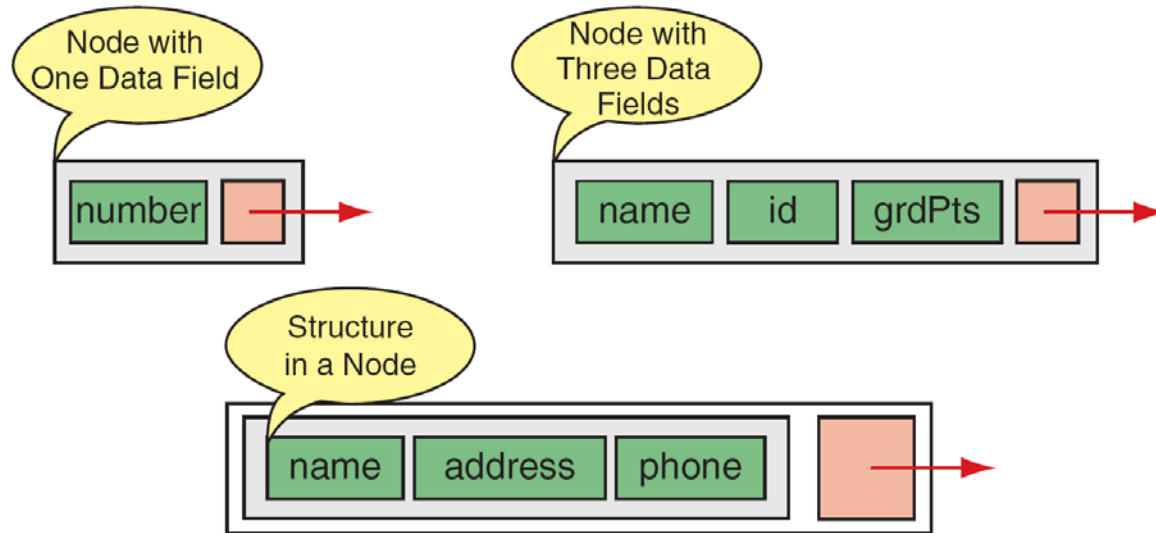
# Linked List

- 한개만 가리킬 수도 있고, 여러개를 가리킬 수도 있고...



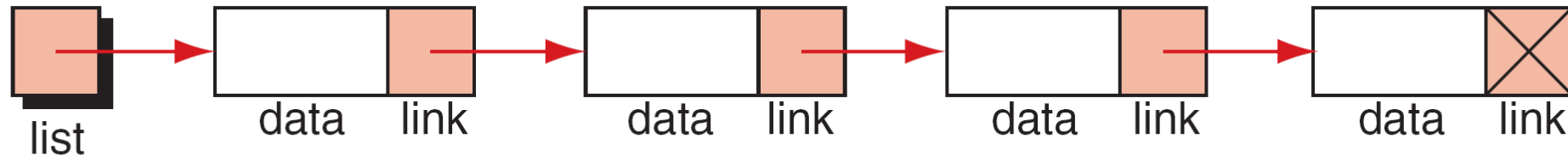
# Linked List Node Structures

- 데이터 부분 + 포인터 부분(다른 Element를 가리킴)



# General Linear Lists

- 가장 일반적인 직렬 리스트
  - Retrieve
  - Insert
  - Change
  - Delete
  - Traversing
  - Building

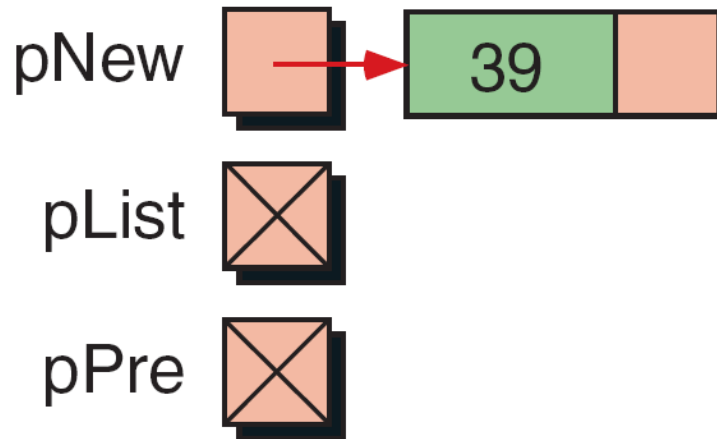


(a) Linear list

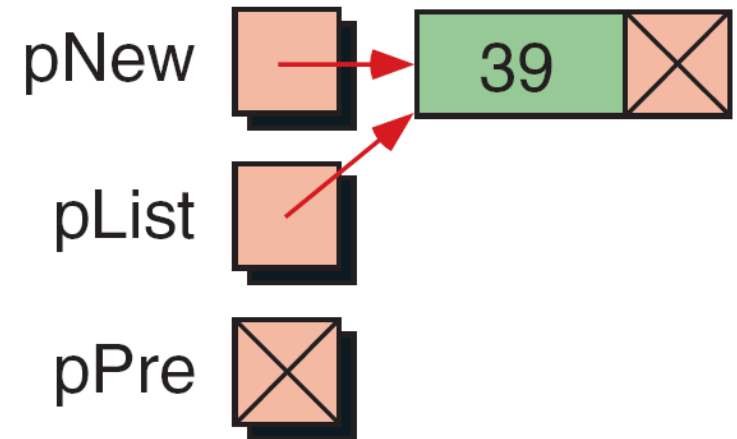
# Insert a Node to Empty List

```
pNew->link = pList ;  
pList      = pNew ;
```

