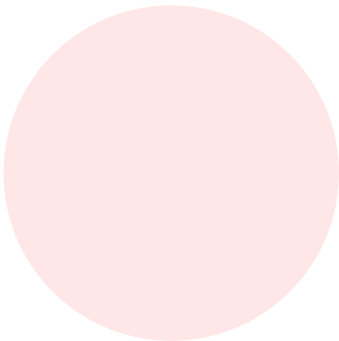


제13장

동적 메모리와 연결리스트

6.2.1



동적 할당 메모리의 개념

- 실행 프로그램이 메모리를 할당 받는 방법
 - 정적(static)
 - 동적(dynamic)
- 정적 메모리 할당 (static memory allocation)
 - 프로그램 수행 전에 일정한 크기의 메모리를 할당 받는 방법
 - 메모리의 크기는 프로그램이 시작하기 전에 결정

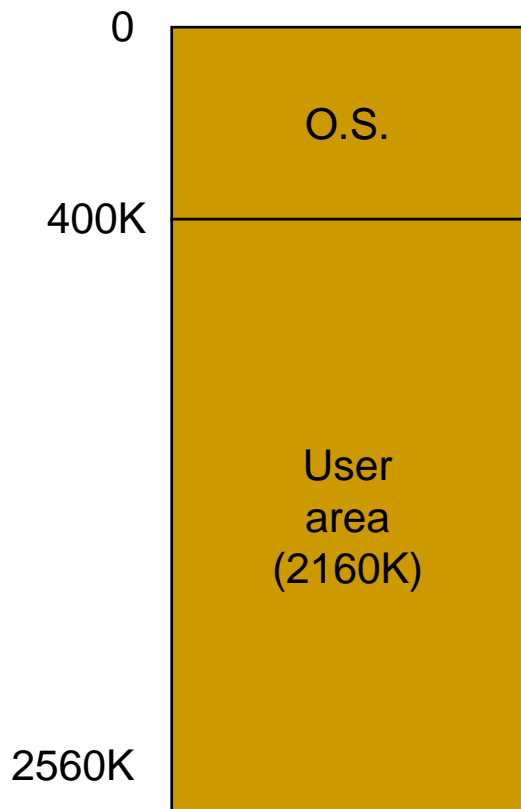
```
int i, j;  
int buffer[80];  
char name[] = "data structure";
```

