

전산 SMP 11주차

2015. 12. 08

김범수

bskim45@gmail.com

Special thanks to 박기석 (kisuk0521@gmail.com)

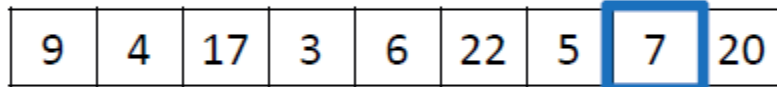
기말고사 대비 총정리

기말고사 화이팅

자료구조

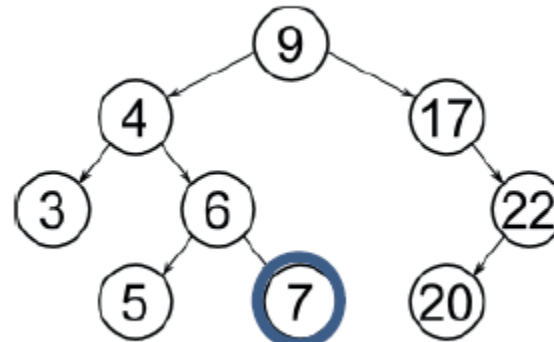
- 많은 데이터를 예쁘게 잘 저장해서
 - 쉽고 빠르게 꺼내 쓰려고
 - 데이터가 많아지면 찾는 데도 오래 걸린다.
-
- Example: finding a number from a set of numbers
 - How many comparisons do we need to retrieve 7?

In linear array



8 comparisons

In binary search tree

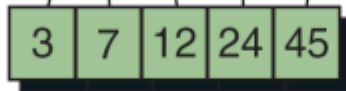


4 comparisons

Array

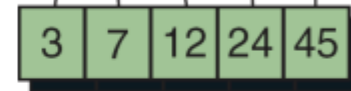
(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```



(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



(c) Partial Initialization

```
int numbers[5] = {3,7};
```



The rest are filled with 0s

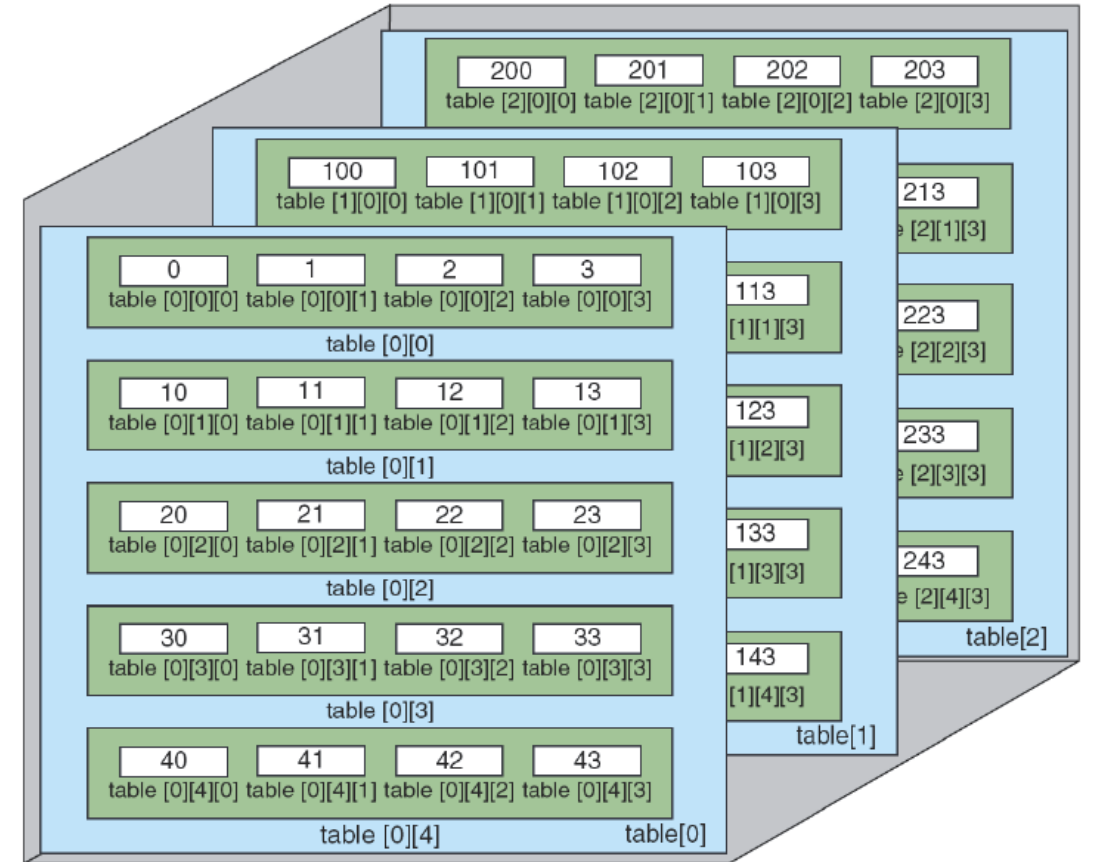
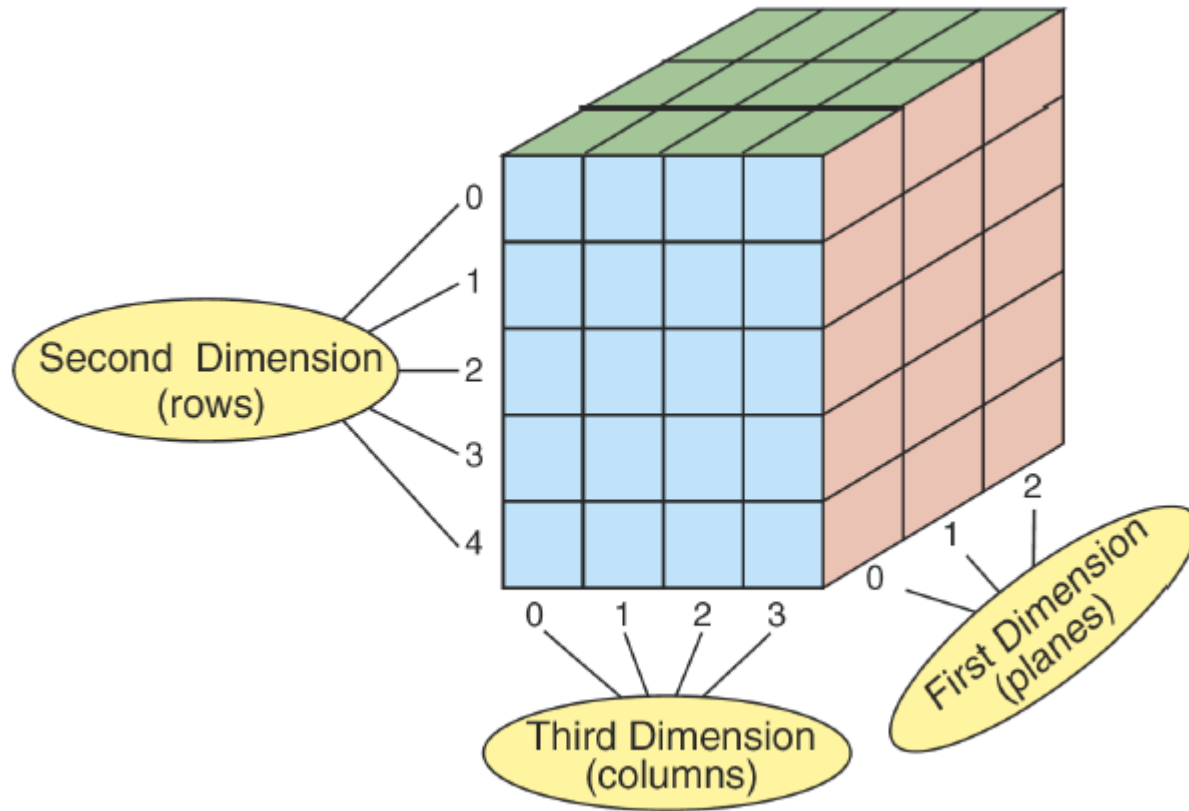
(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```



All filled with 0s

Multi-dimension array



Sorting

웬 갑자기 Sorting

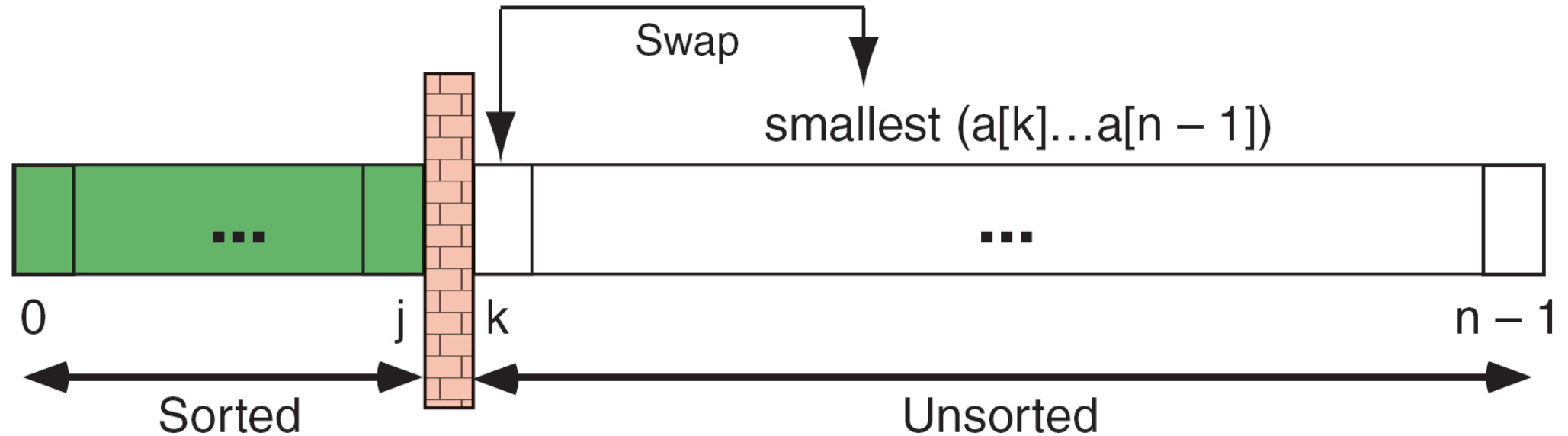
- Array는 자료구조
- 우리가 원하는 데이터를 Array에서 빠르게 찾으려면?
- 저장할 때 잘 저장해서 꺼내 쓸 때 편해야 한다