Before Add

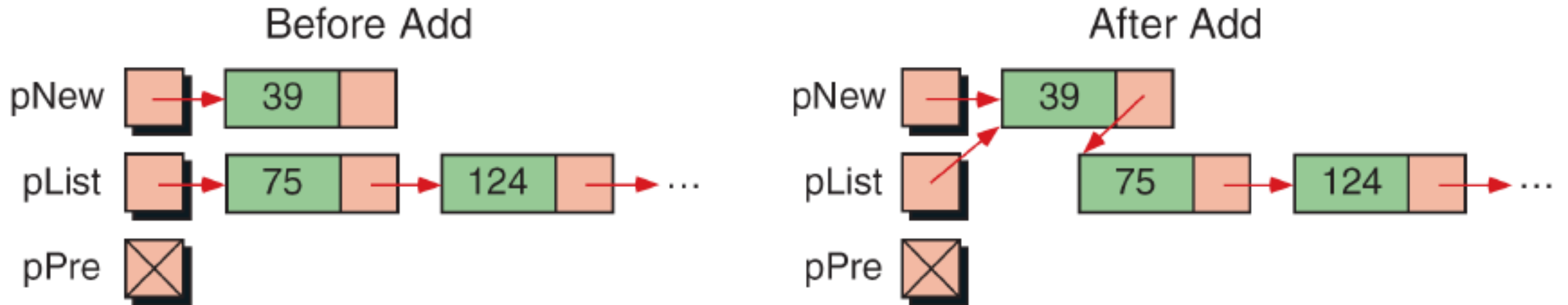


After Add



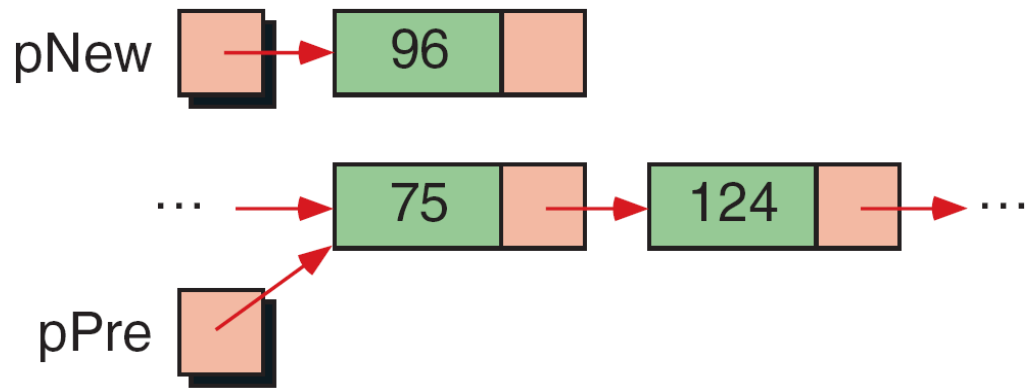
# Insert a Node at Beginning

```
pNew->link = pList;  
pList = pNew;
```