- 수행 전에 할당된 크기보다 더 큰 자료가 필요한 경우 처리하지 못함
- 더 작은 자료라면 남은 메모리 공간은 낭비

메모리 관리

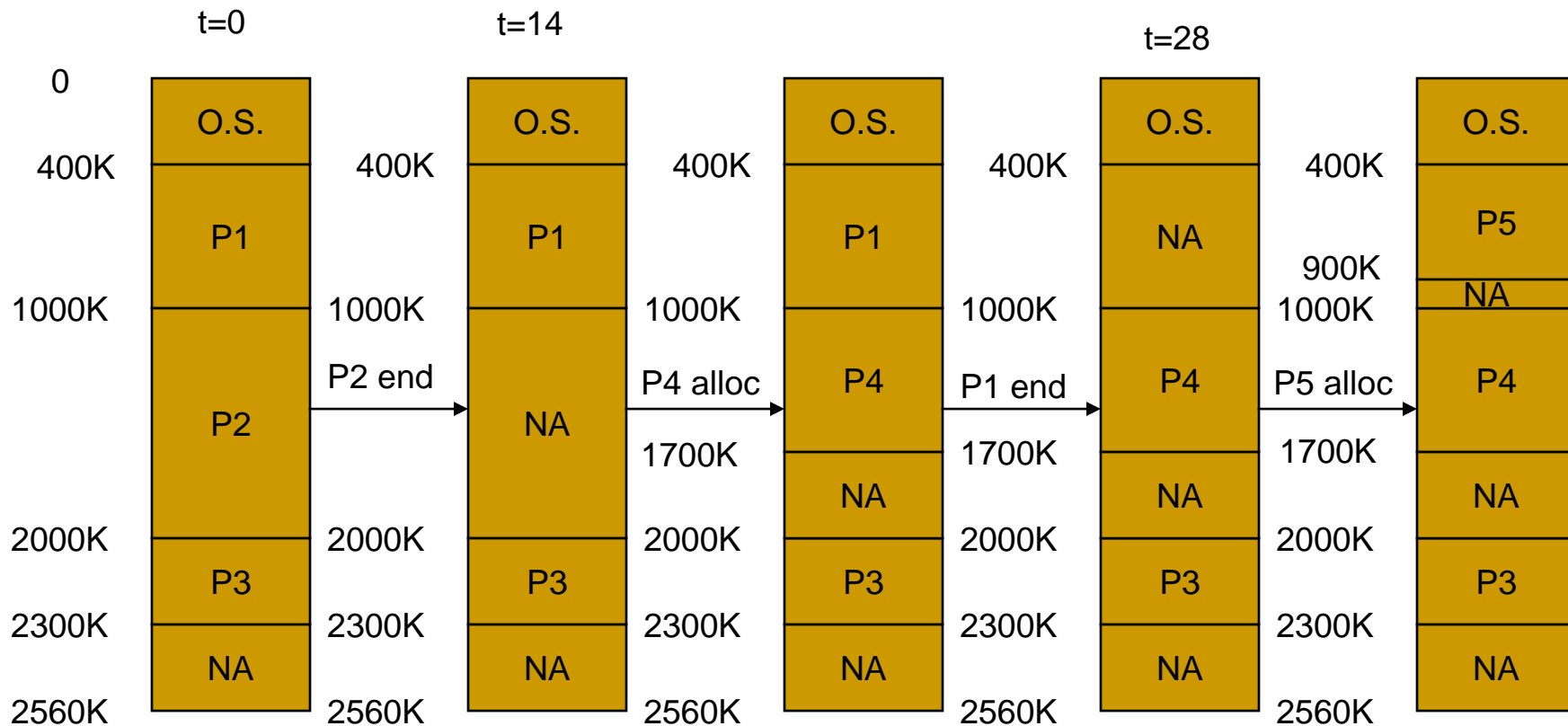
- Memory Partition 방법에 따라
 - 연속 할당(Contiguous Allocation)
 - 단일 분할 할당
 - 다중 분할 할당
 - 비 연속 할당(Non-contiguous Allocation)
 - Paging
 - Segmentation
 - Segmentation with Paging
- Memory allocation 방법에 따라
 - Preemptive allocation
 - Non-preemptive allocation
 - Dynamic allocation
 - Static allocation

연속 다중 분할 예



process	size	exe. time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

Assume Round-robin scheduling
with time-slice of 1 time unit



동적 메모리 할당

- 필요정보
 - **available space list**
 - each entry : base address of a free block, and the size
 - occupied space list
 - each entry : block name, base address, and block size. 그리고 각 블록의 크기가 일정한 크기를 가진다면 블록 크기에 대한 정보는 없어도 무관하다
 - secondary-memory directory 혹은 list
 - 현재 수행중인 프로그램과 연관된 블록들을 포함하는 디렉토리들의 리스트

동적 메모리 할당 알고리즘

- S_1, S_2, \dots, S_n 을 사용 가능한 n 개의 hole의 크기라 하고, S 를 현재 할당을 필요로 하는 프로그램 혹은 데이터의 크기라고 하자
- 4가지 할당 알고리즘
 - First-Fit Algorithm
 - Best-Fit Algorithm
 - Worst-Fit Algorithm
 - Buddy system
- Available space list의 예를 들면...


address	size
A1	S1
A2	S2
A3	S3
...	...
An	Sn

First-Fit Algorithm

- Available Space List(ASL)
 - 이 알고리즘은 available space list(ASL)가, 각 hole들의 시작 주소에 대해서 올림차순으로 정렬(sort)되어 있다는 전제를 하고 있다
- ASL 탐색 및 hole의 발견
 - ASL을 처음부터 탐색하여, 요청 크기인 S 보다 큰 hole들 중에서 처음 만나는 hole을 찾아 할당하는 방법이다.
 - 즉, ASL이 hole들의 시작 주소에 대해 올림차순으로 정렬되어 있으므로, 탐색을 시작하여 처음 만나는, S 보다 같거나 큰 크기의 hole을 찾아 할당하면 된다

First-Fit 알고리즘 예

- 현재 메모리의 상황이 아래와 같은 ASL로 표현되고 있을 때, 어떤 프로그램의 수행을 위해 **5K**의 메모리 할당을 요청하면, **First-Fit** 방식을 사용할 경우, 어느 주소에 할당되겠는가?