- 데이터를 효율적으로 관리하는 방법
 - 데이터를 저장하는 자료구조 + 데이터를 찾는 알고리즘

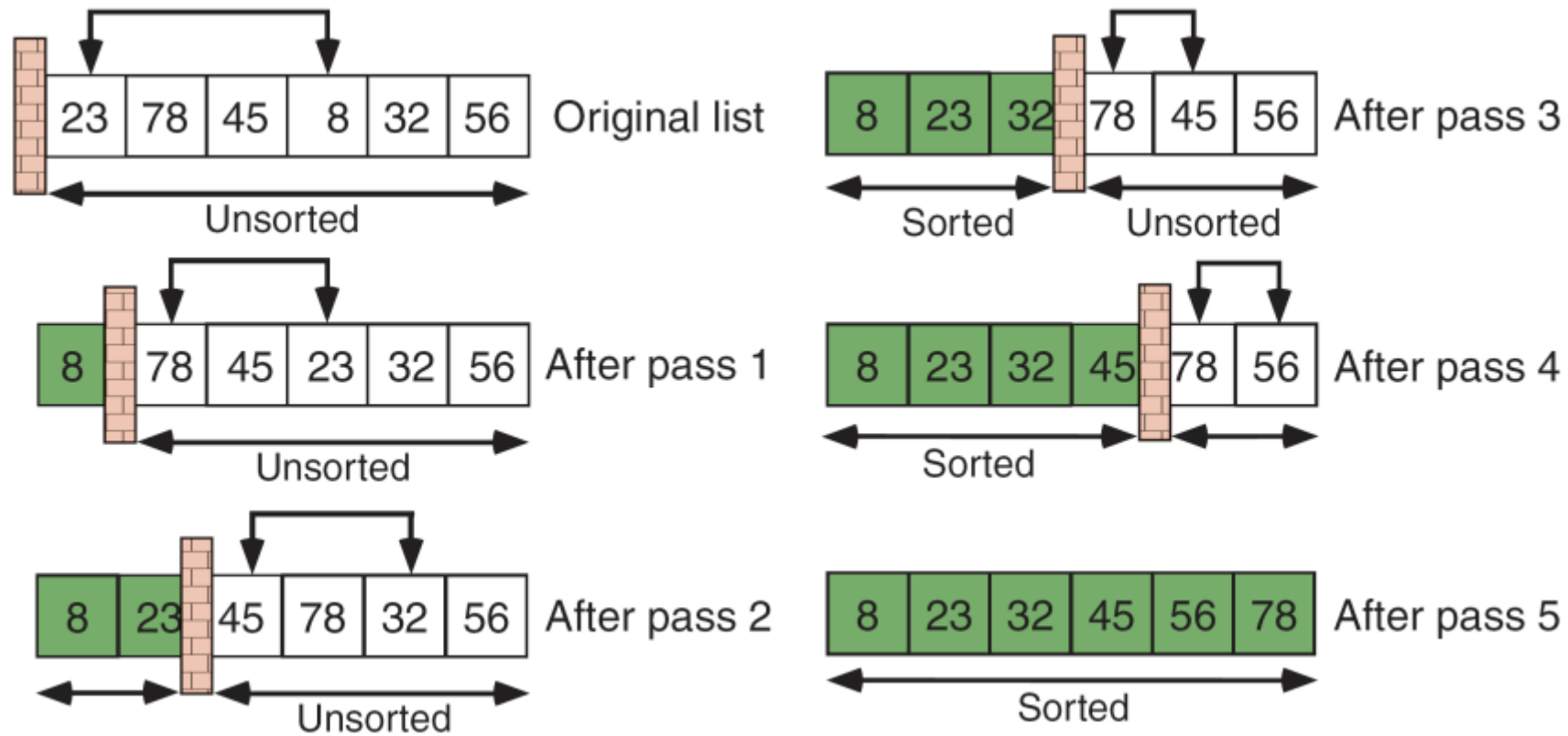
Array Sorting

- Selection Sort
- Bubble Sort
- Insertion Sort

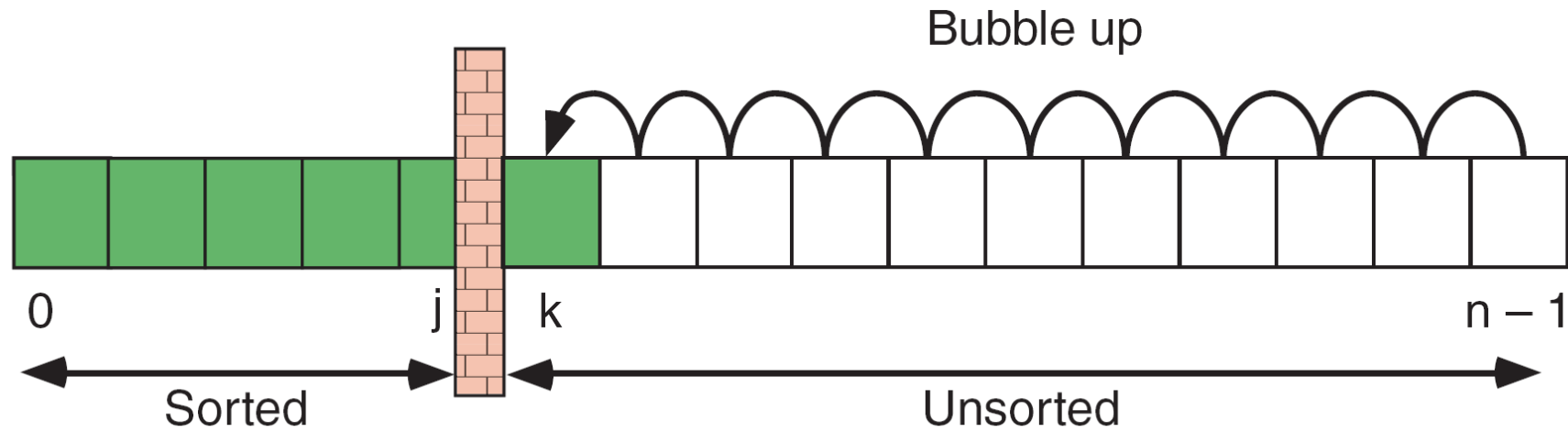
Selection Sort Concept



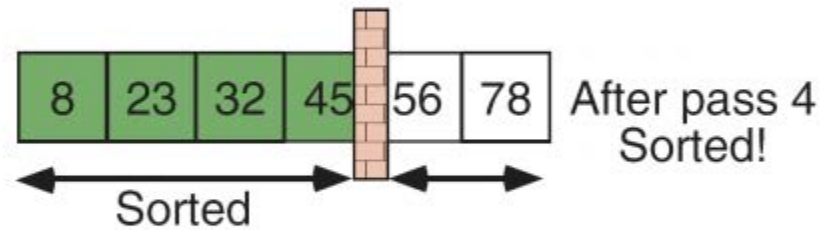
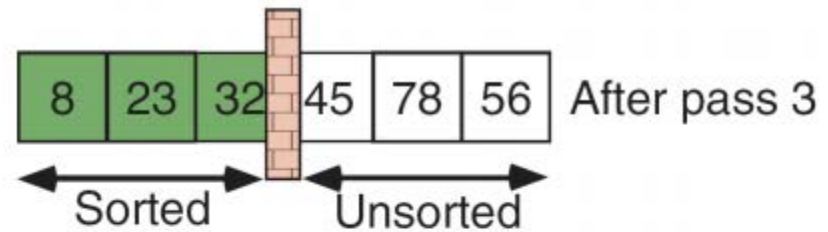
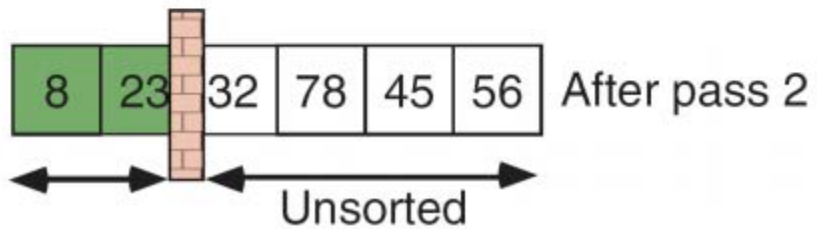
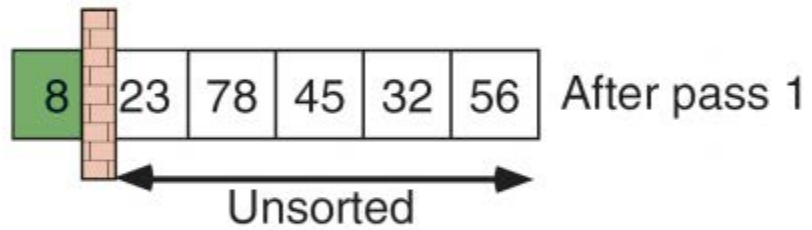
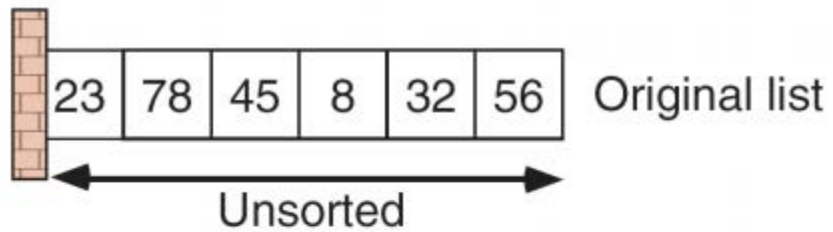
Selection Sort