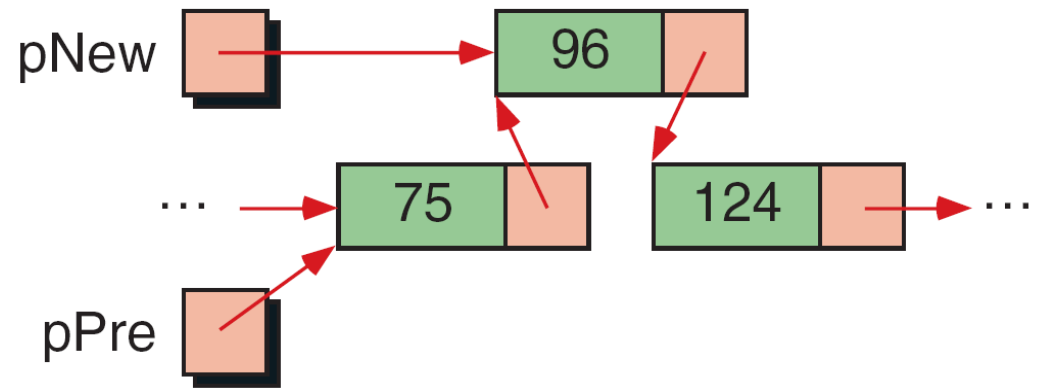


# Insert a Node in Middle

```
pNew->link = pPre->link;  
pPre->link = pNew;
```



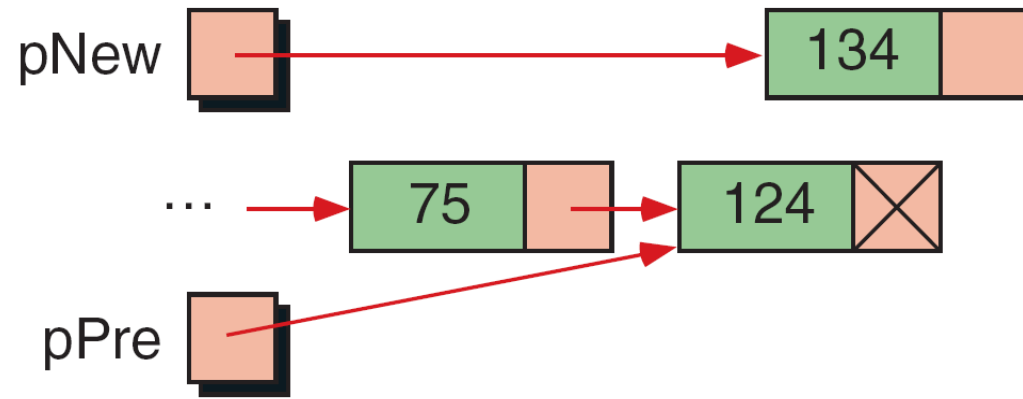
Before Add



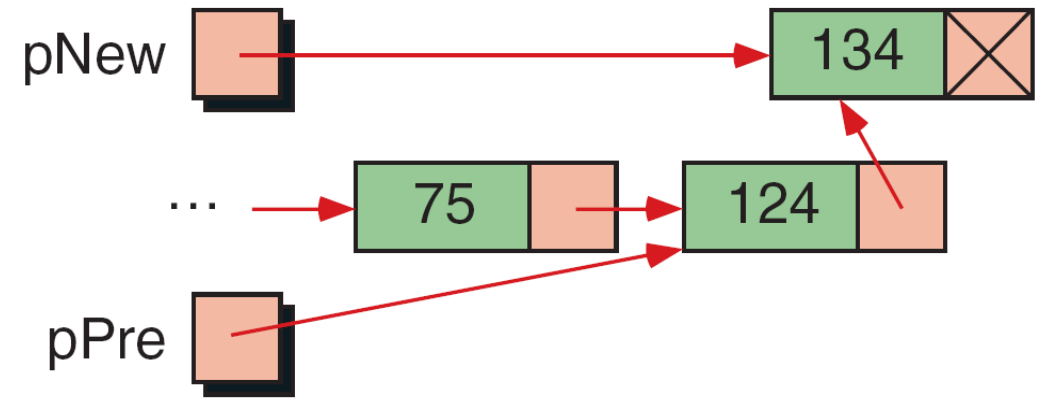
After Add

# Insert a Node at End

```
pNew->link = pPre->link;  
pPre->link = pNew;
```



Before Add

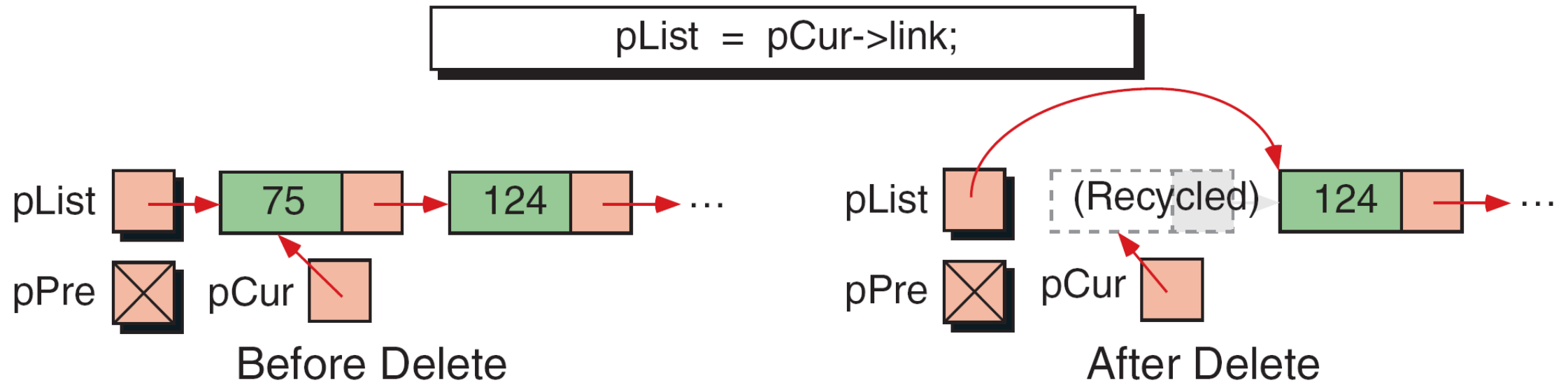


After Add

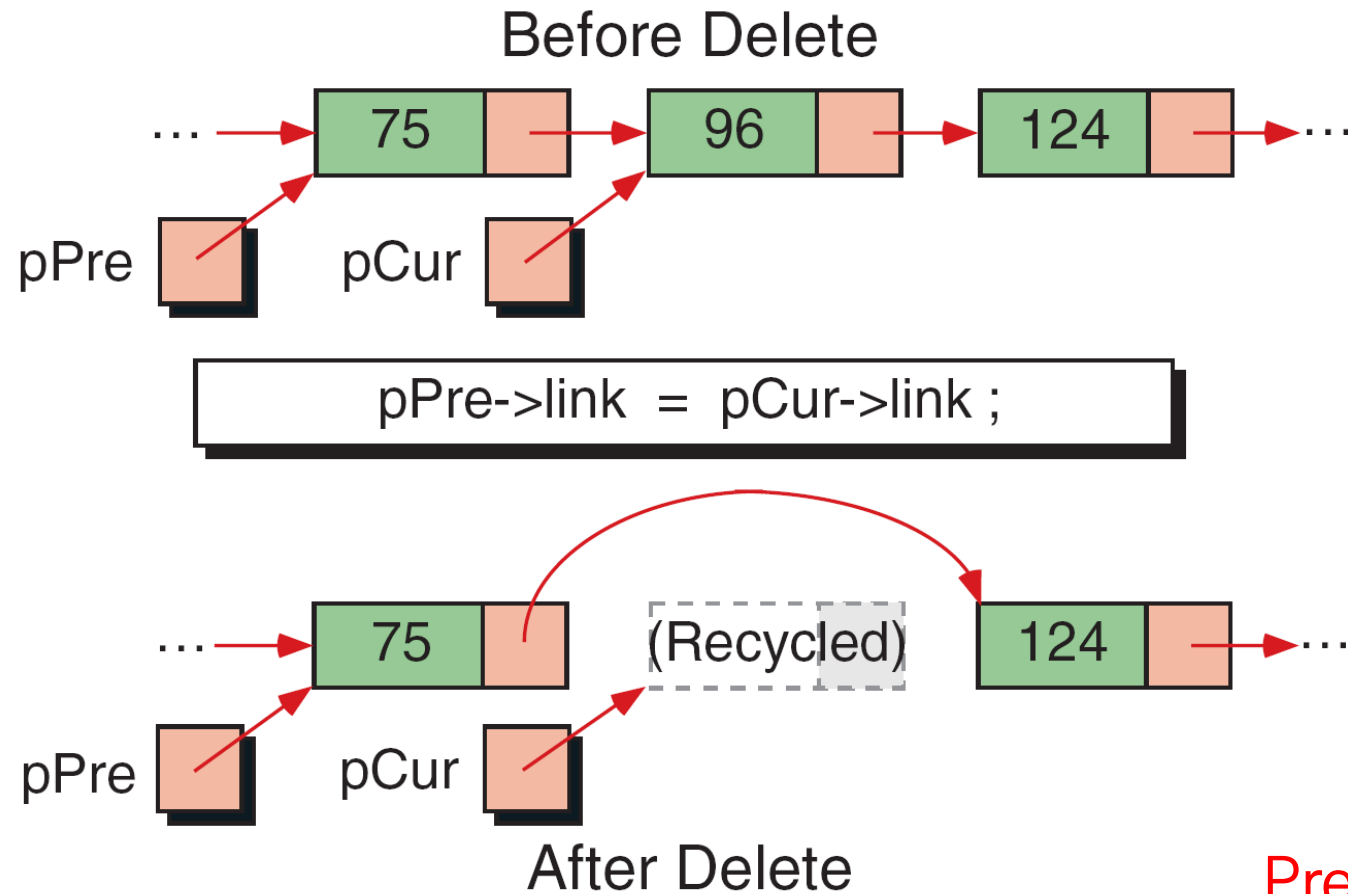
# Insert a Node – Code

```
 8  NODE* insertNode (NODE* pList, NODE* pPre, DATA item)
 9  {
10  // Local Declarations
11  NODE* pNew;
12
13  // Statements
14  if (!(pNew = (NODE*)malloc(sizeof(NODE))))
15      printf("\aMemory overflow in insert\n"),
16          exit (100);
17
18  pNew->data = item;
19  if (pPre == NULL)
20  {
21      // Inserting before first node or to empty list
22      pNew->link = pList;
23      pList     = pNew;
24  } // if pPre
25  else
26  {
27      // Inserting in middle or at end
28      pNew->link = pPre->link;
29      pPre->link = pNew;
30  } // else
31  return pList;
32 } // insertNode
```

# Delete First Node



# Delete – General Case



Pre, Cur 두 개가  
같이 움직이는 거 주의!!