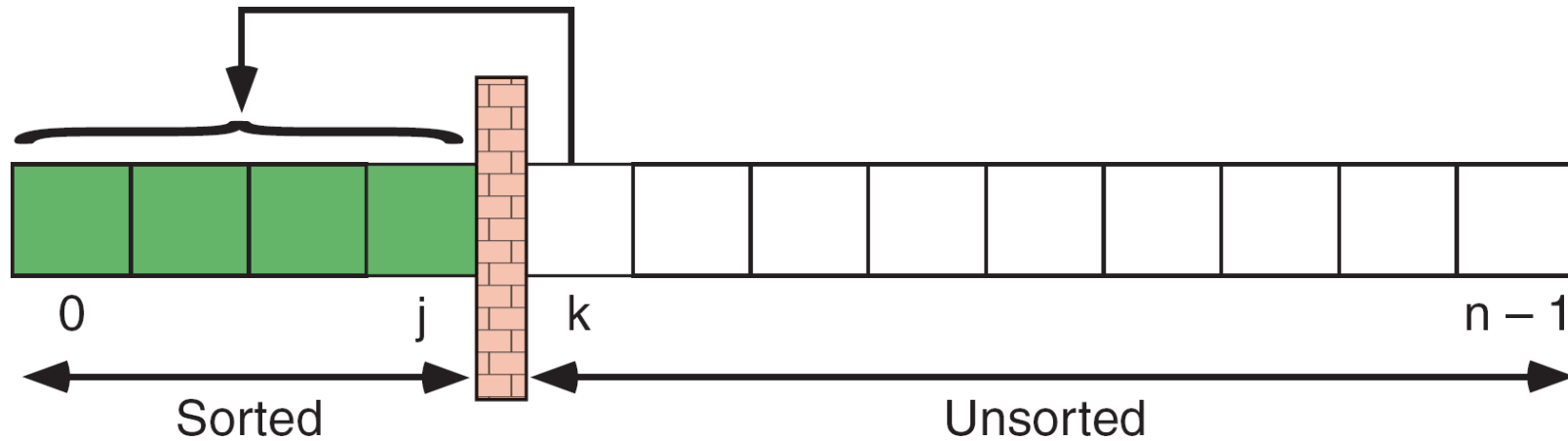
Bubble Sort Concept



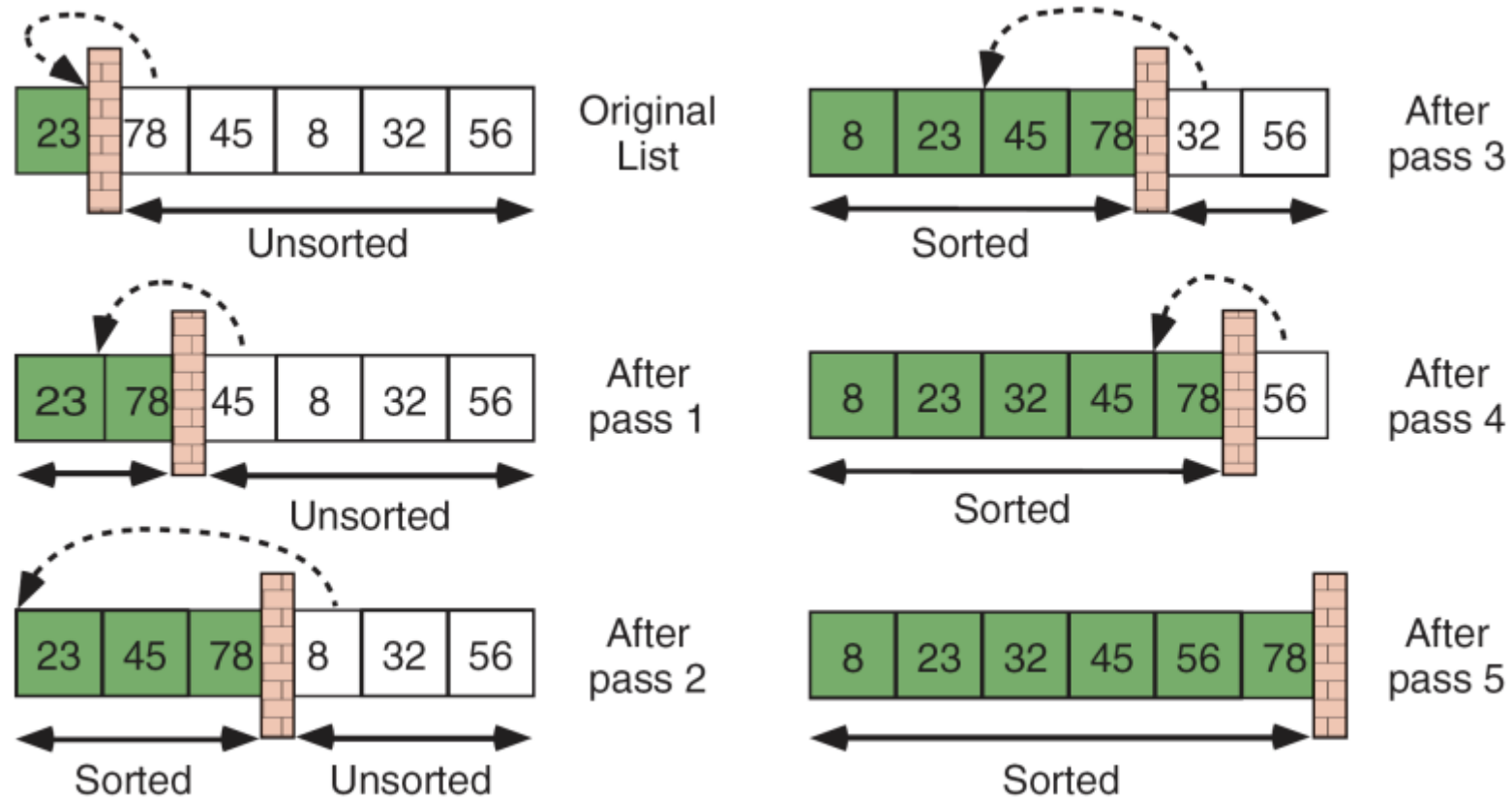
Bubble Sort Example



Insertion Sort Concept



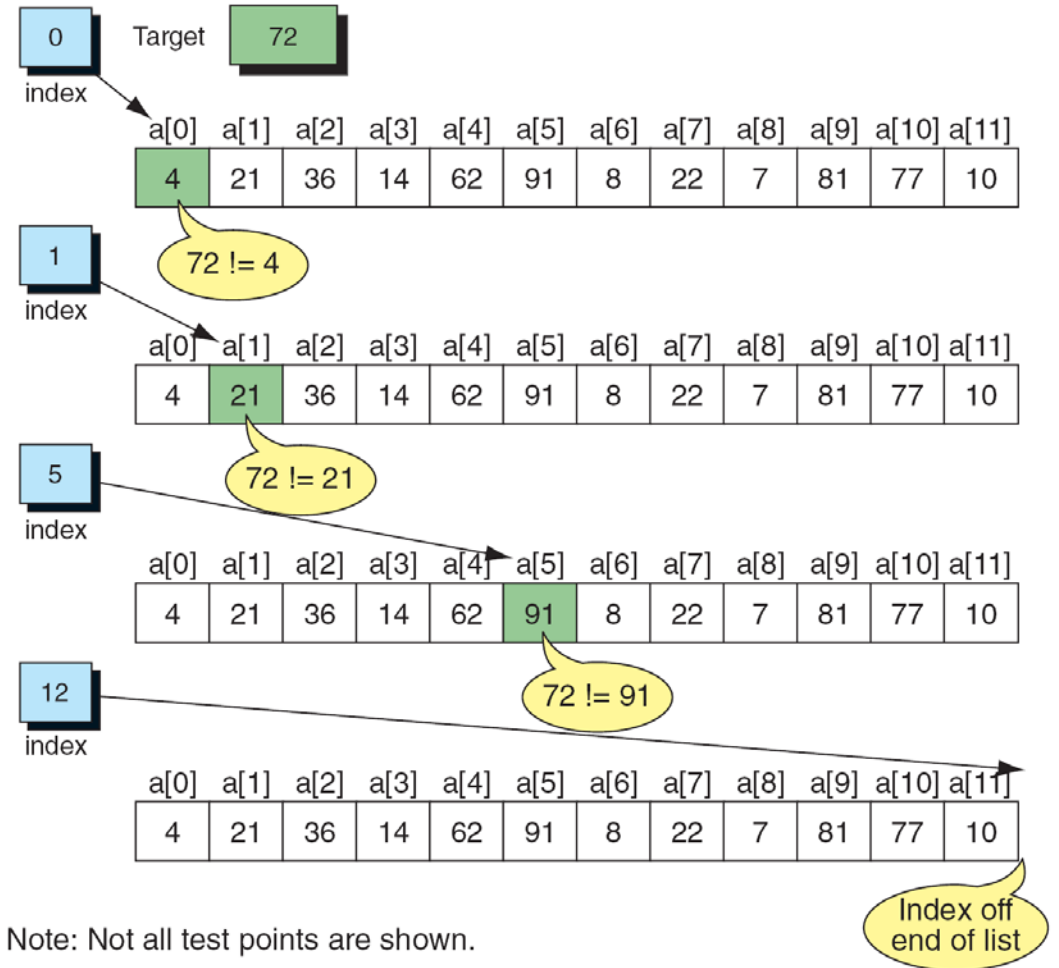
Insertion Sort Example



Search

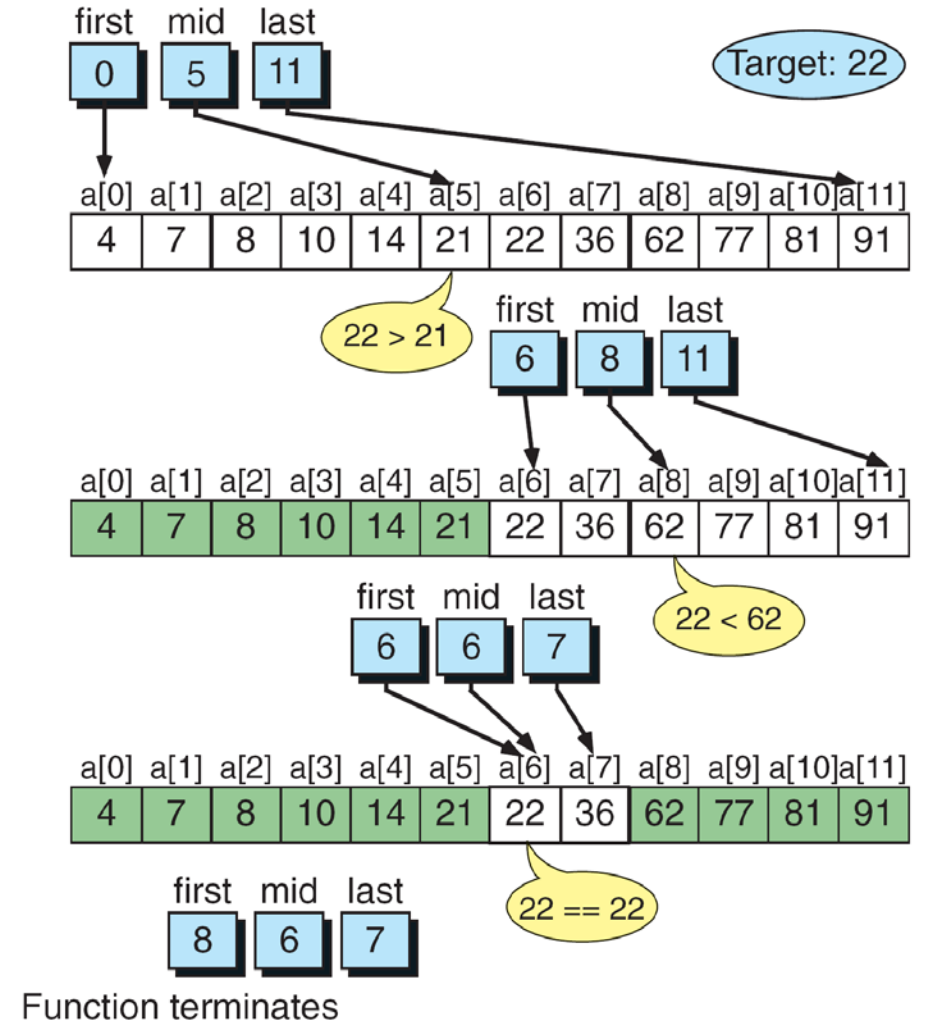
Sequential Search 특징

- Unsorted 배열에 대해서는 처음부터 꼭 봐야하기 때문에 비효율적이다.
- Worst Case: $O(n)$



Binary Search 특징

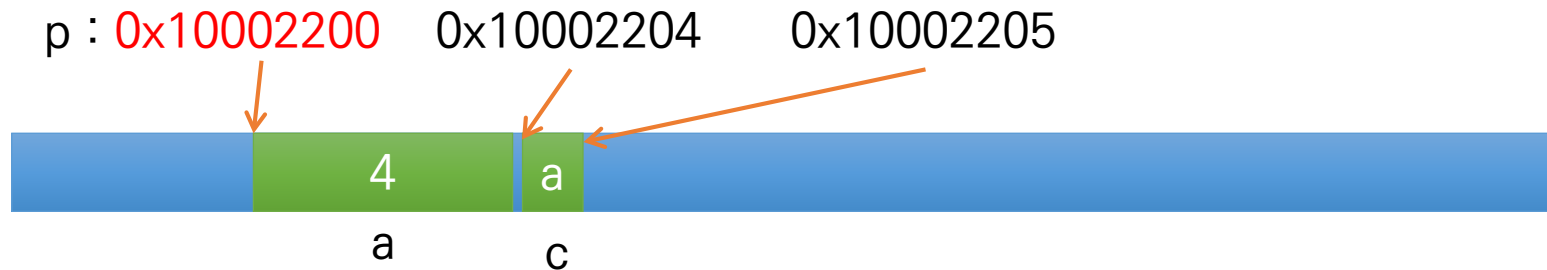
- 검색 대상 배열은 **sorted** 인 상태여야 한다.
 - “정렬된 데이터에 대해서는 이진검색이 빠르다”
- 매 iteration에서 반쯤 (1/2)을 서치 후보에서 제외시킨다. (제외된 거는 볼 필요 없음)
- $O(\log n)$



Pointer

Pointer란?

- 데이터 접근에 사용되는 **주소**를 저장하는 타입 - 포인터도 타입이다
- `int a = 4; char c = 'a'; int *p = &a;`



- 메모리는 긴 일차원 공간으로 볼 수 있고, 각 byte는 자신만의 주소를 가지고 있다.

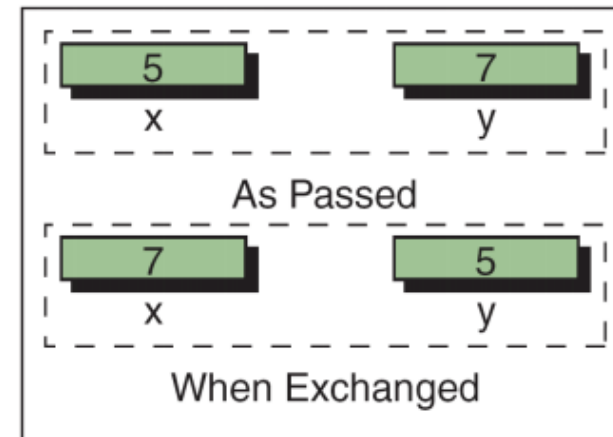
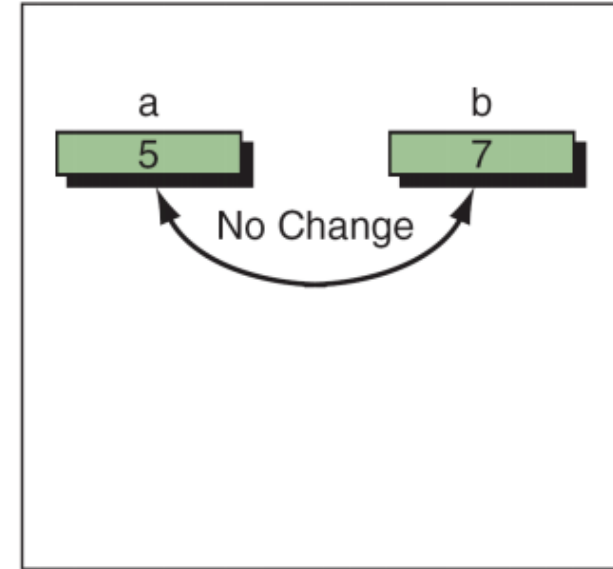
Call-by-value

```
// Function Declarations
void exchange (int x, int y);

int main (void)
{
    int a = 5;
    int b = 7;
    exchange (a, b);
    printf("%d %d\n", a, b);
    return 0;
} // main
```

```
void exchange (int x, int y)
{
    int temp;

    temp = x;
    x     = y;
    y     = temp;
    return;
} // exchange
```



Call-by-reference

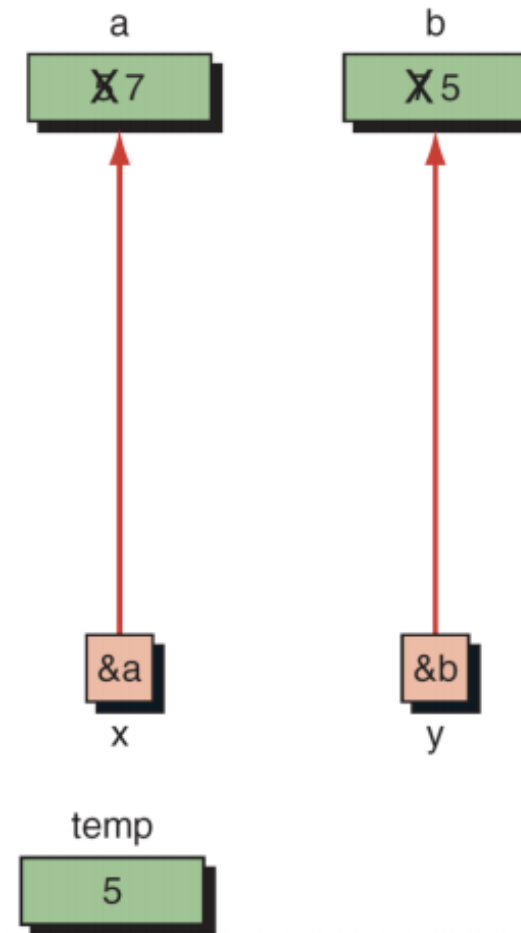
```
// Function Declaration
void exchange (int*, int*);

int main (void)
{
    int a = 5;
    int b = 7;

    exchange (&a, &b);
    printf("%d %d\n", a, b);
    return 0;
} // main
```

```
void exchange (int* px, int* py)
{
    int temp;

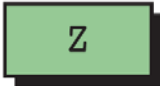
    temp = *px;
    *px = *py;
    *py = temp;
    return;
} // exchange
```



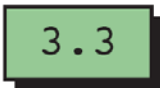
Pointer를 선언할 때 타입이 필요한 이유


```
int *p; char *q;
```


- 어차피 크기는 4byte(32 bit)로 다 똑같은 거 아닌가?
- 컴퓨터가 포인터로 메모리에 접근해 데이터를 읽어올 때!
 - 그 포인터 타입에 따라 가져오는 byte 수가 달라진다.
 - “int * 포인터면 여기서 부터 4byte 까지 읽어서 정수로 쳐”
 - “char * 포인터면 여기서 부터 1byte 만 읽어서 문자로 쳐”


char a; 

int n; 

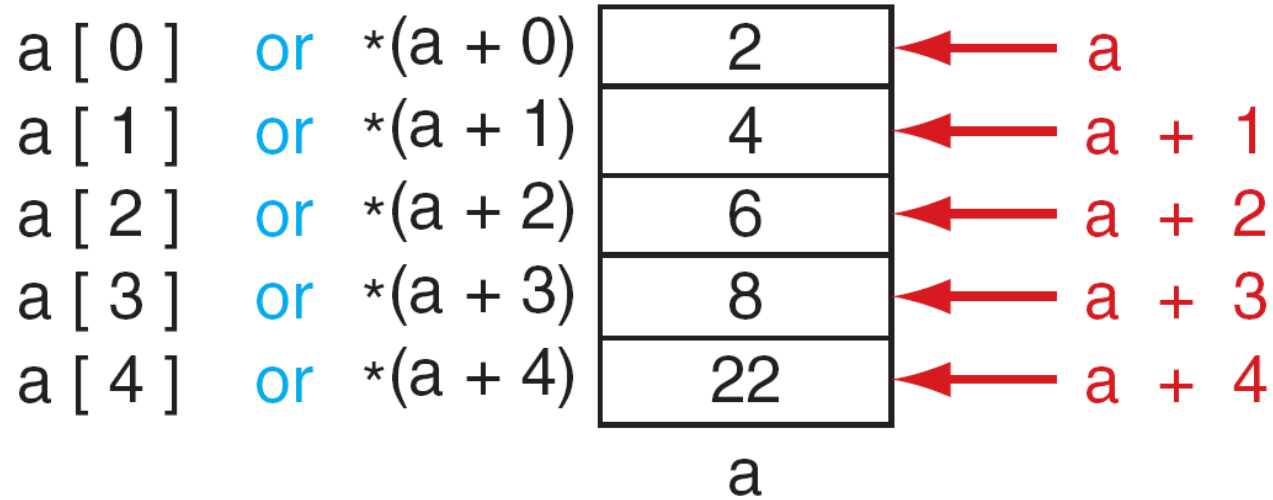
float x; 

char* p; 

int* q; 

float* r; 

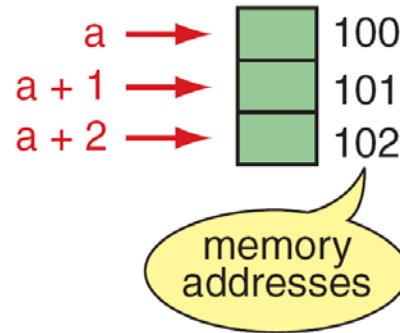
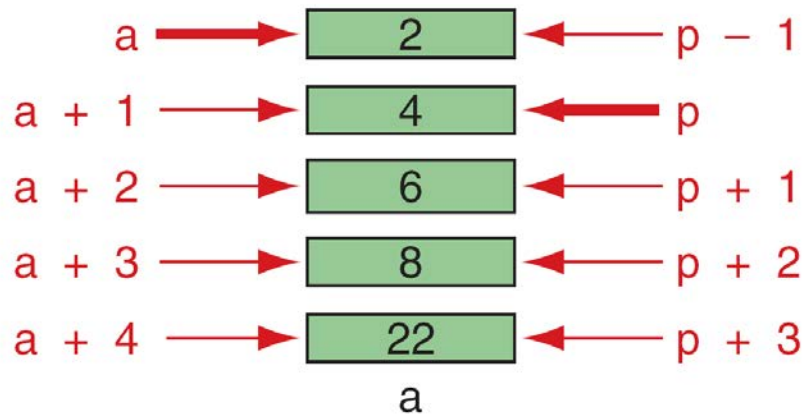
배열 포인터 (Pointer to Arrays)



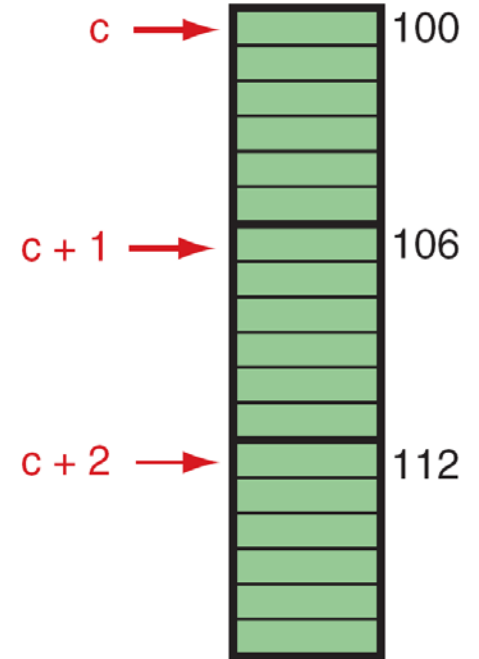
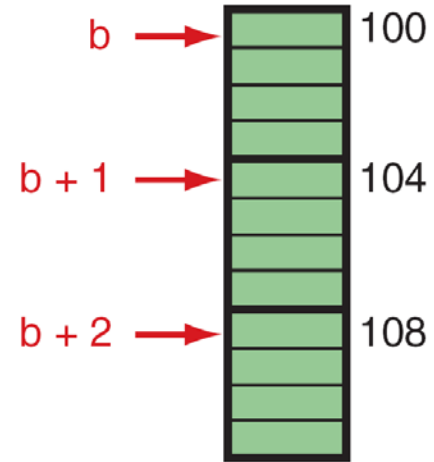
$*(a + i) \longleftrightarrow a[i]$

포인터 연산

- 포인터 $p \rightarrow p \pm n \rightarrow p + n * (\text{sizeof}(\text{one element}))$



```
char a[3];  
int b[3];  
float c[3];
```



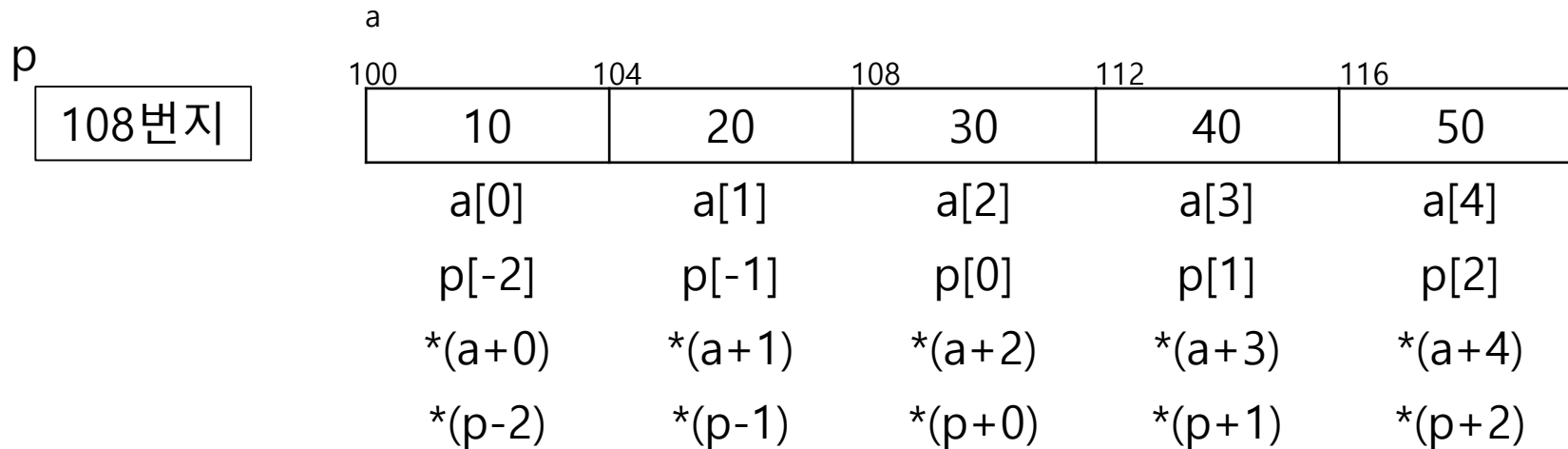
포인터와 배열

```
int a[5] = {10, 20, 30, 40, 50};
```

```
int *p = &a[2];
```

```
printf("%d %d %d %d\n", a[2], *(a+2), p[0], *(p+0)); // 30출력
```

```
printf("%d %d %d %d\n", a[1], *(a+1), p[-1], *(p-1)); // 20출력
```