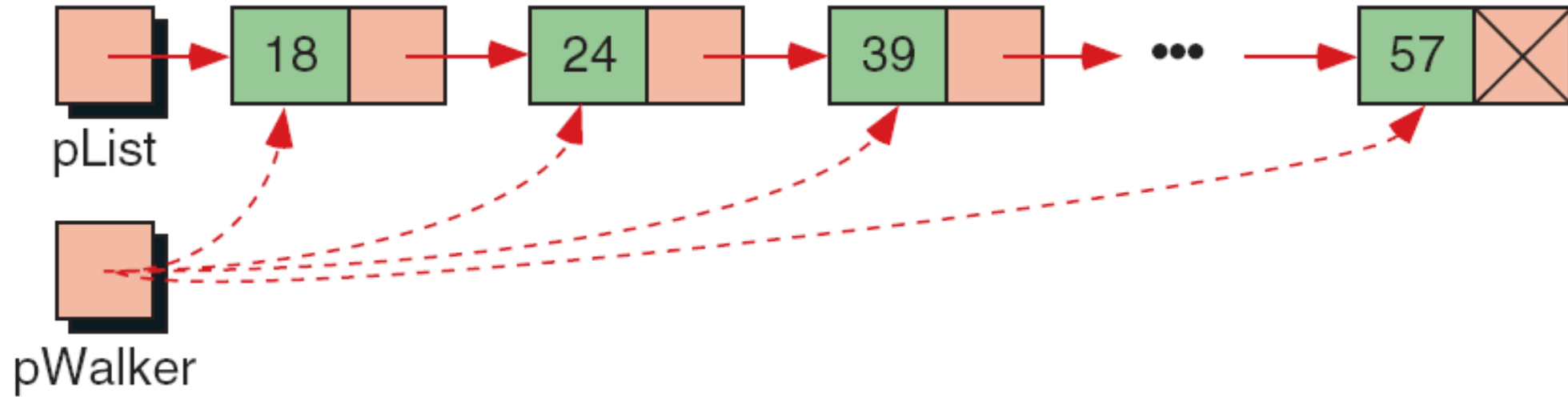
# Delete a Node – Code

```
1  /* ===== deleteNode =====
2     This function deletes a single node from the link list.
3     Pre   pList is a pointer to the head of the list
4           pPre points to node before the delete node
5           pCur points to the node to be deleted
6     Post  deletes and recycles pCur
7           returns the head pointer
8  */
9  NODE* deleteNode (NODE* pList, NODE* pPre, NODE* pCur)
10 {
11 // Statements
12     if (pPre == NULL)
13         // Deleting first node
14         pList = pCur->link;
15     else
16         // Deleting other nodes
17         pPre->link = pCur->link;
18     free (pCur);
19     return pList;
20 }
```

# Search Linear List

```
//리스트 pList가 있다.  
struct Node *pWalker = pList;  
while(pWalker)  
{  
    If(pWalker->data == 찾는 데이터)  
        return 1;  
    pWalker = pWalker->next;  
}  
return 0;
```

# Linear List Traversal



# Print Linear List

```
5 void printList (NODE* pList)
6 {
7 // Local Declarations
8     NODE* pWalker;
9
10 // Statements
11     pWalker = pList;
12     printf("List contains:\n");
13
14     while (pWalker)
15     {
16         printf("%3d ", pWalker->data.key);
17         pWalker = pWalker->link;
18     } // while
19     printf("\n");
20     return;
```

# Average Linear List

```
5 double averageList (NODE* pList)
6 {
7 // Local Declarations
8     NODE* pWalker;
9     int total;
10    int count;
11
12 // Statements
13    total = count = 0;
14    pWalker = pList;
15    while (pWalker)
16    {
17        total += pWalker->data.key;
18        count++;
19        pWalker = pWalker->link;
20    } // while
21    return (double)total / count;
22 } // averageList
```

# Stack

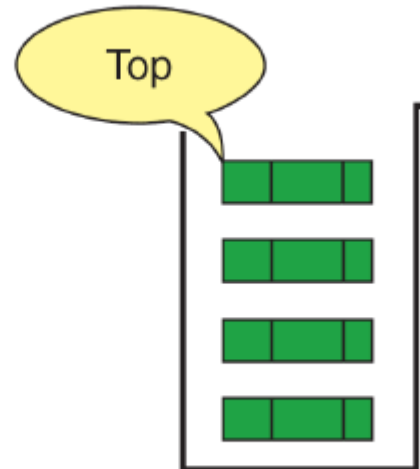
- Last in First out (LIFO)의 데이터 스트럭처
- Insertion과 deletion이 항상 한쪽 끝(Top)에서만 일어난다.
- Push : 새로운 것을 넣는 작업
- Pop : 젤 위에서 하나 빼는 작업



Stack of Coins



Stack of Books

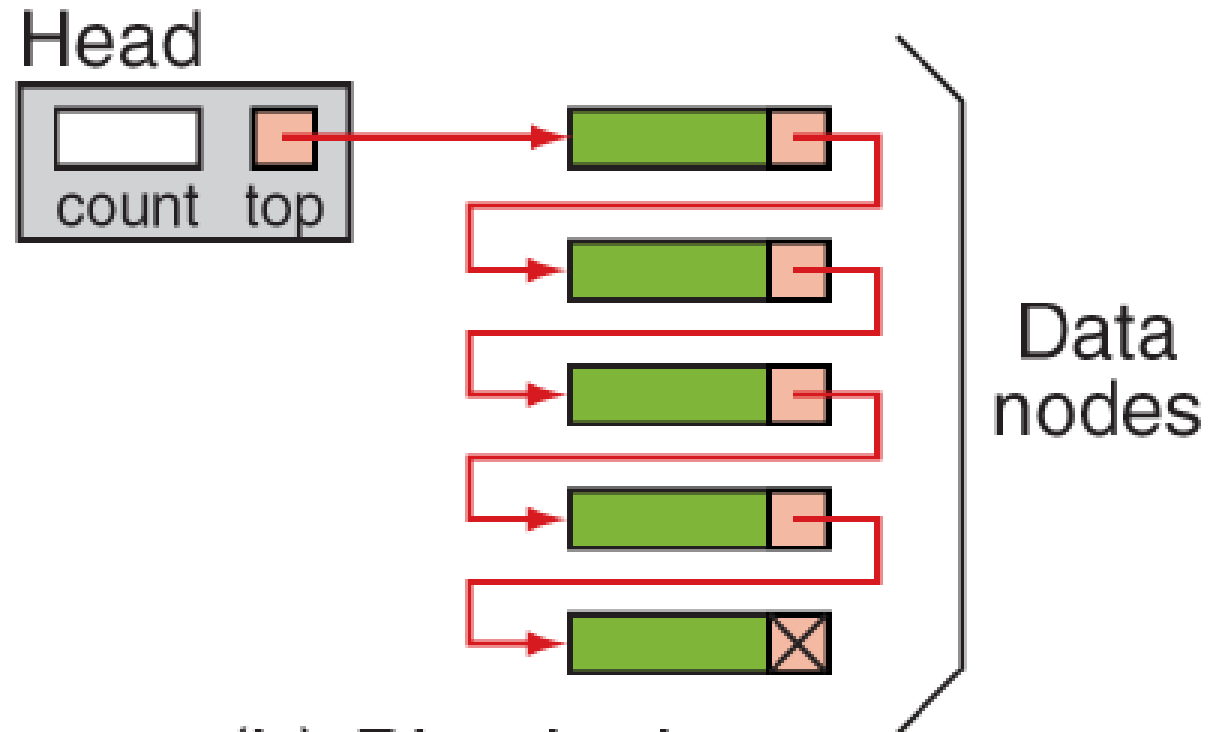


Computer Stack

# Stack Implementations

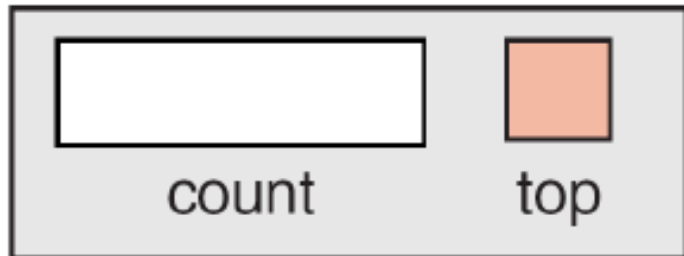


(a) Conceptual

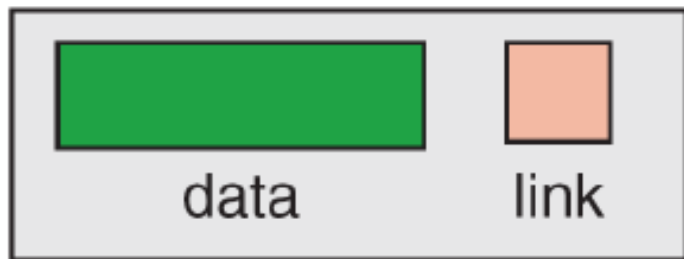


(b) Physical

# Stack Data Structure



Stack Head Structure

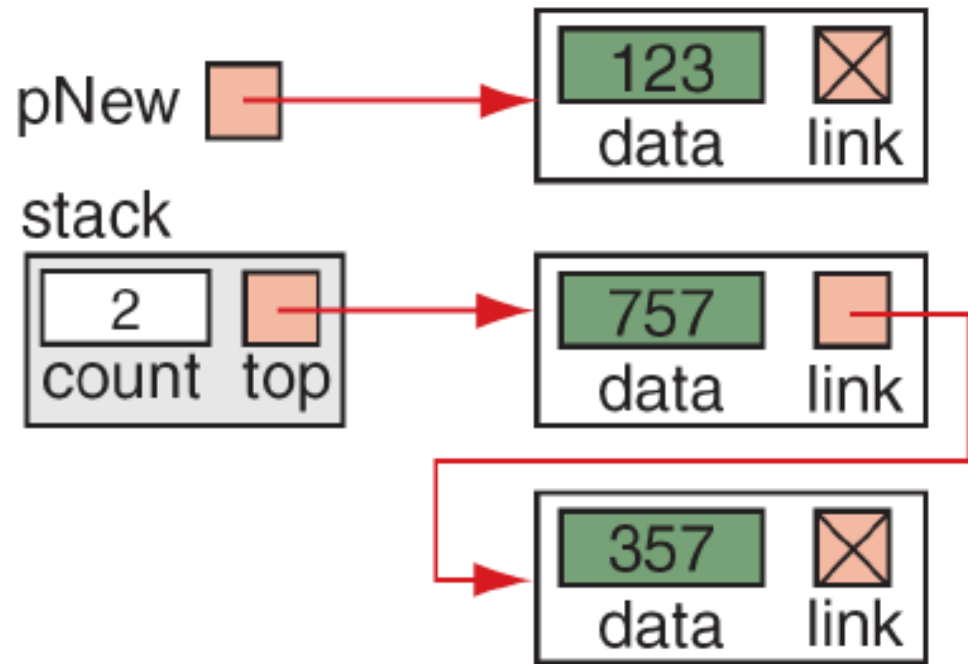


Stack Node Structure

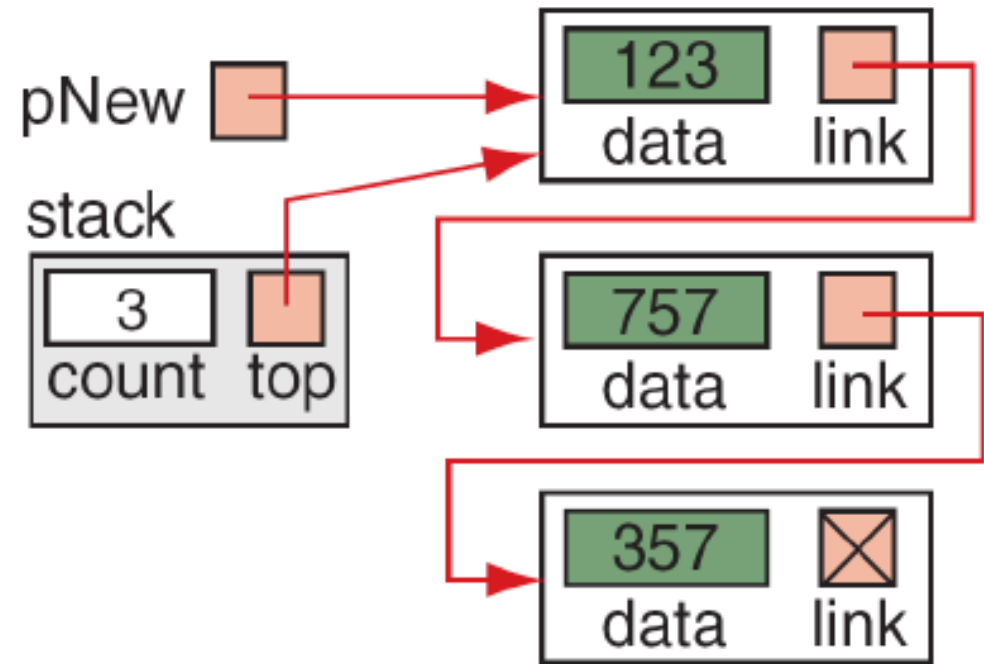
```
typedef struct
{
    int          count;
    struct node* top;
} STACK;

typedef struct node
{
    int          data;
    struct node* link;
} STACK_NODE;
```