2차원 Array

`table[2] == *(table+2)`

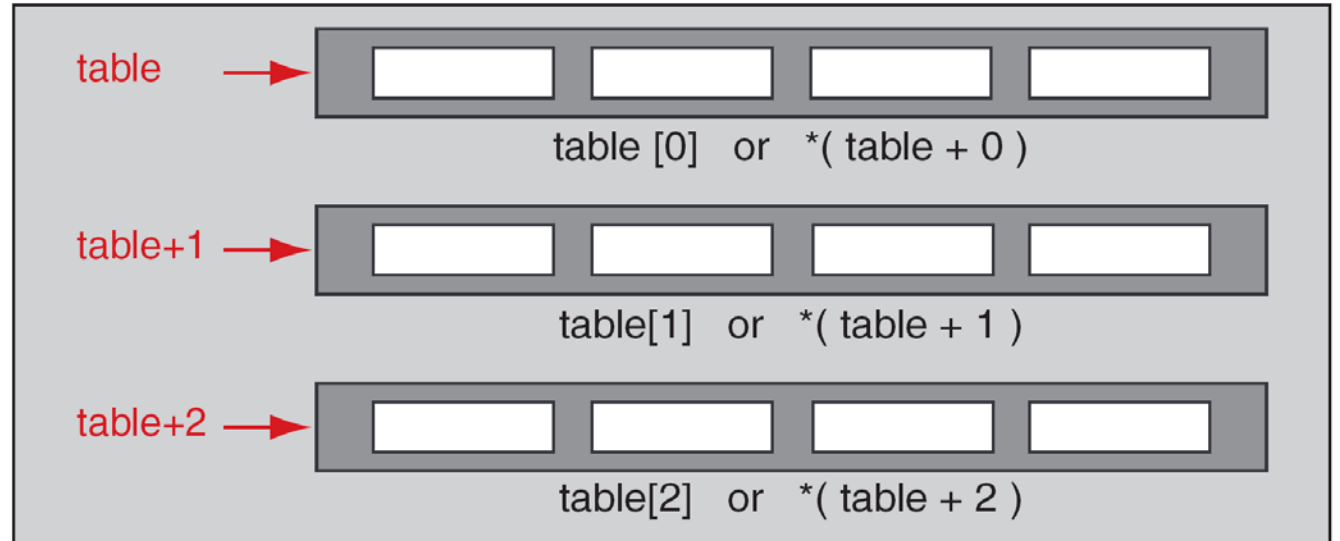
`table[2][1]`

`== (*(table+2))[1]`

`== *(table[2]+1)`

`== *(* (table+2)+1)`

- 헷갈리니 다차원 배열에서는 Index를 사용하자



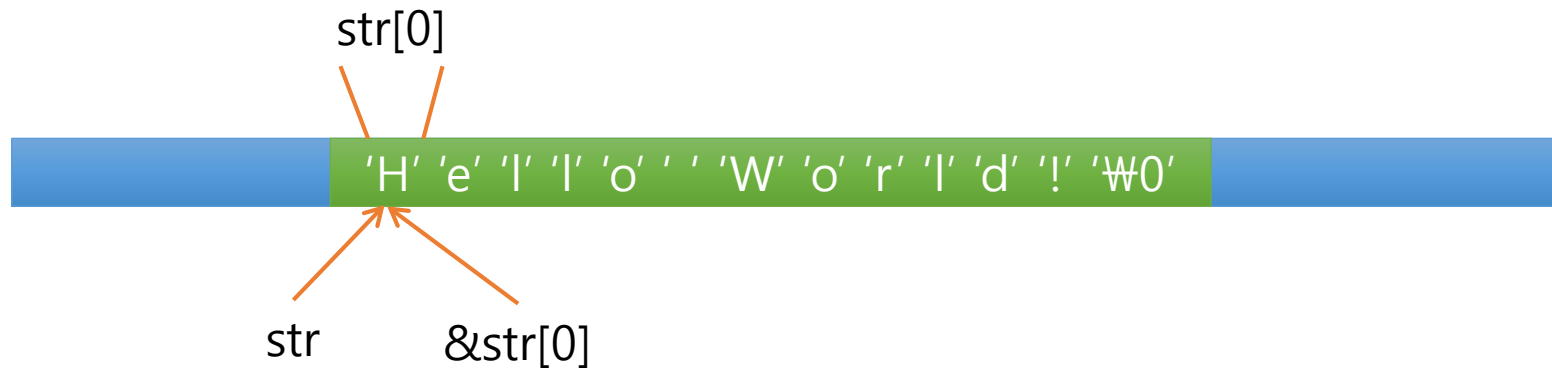
`int table[3][4];`

```
for (i = 0; i < 3; i++)  
{  
    for (j = 0; j < 4; j++)  
        printf("%6d", *(* (table + i) + j));  
    printf( "\n" );  
} // for i
```

Print Table

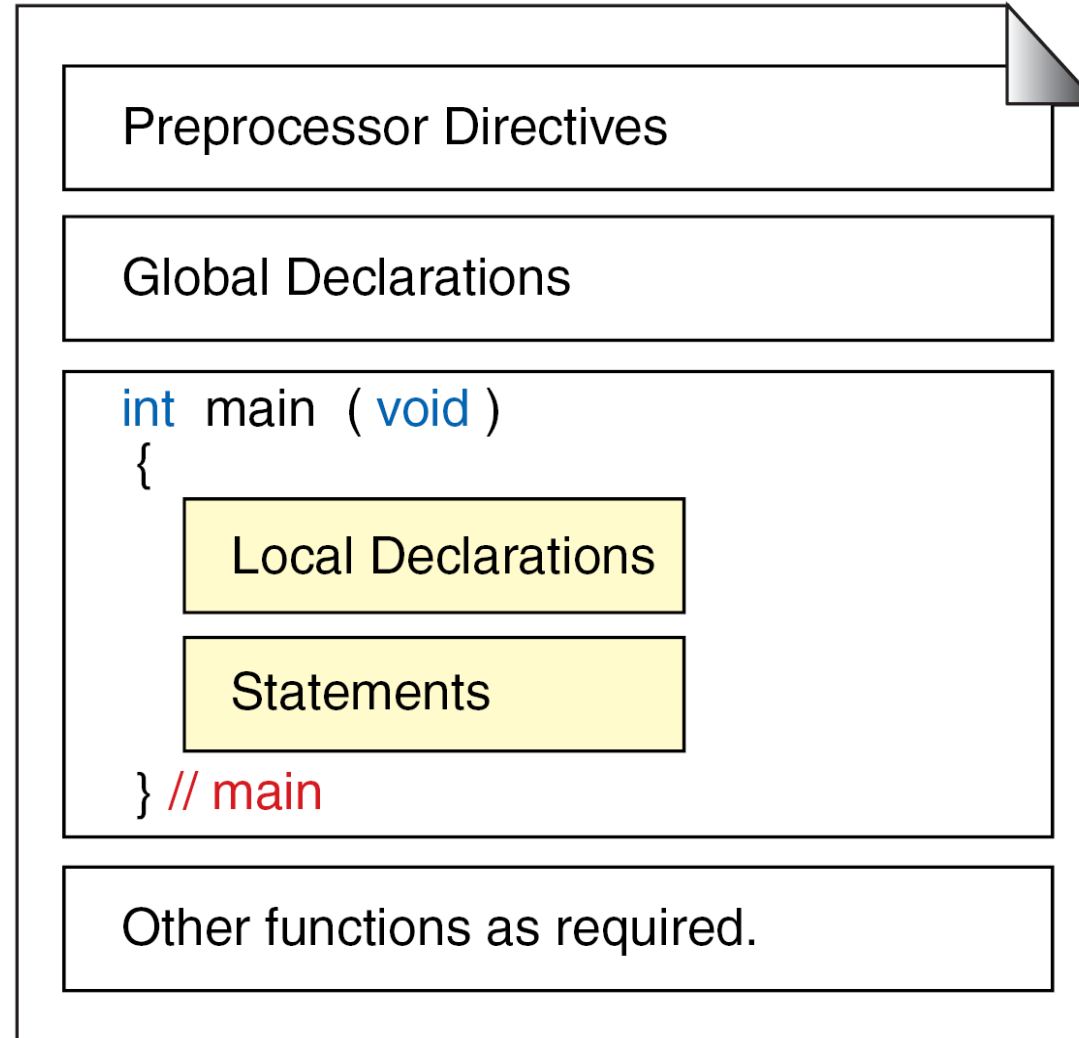
문자열과 포인터

- 다시한번! 여러개의 문자+ '₩0' 이 들어있는 배열
- 배열의 이름은 첫번째 element의 주소와 같다.



- 그래서 scanf로 %s 받을 때 & 안 붙인다!

Scopes



Storage Class (변수의 존재기간과 접근범위)

	지역변수	정적변수	전역변수	register변수
지정자	auto	static	extern	register
저장장소	스택	정적 데이터 영역	정적 데이터 영역	레지스터
선언위치	함수내부	함수내부/외부	함수내부/외부	함수내부
유효범위	함수내부	함수내부/외부	프로그램 전체	함수내부
생존기간	함수종료시	프로그램 종료시	프로그램 종료시	함수종료시
초기값	초기화 안됨	0으로 초기화	0으로 초기화	초기화 안됨

Dynamic Memory Allocation

동적할당

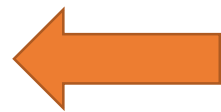
왜 쓰나요?

- 배열의 크기를 입력 받아 그 크기만큼의 배열을 만들고 싶을 때

```
int main(void) {  
    int size;
```

```
    scanf("%d", &size);
```

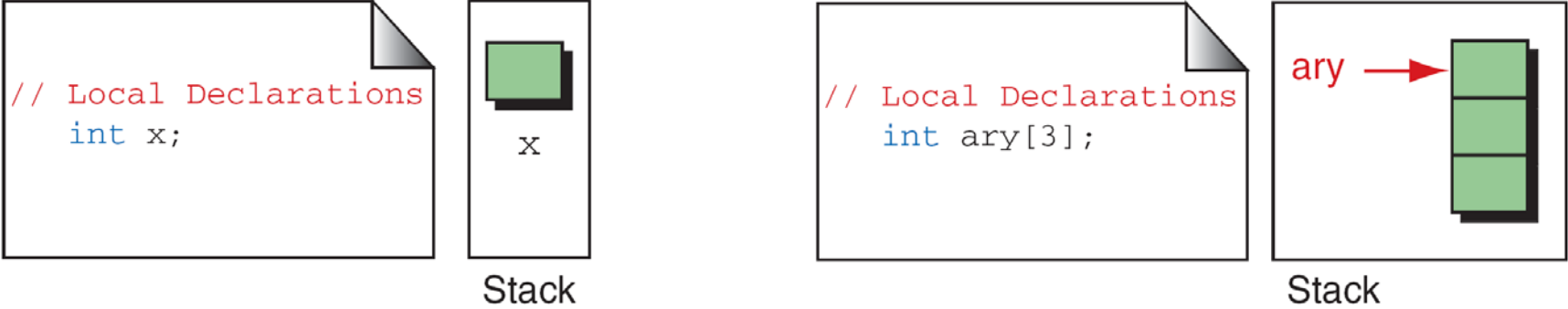
```
    int array[size];  
}
```



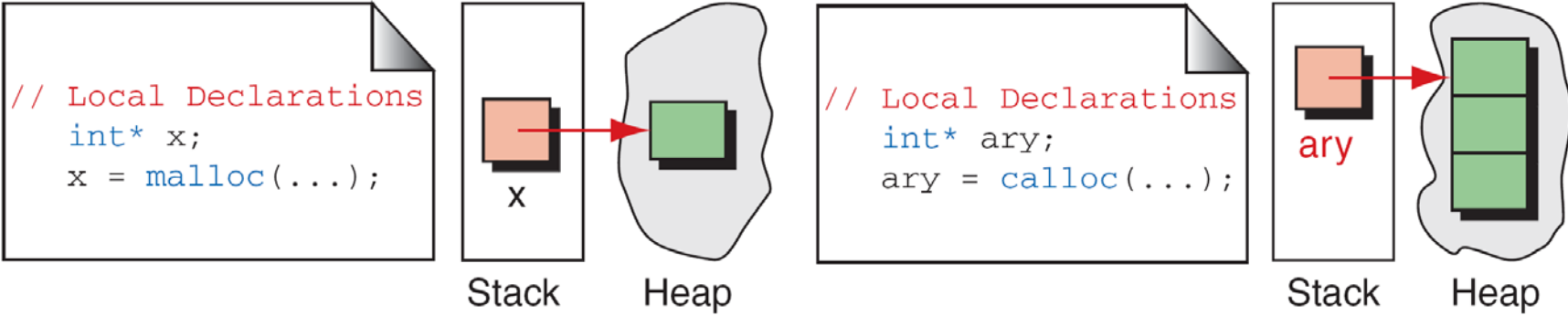
Compile Error! 선언은 항상 맨 위에 와야 한다.
선언 후에 배열이 할당되어야 하는데... 이럴 때는 어떻게?

- 프로그램 실행 중에 메모리를 할당해 데이터를 저장할 공간을 생성하는 방법

Accessing Dynamic Memory



(a) Static Memory Allocation



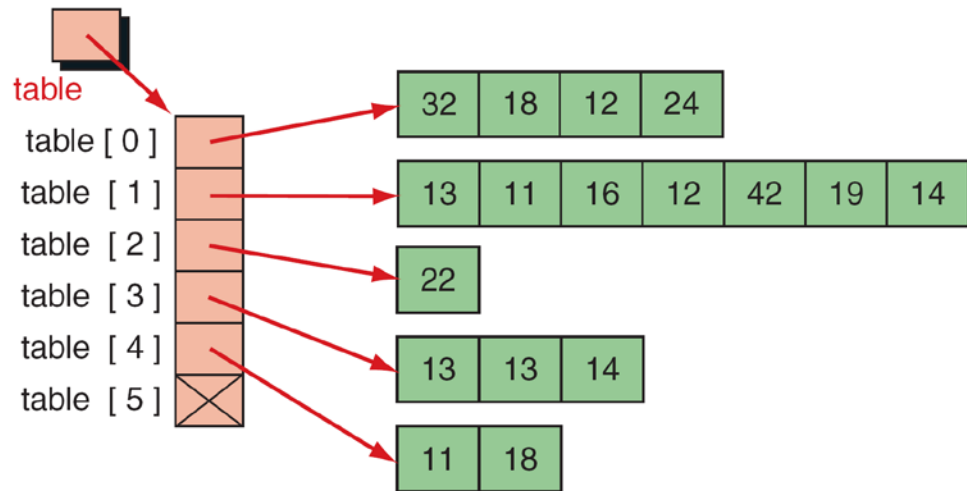
(b) Dynamic Memory Allocation

Dynamic Memory Allocation

- `<stdlib.h>`
- `void *malloc (size_t size);`
- `void *calloc (size_t count, size_t size);`
- `void *realloc (void* ptr, size_t newSize);`
- `void free (void* ptr);`

동적할당으로 2차원 배열 만들기 - 포인터 배열

```
int **table;  
table = (int **)calloc (3, sizeof(int *));  
table[0] = (int *)calloc(4, sizeof(int));  
...  
...
```



```
table = (int**)calloc (rowNum + 1, sizeof(int*));  
table[0] = (int*)calloc (4, sizeof(int));  
table[1] = (int*)calloc (7, sizeof(int));  
table[2] = (int*)calloc (1, sizeof(int));  
table[3] = (int*)calloc (3, sizeof(int));  
table[4] = (int*)calloc (2, sizeof(int));  
table[5] = NULL;
```

포인터의 부적절한 사용으로 인한 주요 부작용들

- **Unreachable Memory** (이 공간을 가리키는 포인터가 존재하지 않는다)
- **Dangling Pointer** (어딘가 가리키고 있기는 하지만, 그 위치에 뭐가 있는지 알 수 없다.)
- **Buffer Overflow** (할당되어 있는 양 이상으로 메모리를 다루게 되는 경우)
- **Segmentation Fault** (잘못된 주소 접근)

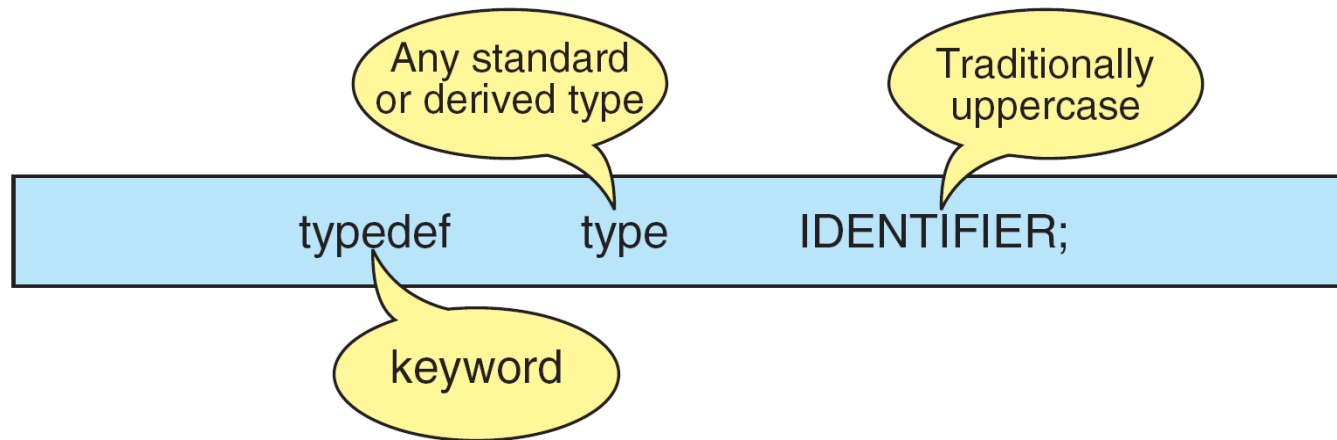
Tidy Up

- 동적으로 할당된 메모리는 **반드시 시스템에 반환**해야 함!
→ 그래야 다른 프로그램이 메모리 공간을 활용할 수 있다. (메모리 leak)
- **실행 전에 미리 할당 vs 실행 중에 동적으로 할당하고 반환**
→ 공간이 **필요한 만큼만** 쓰기 때문에 효율적!
→ 메모리 공간의 크기는 정해져 있다.
- 얼마나 큰 공간이 필요한 지 알 수 없을 때
→ 적당히 크게? - 공간이 부족하거나, 공간이 낭비되거나

Structure

The Type Definition (typedef)

- A type definition, typedef, gives a name to a data type by creating a new type that can then be used anywhere a type is permitted.
- 프로그래머는 길게 쓰는 것을 싫어한다



```
typedef int INGETER;
```

Enumerate Type

- 항목 혹은 선택지, 일종의 상수 항목을 만들 때 사용
- 각 항목들에 대해 identifier로써 하나의 정수 (enumeration constant)가 할당됨

- 상황) 메뉴를 만들고 숫자 하나 입력받아서 switch 문으로 각기 다른 함수를 실행해야 한다.
- 그냥 숫자 0, 1, 2, 3으로 나누면 나중에 헷갈리고 불편함

Initializing Enumerated Constants

```
enum MONTHS {JAN, FEB, MAR, APR};
```

identifier 자동 배정 0 1 2 3

```
enum MONTHS {JAN = 1, FEB, MAR, APR};
```

```
enum COLORS {RED, ROSE=0, CRIMSON=0, BLUE, AQUA=1};
```

Operations on Enumerated Types

- `enum COLORS color1, color2;`

```
color1 = BLUE;
```

```
color2 = color1;
```

- `if (color1 == color2)`

```
    if (color1 == BLUE)
```

- `switch (color1) {`

```
    case BLUE: ...
```

Initializing Enumerated Constants

- enum months {JAN, FEB, MAR, APR};

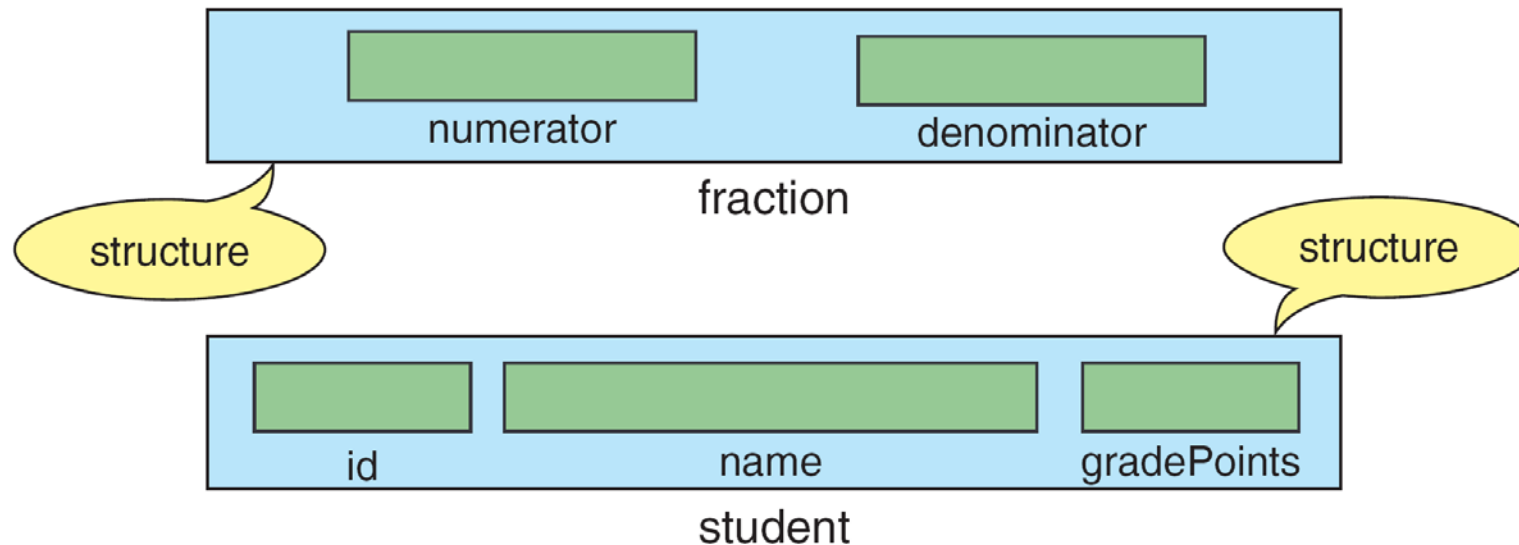
identifier 자동 배정 0 1 2 3

- enum months {JAN = 1, FEB, MAR, APR};

- enum color {RED, ROSE = 0, CRIMSON = 0, BLUE, AQUA = 1};

Structure Type

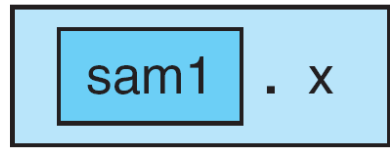
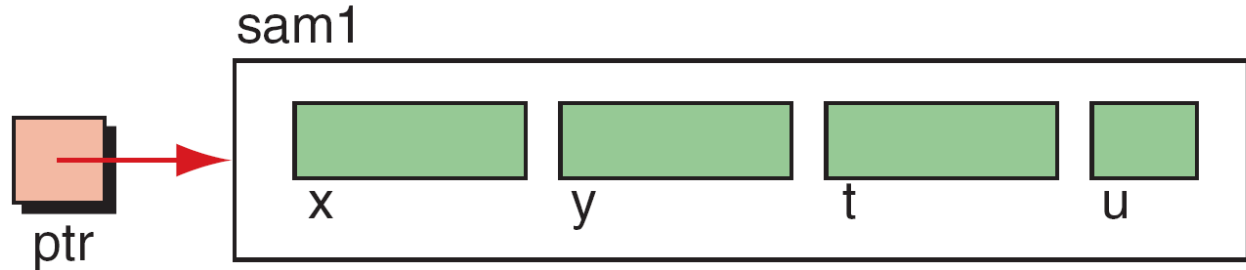
- 관련되는 element 들의 집합.
- 어떤 타입의 변수라도 올 수 있다.
- 단, 함수는 구조체의 element로 안 된다. 오직 변수만 가능



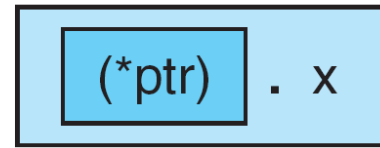
Accessing Structure Elements

```
typedef struct
{
    int    x;
    int    y;
    float  t;
    char   u;
} SAMPLE;

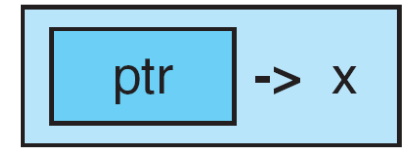
...
SAMPLE  sam1;
SAMPLE* ptr;
...
ptr = &sam1;
...
```



Direct Selection



Indirection



Indirect Selection

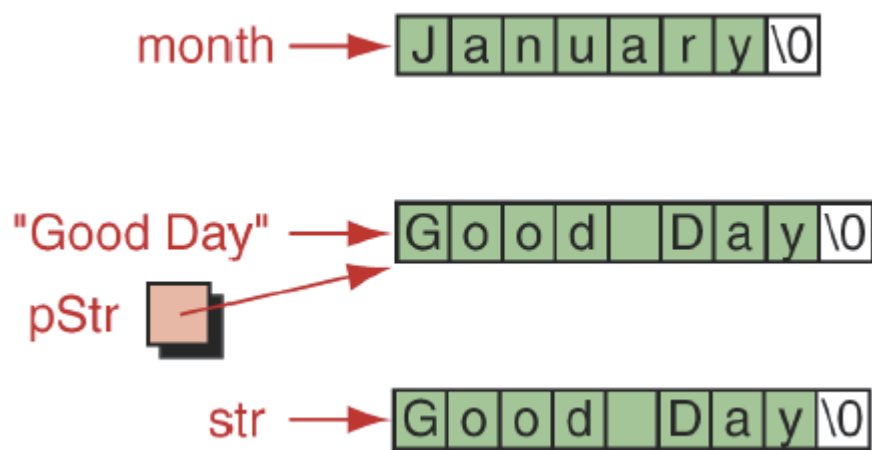
Three Ways to Reference the Field **x**

(*pointerName).fieldname ↔ **pointerName->fieldName.**

String

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', '\0', 'D', 'a', 'y', '\0'};`



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

(a) String Declaration

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

(b) String Pointer Declaration



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).

Difference btw. Input functions

gets/fgets

- gets does not include newline ('\n'), fgets does.
- gets does not allow to specify a maximum size for *str* (which can lead to buffer overflows).

scanf/gets

- scanf는 단어 단위로 (space), gets는 줄 단위로 (newline)

Difference btw. Output functions

- puts appends a newline ('\n') at the end automatically
- fputs does not write additional characters