# Stack\_Push

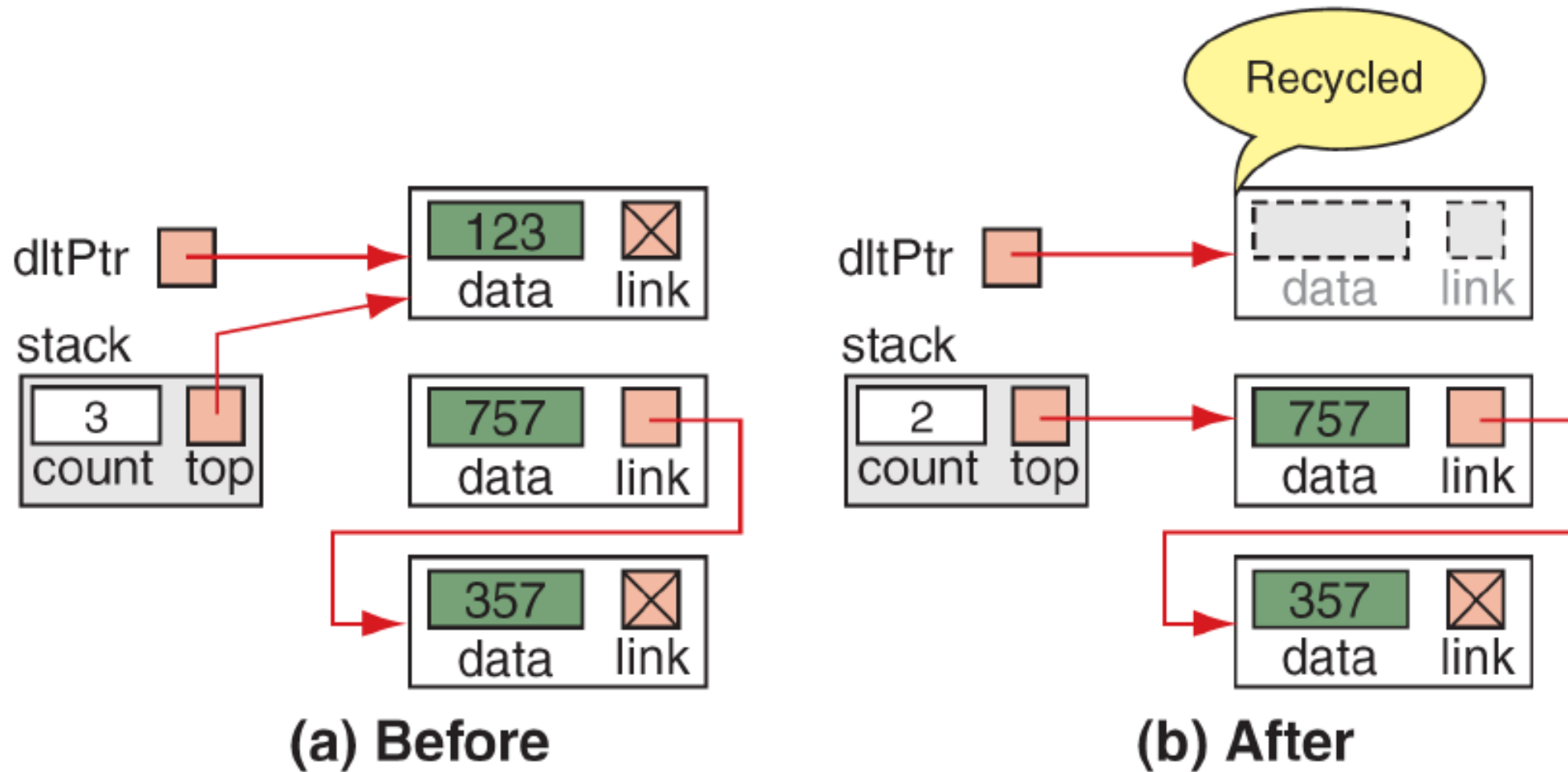


(a) Before



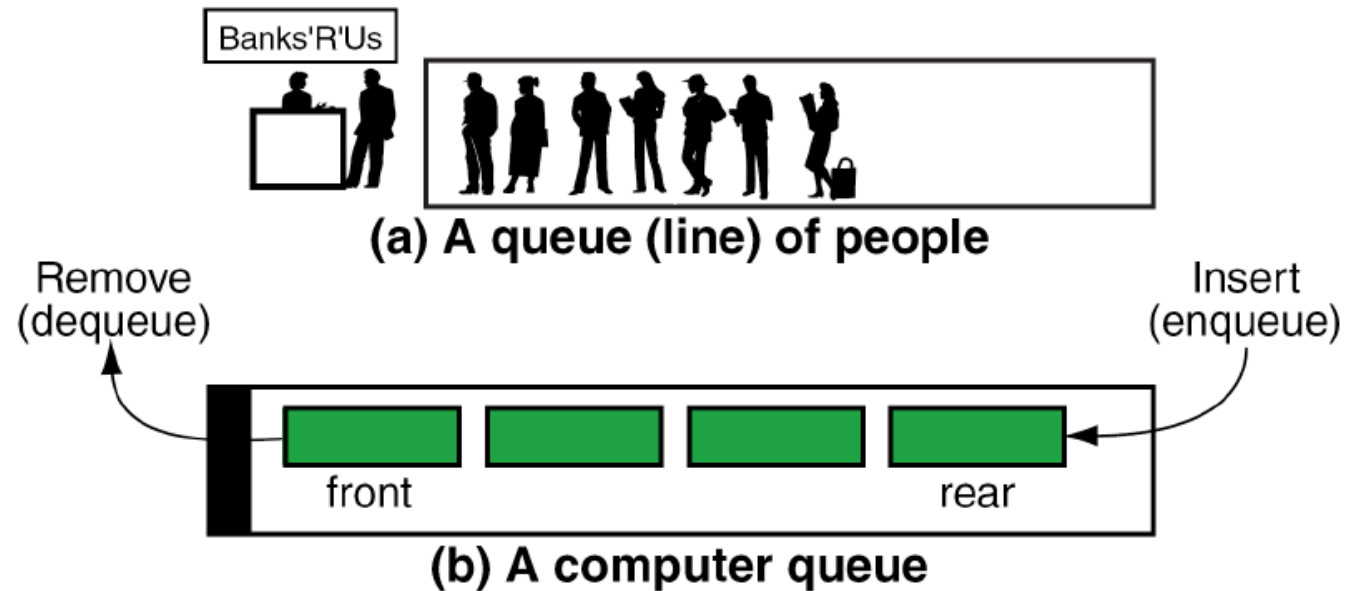
(b) After

# Stack\_Pop

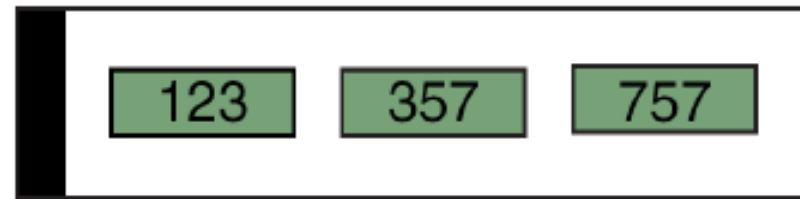


# Queue

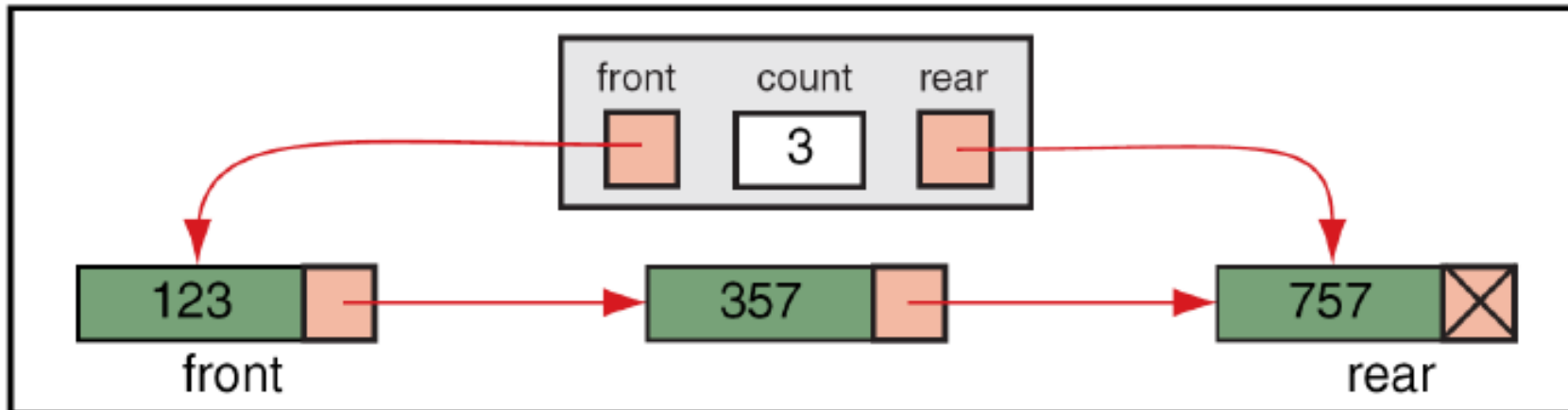
- **First in First out(FIFO)**의 데이터 스트럭처
- Insertion은 한쪽 끝, Deletion은 **다른** 한쪽 끝에서만 일어난다. :
- Enqueue : 큐에 새로운 걸 넣는 작업 (rear)
- Dequeue : 큐에서 하나를 제거하는 작업 (front)