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 }
```

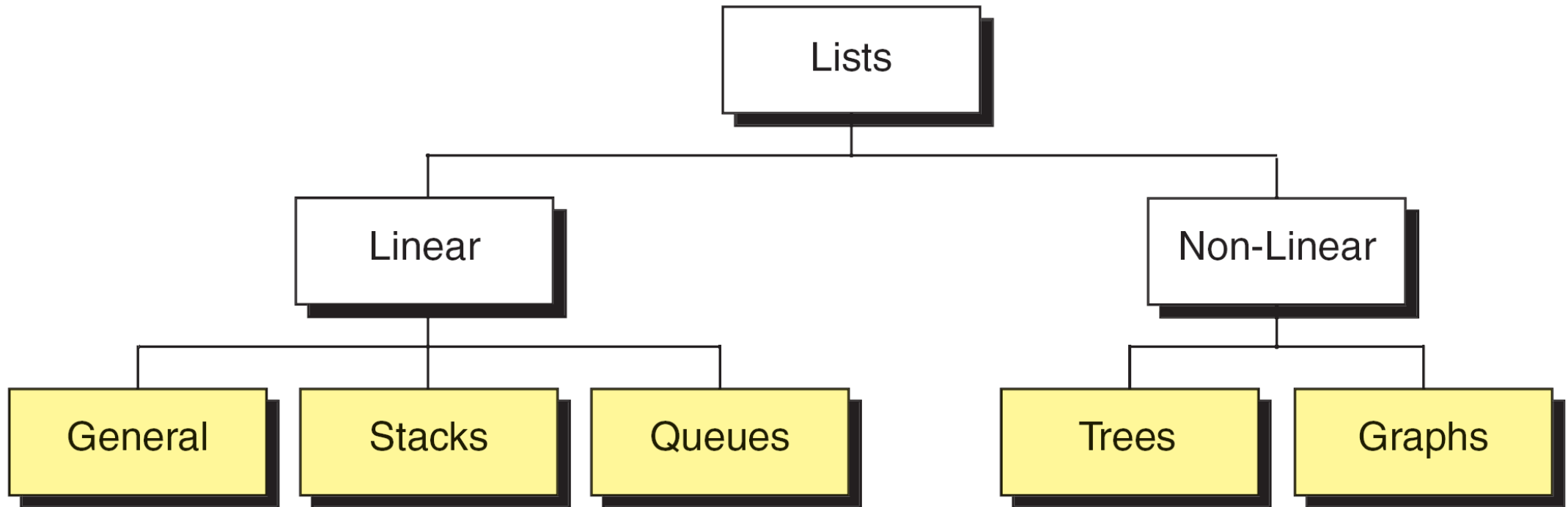
```
fflush();  
#define FLUSH while (getchar() != '\n')
```

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

List

What to learn...



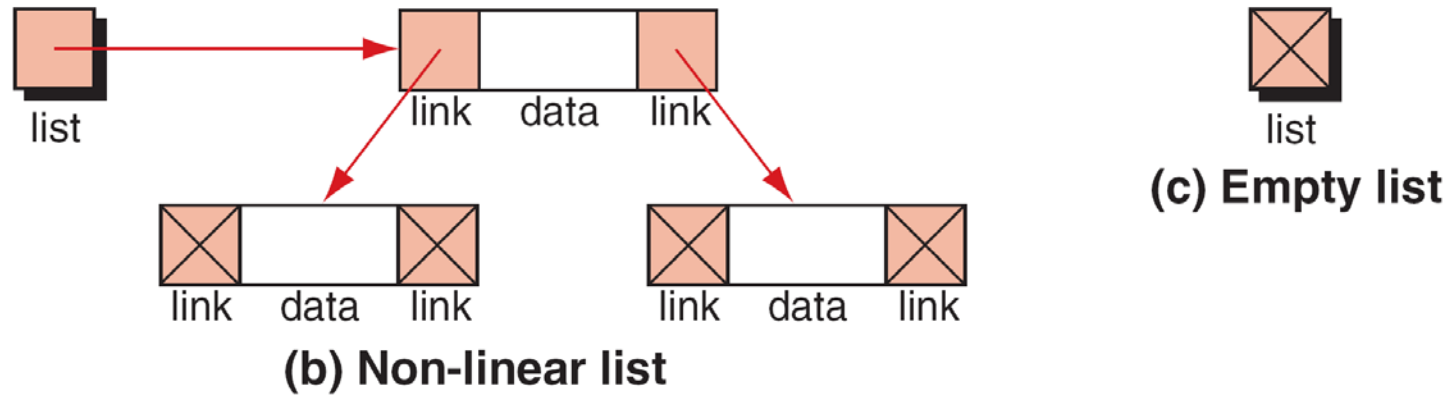
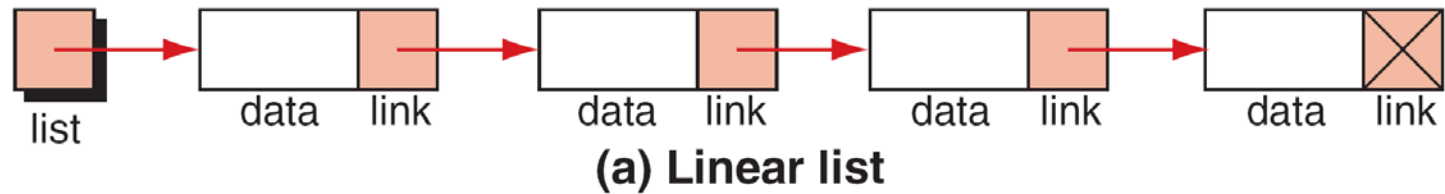
Linked List

- 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	$O(n)$	$O(1)$

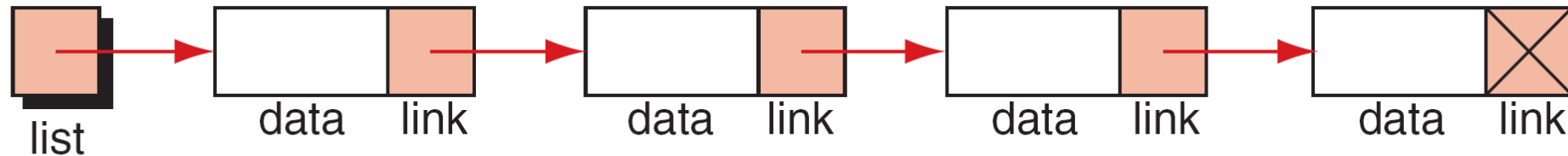
Linked List

- 한개만 가리킬 수도 있고, 여러개를 가리킬 수도 있고...



General Linear Lists

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

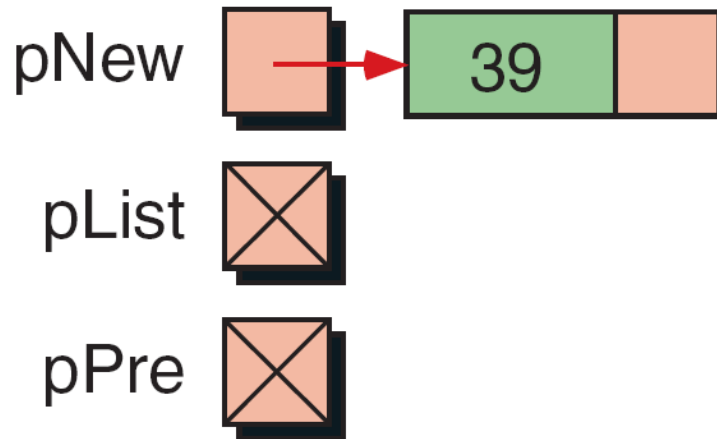


(a) Linear list

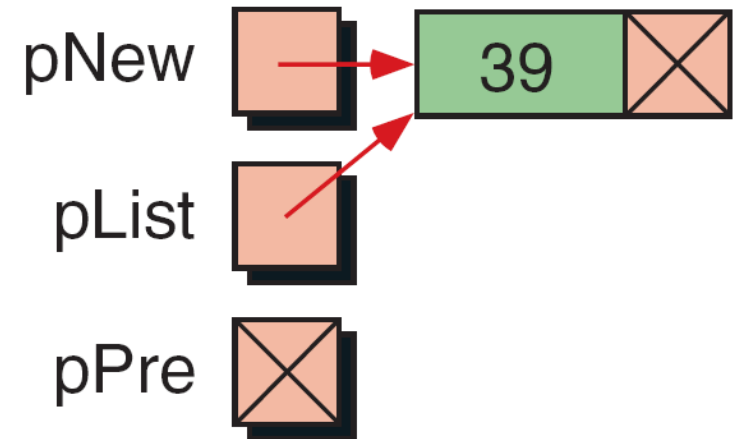
Insert a Node to Empty List

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

Before Add

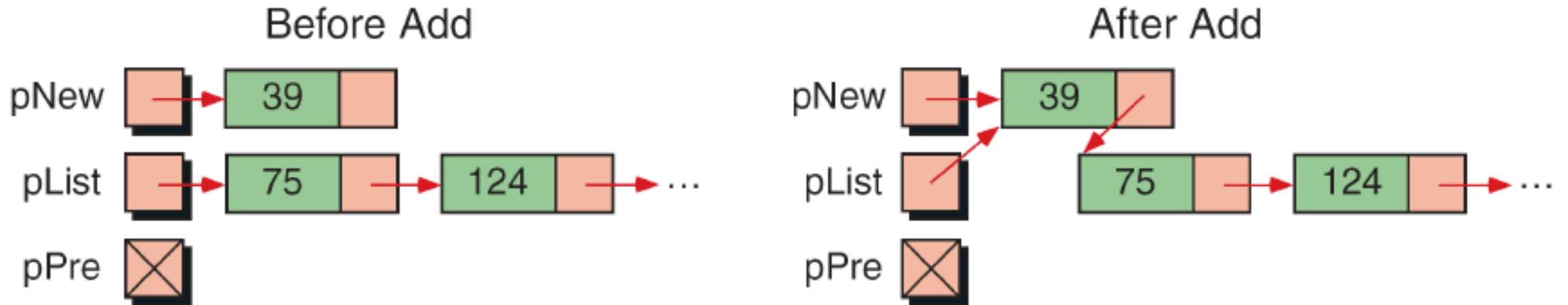


After Add



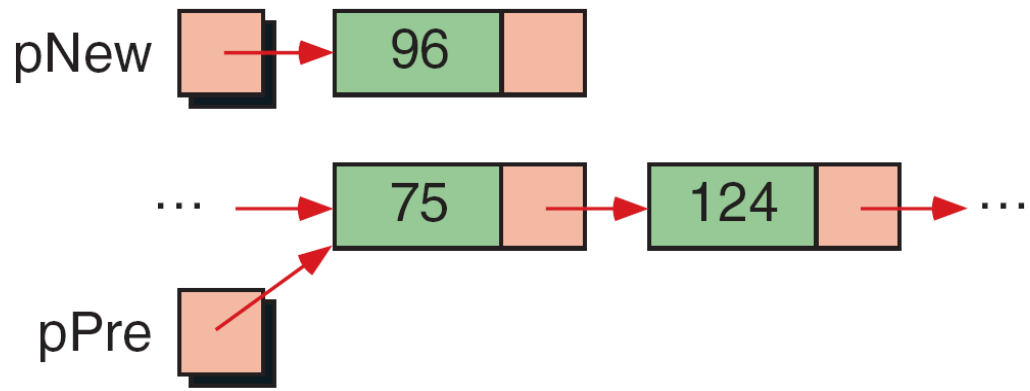
Insert a Node at Beginning

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

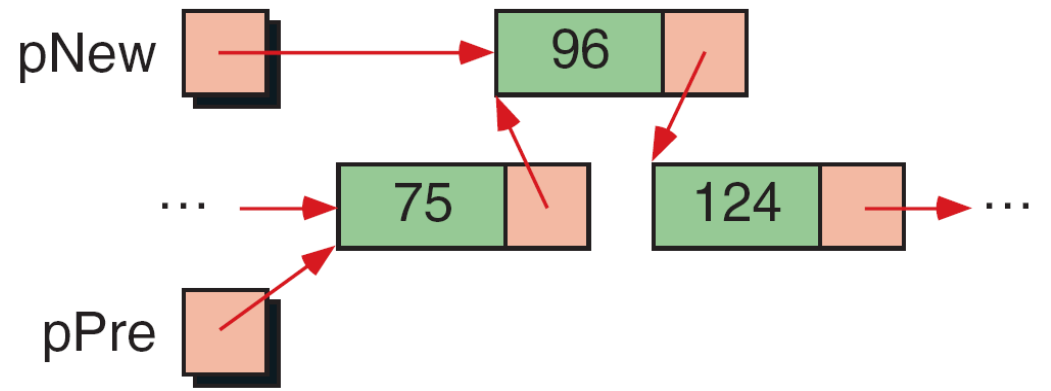


Insert a Node in Middle

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



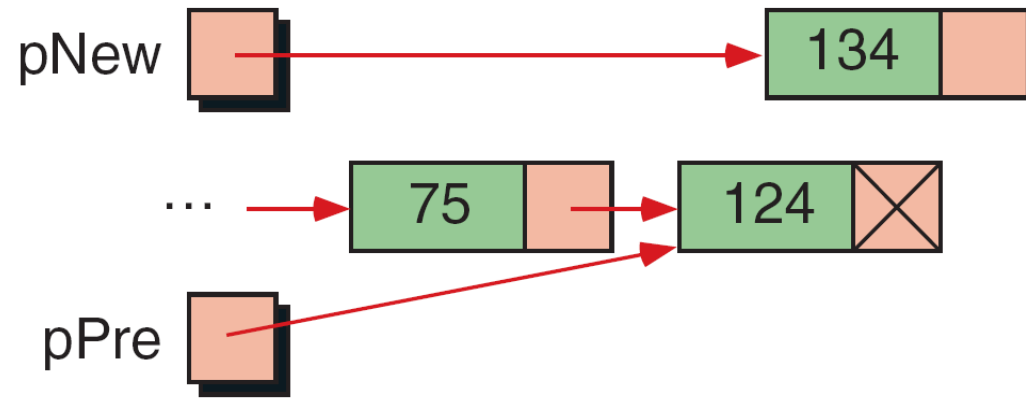
Before Add



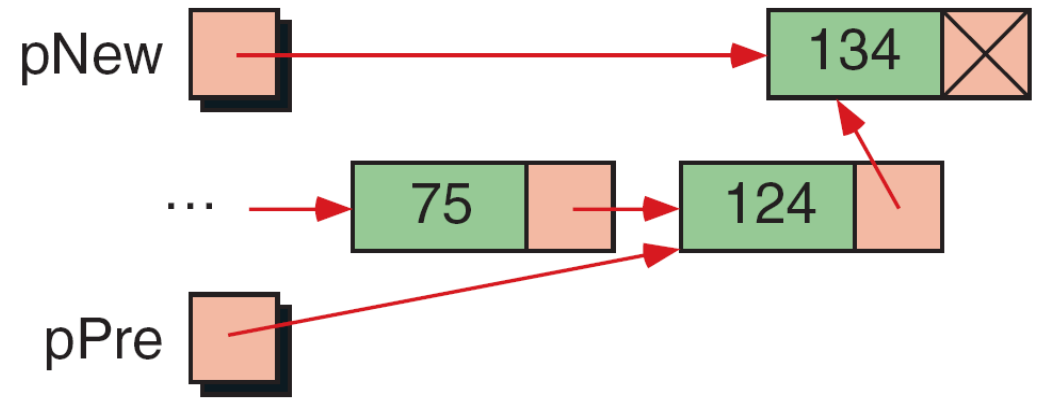
After Add

Insert a Node at End

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



Before Add

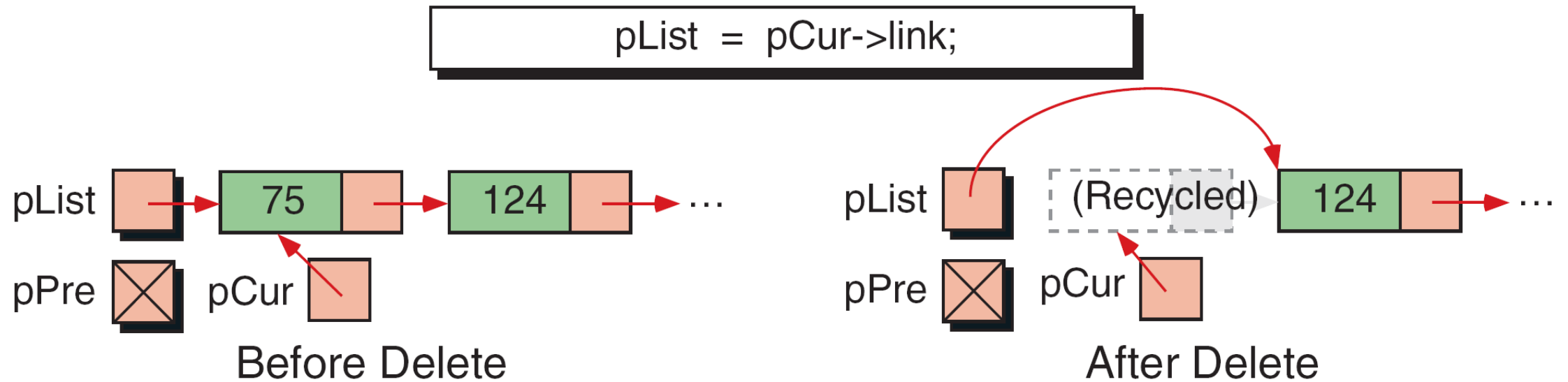


After Add

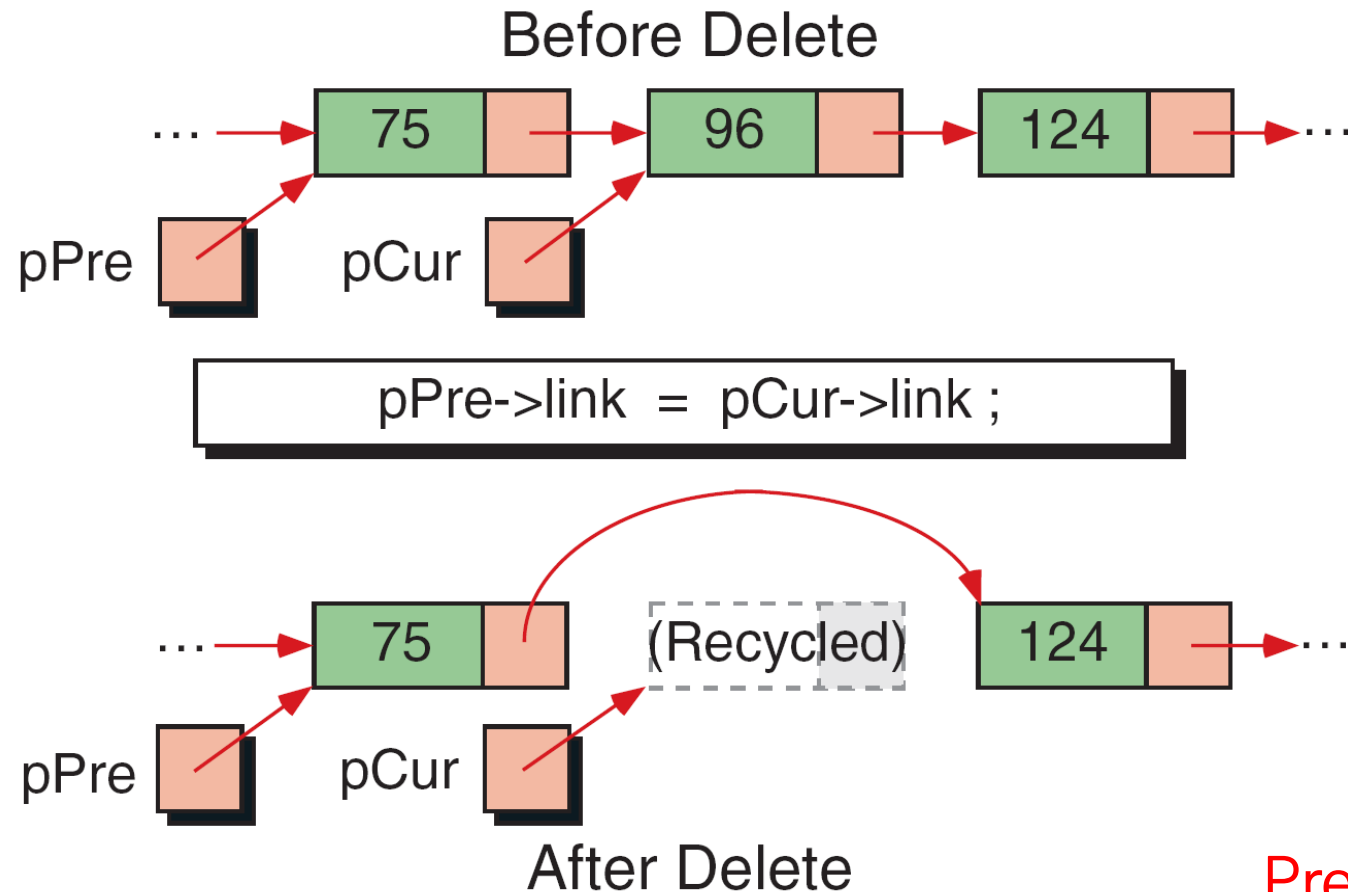
Insert a Node – Code

```
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!!

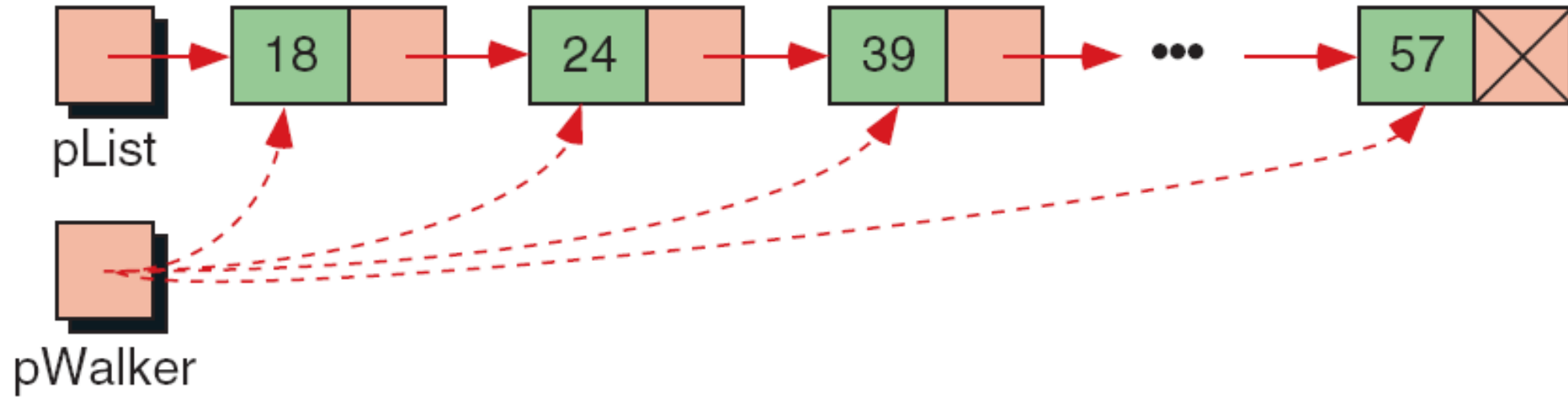
Delete a Node – Code

```
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 } // deleteNode
```

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

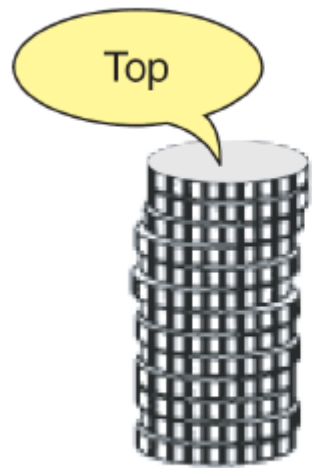
```
5 void printList (NODE* pList)
6 {
7 // Local Declarations
8     NODE* pWalker;
9
10 // Statements
11     pWalker = pList;
12     printf("List contains:\n");
13
14     while (pWalker)
15     {
16         printf("%3d ", pWalker->data.key);
17         pWalker = pWalker->link;
18     } // while
19 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

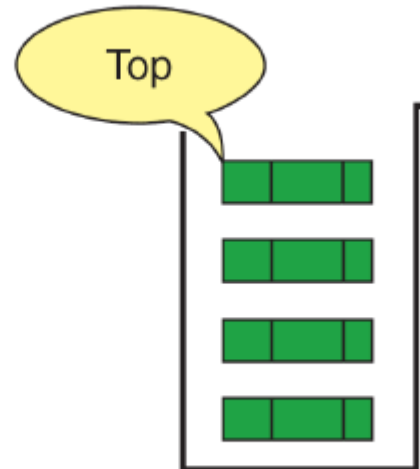
- Last in First out (LIFO)의 데이터 스트럭처
- Insertion과 deletion이 항상 한쪽 끝(Top)에서만 일어난다.
- Push : 새로운 것을 넣는 작업
- Pop : 쉘 위에서 하나 빼는 작업



Stack of Coins



Stack of Books

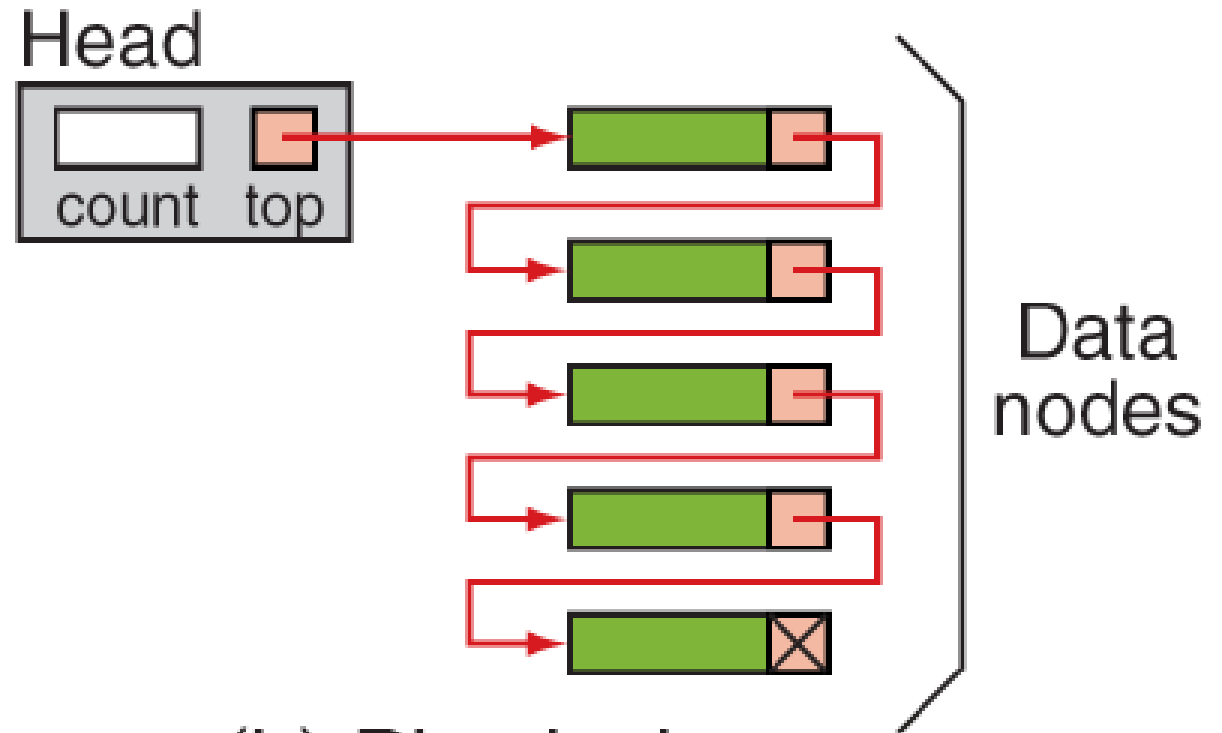


Computer Stack

Stack Implementations

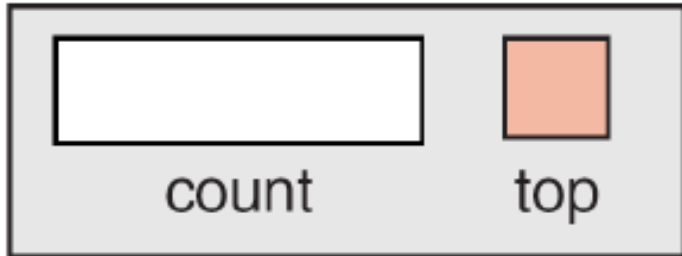


(a) Conceptual

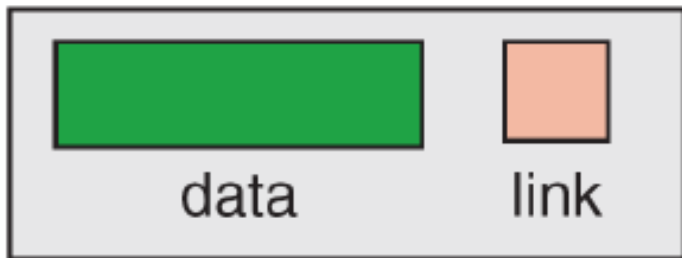


(b) Physical

Stack Data Structure



Stack Head Structure

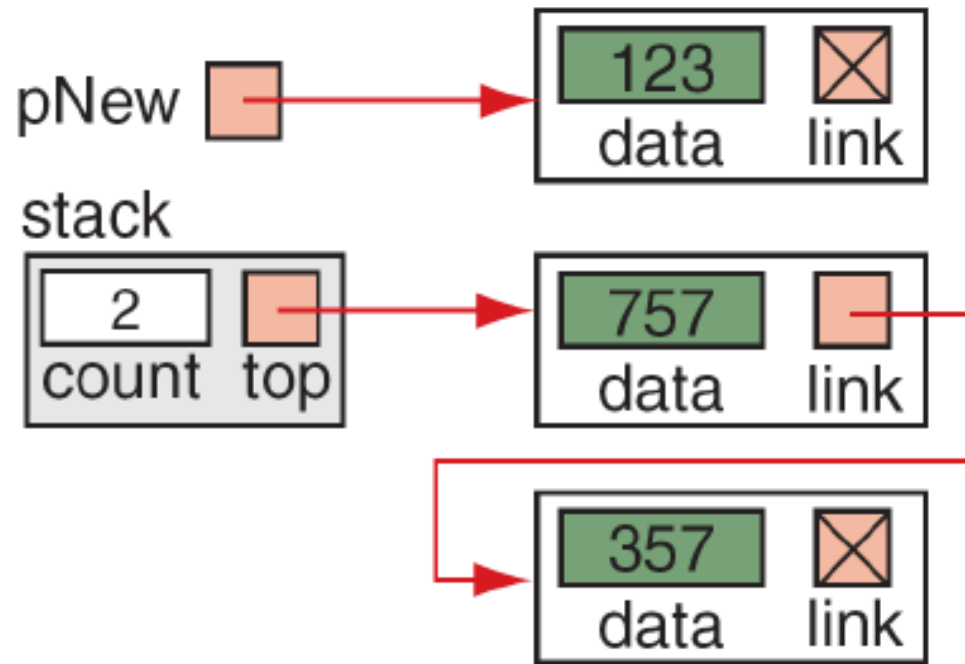


Stack Node Structure

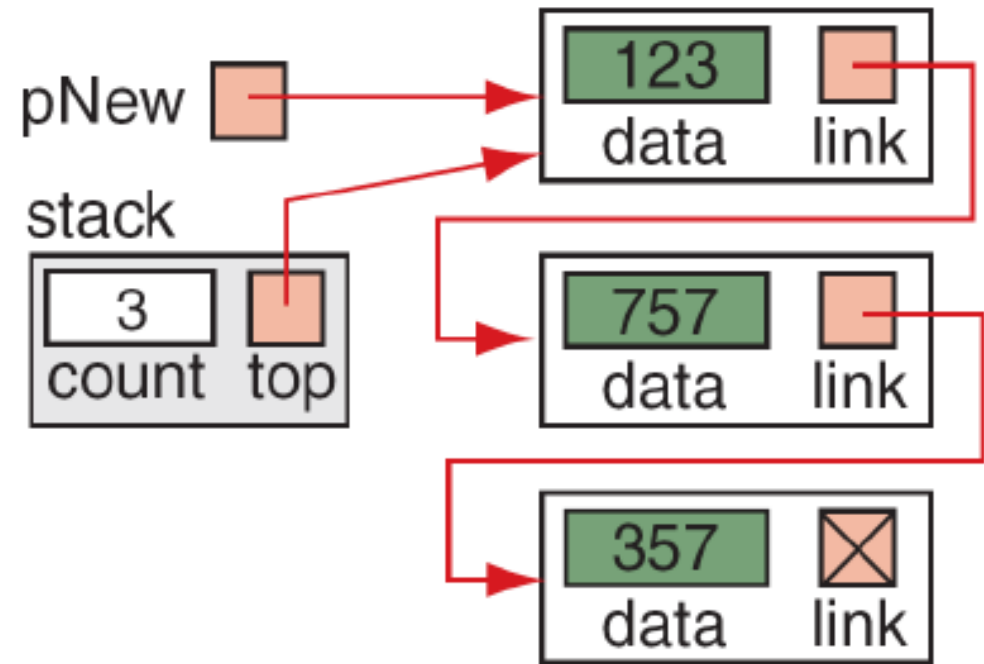
```
typedef struct
{
    int          count;
    struct node* top;
} STACK;