# Queue Implementations



(a) Conceptual queue

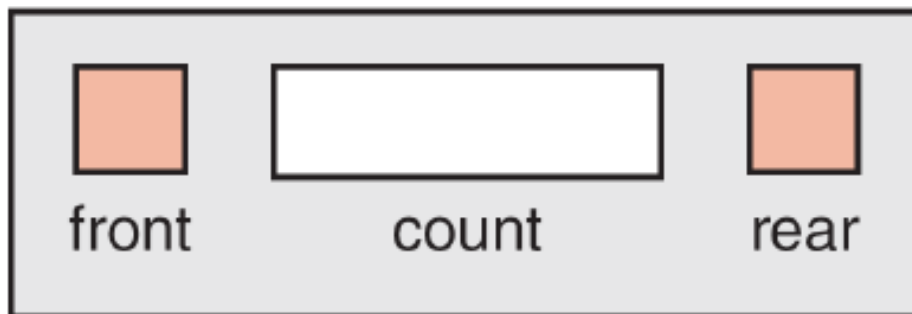


(b) Physical queue

# Queue Data Structure



**Node Structure**

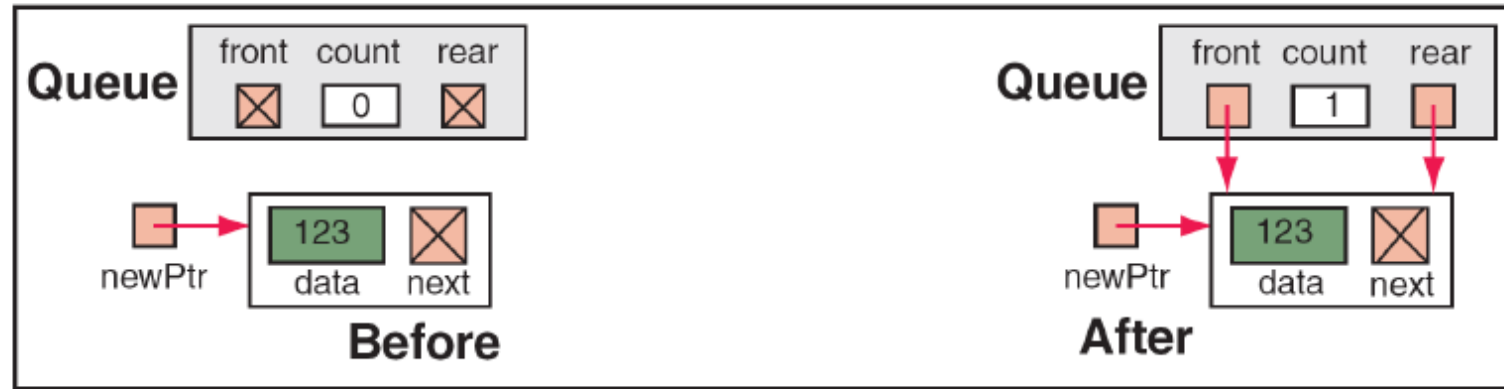


**Head Structure**

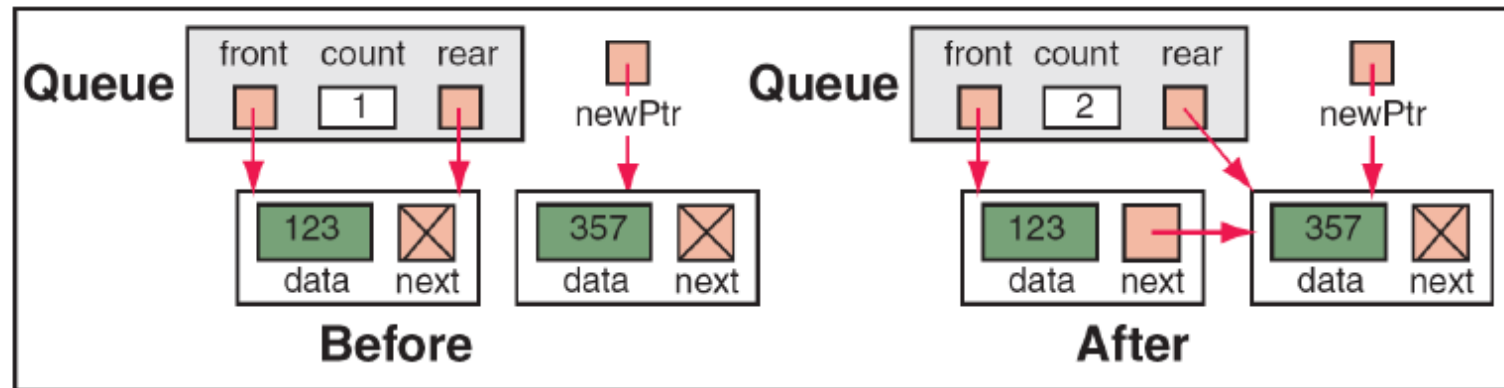
```
typedef struct node
{
    int          data;
    struct node* next;
} QUEUE_NODE;
```

```
typedef struct
{
    QUEUE_NODE* front;
    int          count;
    QUEUE_NODE* rear;
} QUEUE;
```

# Enqueue

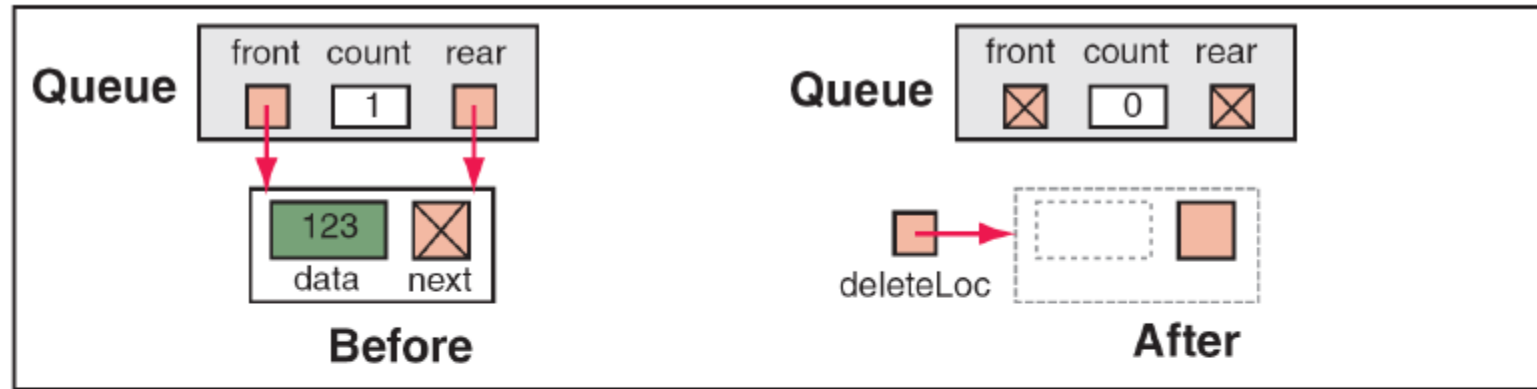


(a) Case 1: Insert into Null Queue

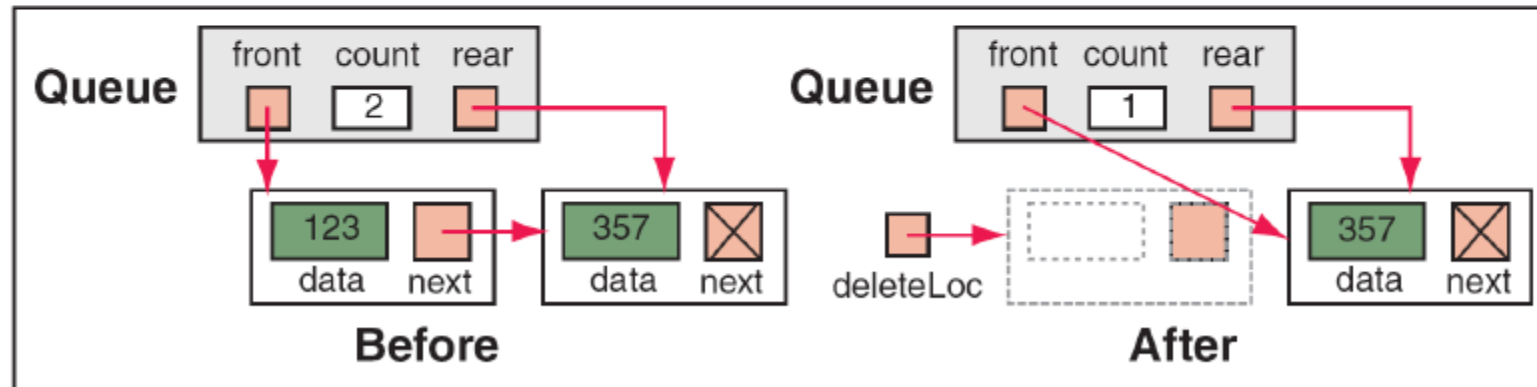


(b) Case 2: Insert into Queue with Data

# Dequeue



**(a) Case 1: Delete only item in queue**



**(b) Case 2: Delete item at front of queue**

# 오늘 배운 것 정리

- 책에 사이사이 박스에 있는 예제 코드들 **전부 다** 한번씩 쪽 보고 이해하기. 코드가 있는 거는 충분히 시험에 코드 짜는 문제로 나올 수 있다!
- Data Structure에서는 다양한 Implementation이 있기 때문에 무작정 외우기 보다는 어떻게 구현하는지 Concept들을 꼭 기억
- DS부분은 코드 반드시 봐야 합니다.
- 빨간색 글씨로 된 거 기억하기

# 다음시간

- 기말고사 대비 총정리
- Assn #5 질문받기