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

Stack_Push

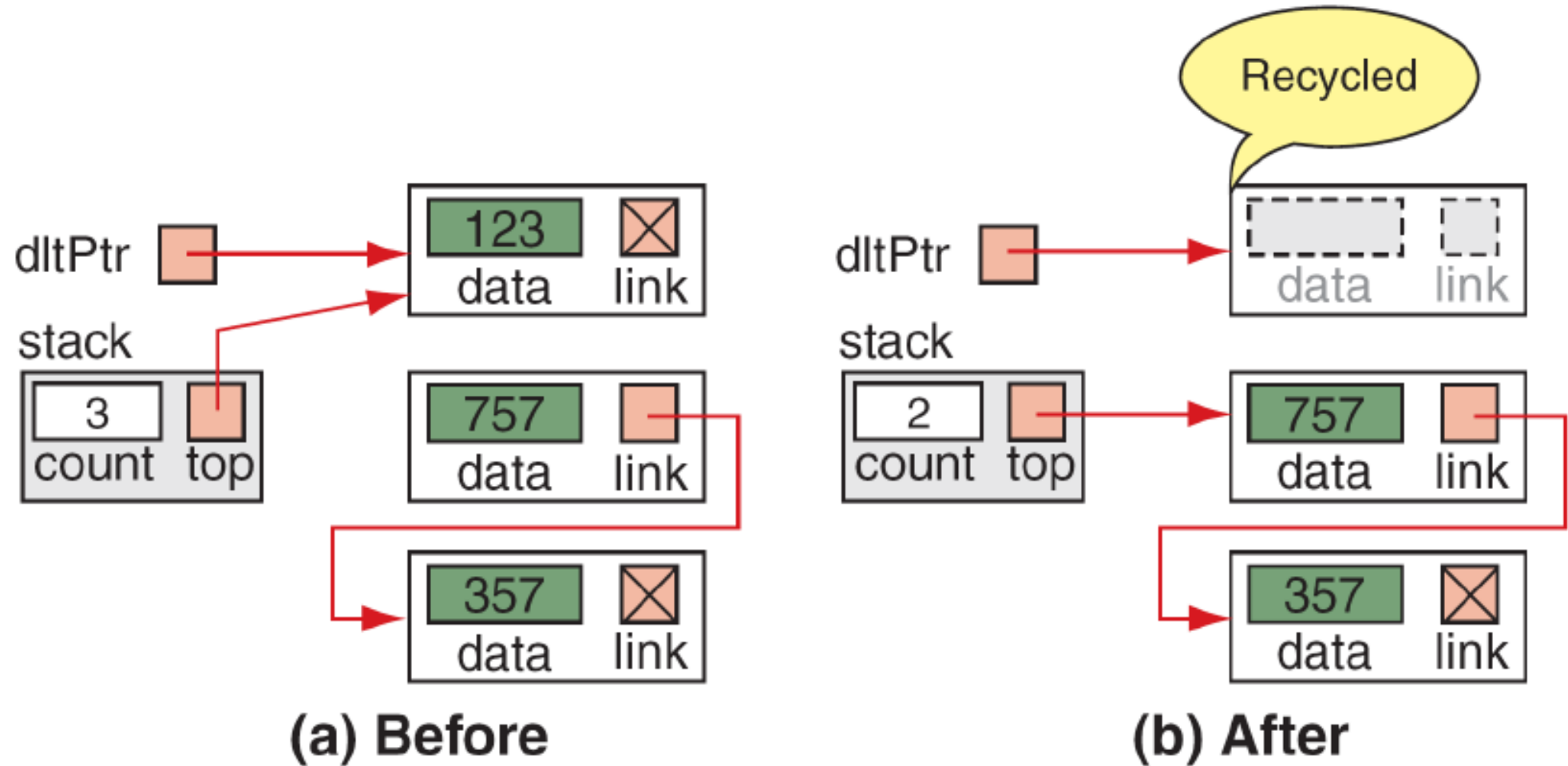


(a) Before



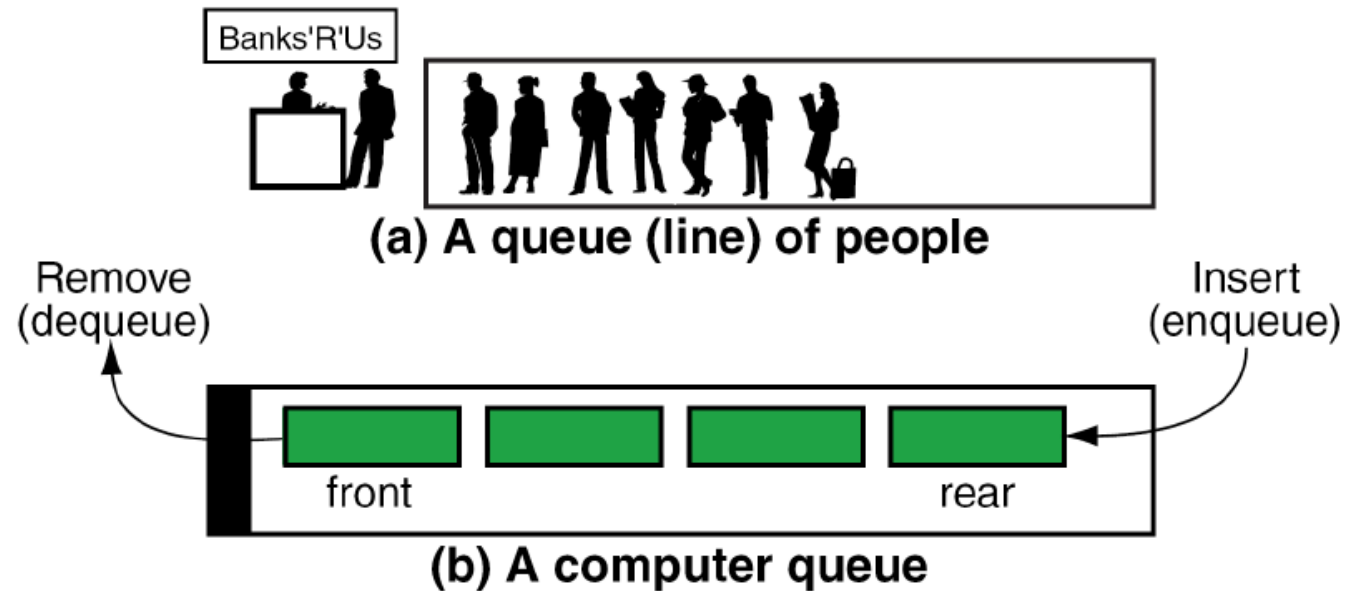
(b) After

Stack_Pop

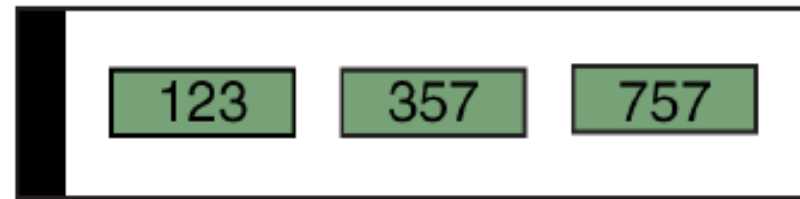


Queue

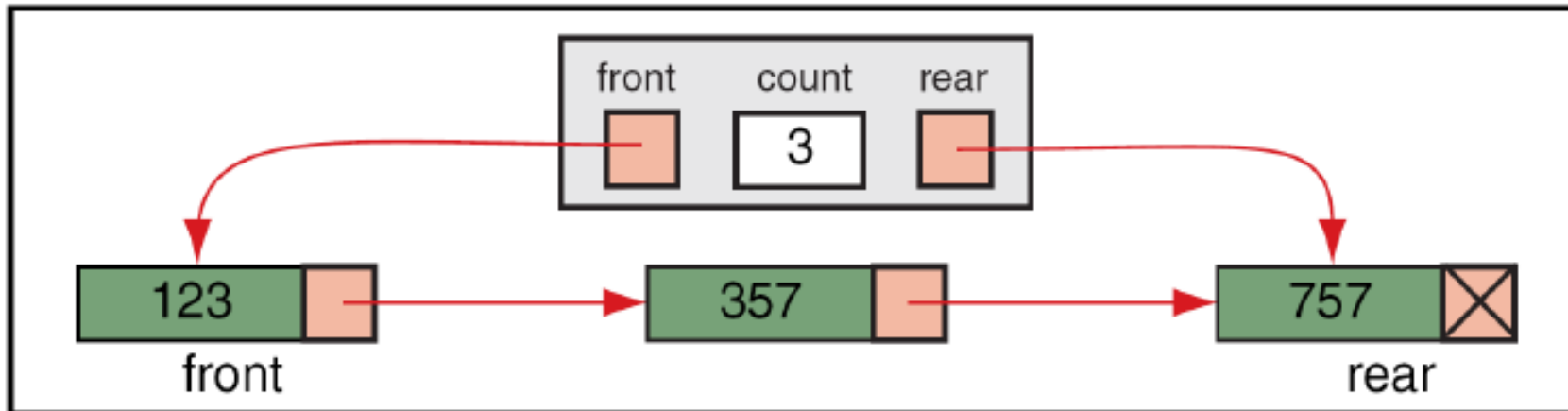
- **First in First out (FIFO)**의 데이터 스트럭처
- Insertion은 한쪽 끝, Deletion은 **다른** 한쪽 끝에서만 일어난다. :
- Enqueue : 큐에 새로운 걸 넣는 작업 (rear)
- Dequeue : 큐에서 하나를 제거하는 작업 (front)



Queue Implementations



(a) Conceptual queue

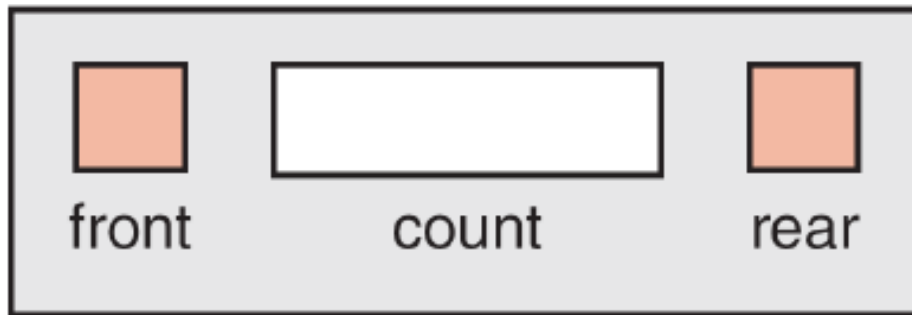


(b) Physical queue

Queue Data Structure



Node Structure

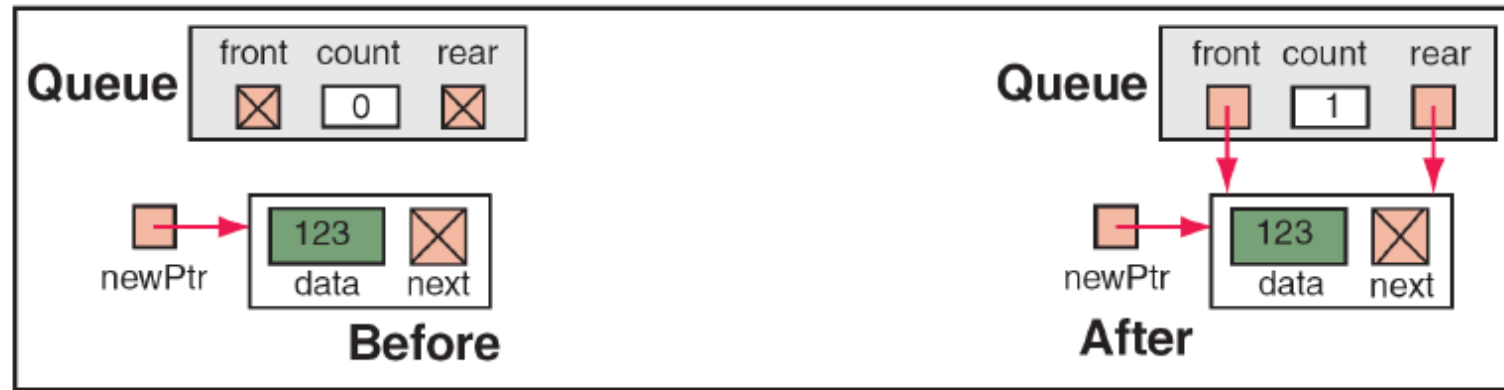


Head Structure

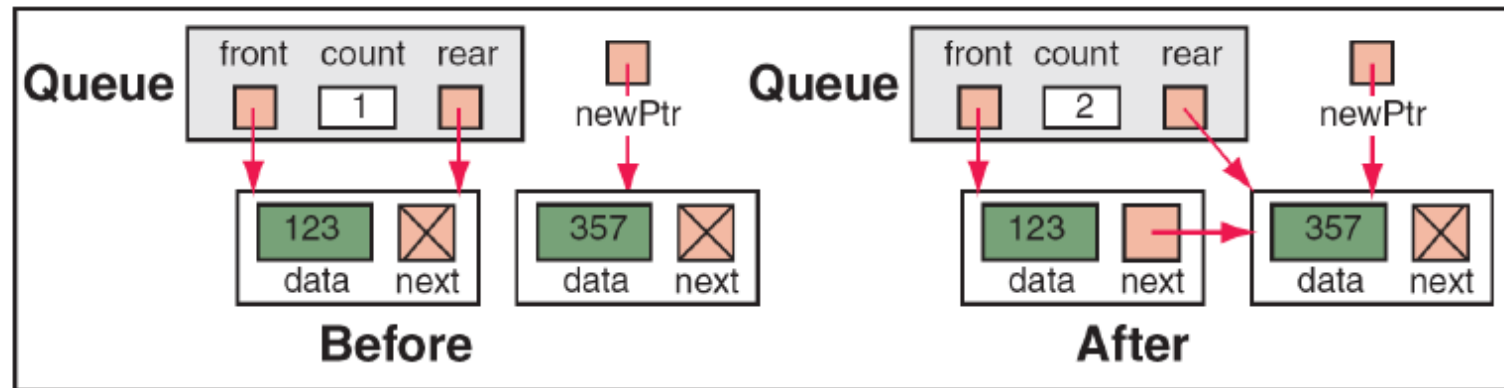
```
typedef struct node
{
    int          data;
    struct node* next;
} QUEUE_NODE;
```

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

Enqueue

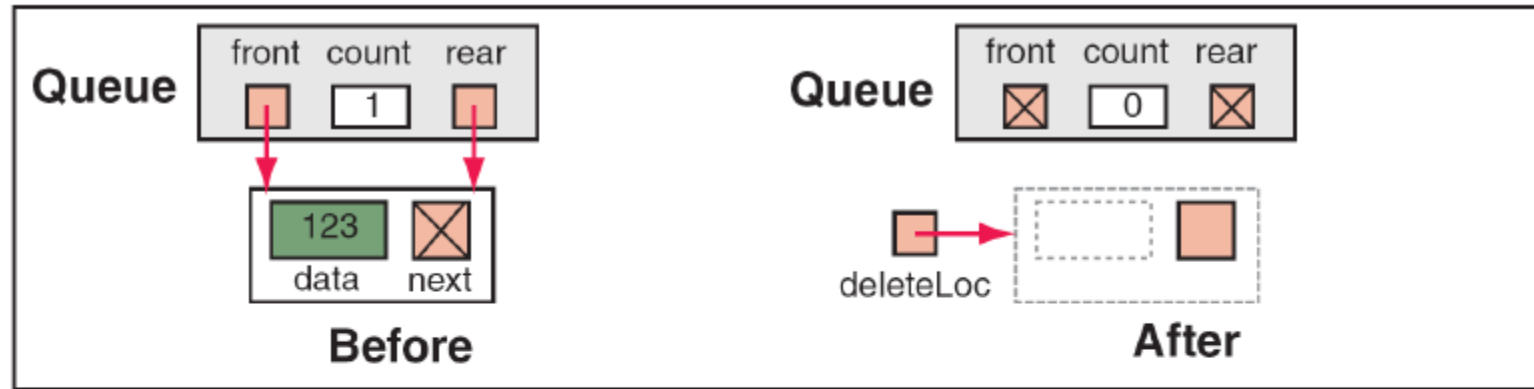


(a) Case 1: Insert into Null Queue

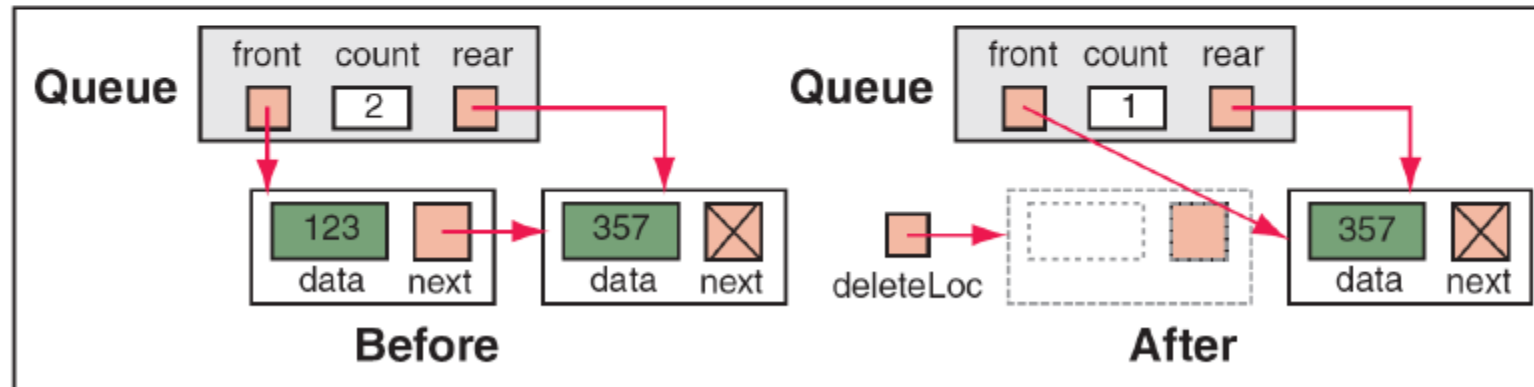


(b) Case 2: Insert into Queue with Data

Dequeue



(a) Case 1: Delete only item in queue



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

한 학기 배운 것 총 정리

- 책에 사이사이 박스에 있는 예제 코드들 **전부 다** 한번씩 꼭 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