address	size
100	1K
1300	10K
15000	5K
25000	8K
50000	20K


Best-Fit Algorithm

- Available Space List(ASL)
 - 이 알고리즘은 available space list(ASL)가, 각 hole들의 크기에 대해서 올림차순으로 정렬(sort)되어 있다는 전제를 하고 있다
- ASL 탐색 및 hole의 발견
 - ASL을 처음부터 탐색하여, 요청 크기인 **S**보다 큰 hole들 중에서 가장 작은 hole을 찾아 할당하는 방법이다. 그러므로, ASL이 이처럼 각 hole의 크기 순으로 정렬되어 있다면, 탐색을 시작하여 처음 만나는, **S** 보다 같거나 큰 크기의 hole을 찾아 할당하면 된다
 - 할당하고 남은 공간을 최소화할 수 있는 방법으로 메모리 조각화(fragmentation) 현상을 최소화 시킬 수 있어, 메모리 사용 효율이 가장 높으나, 메모리 할당할 때마다, ASL을 정렬(sort)해야 하는 overhead때문에 실제로 사용하기에는 무리가 따른다

Best-Fit 알고리즘 예

- 현재 메모리의 상황이 아래와 같은 ASL로 표현되고 있을 때, 어떤 프로그램의 수행을 위해 **5K**의 메모리 할당을 요청하면, **Best-Fit** 방식을 사용할 경우, 어느 주소에 할당되겠는가?

address	size
100	1K
15000	5K
25000	8K
1300	10K
50000	20K




Worst-Fit Algorithm

- Available Space List(ASL)
 - 이 알고리즘은 Best-Fit Algorithm의 경우와 반대로, available space list(ASL)가, 각 hole들의 크기에 대해서, 내림차순으로 정렬(sort)되어 있다는 전제를 하고 있다
- ASL 탐색 및 hole의 발견
 - 이 알고리즘은 ASL의 탐색을 필요로 하지 않는다. 왜냐하면, 무조건 ASL의 가장 처음에 있는 hole에 할당하기 때문이다.
 - 즉, 남아있는 hole들 중에서 가장 크기가 큰 hole에 프로그램을 할당하는 방법이다
- 왜 이러한 알고리즘이 등장하는가?
 - 할당하고 남는 공간이 가장 크게 되는 알고리즘으로, 남는 공간을 또 다른 프로그램에 할당할 수 있는 확률이 가장 높다

Worst-Fit 알고리즘 예

- 현재 메모리의 상황이 아래와 같은 ASL로 표현되고 있을 때, 어떤 프로그램의 수행을 위해 **5K**의 메모리 할당을 요청하면, Worst-Fit 방식을 사용할 경우, 어느 주소에 할당되겠는가?

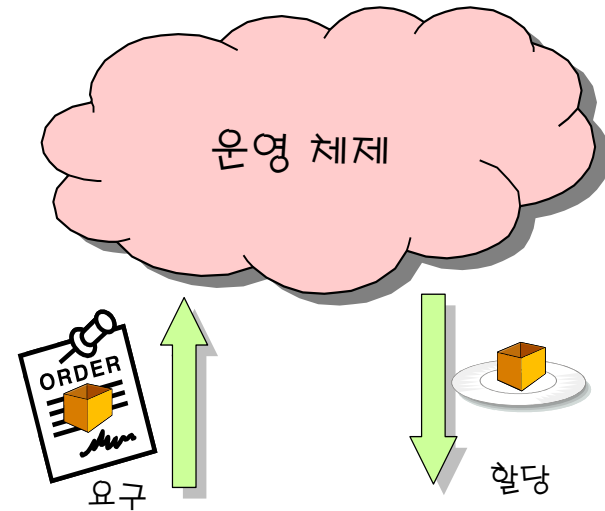
address	size
50000	20K
1300	10K
25000	8K
15000	5K
100	1K



동적 메모리 (dynamic memory allocation)

■ 동적 메모리 할당

- 실행 중에 필요한 메모리를 할당 받는 방법
- 사용이 끝나면 시스템에 반납
- 필요한 만큼만 할당을 받으므로 메모리를 매우 효율적으로 사용
- **malloc()** 계열의 라이브러리 함수를 사용



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

int main(void)
{
    int *p;
    p = (int *)malloc( sizeof(int) );
    ...
}
```

프로그램

동적 메모리 할당의 과정

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

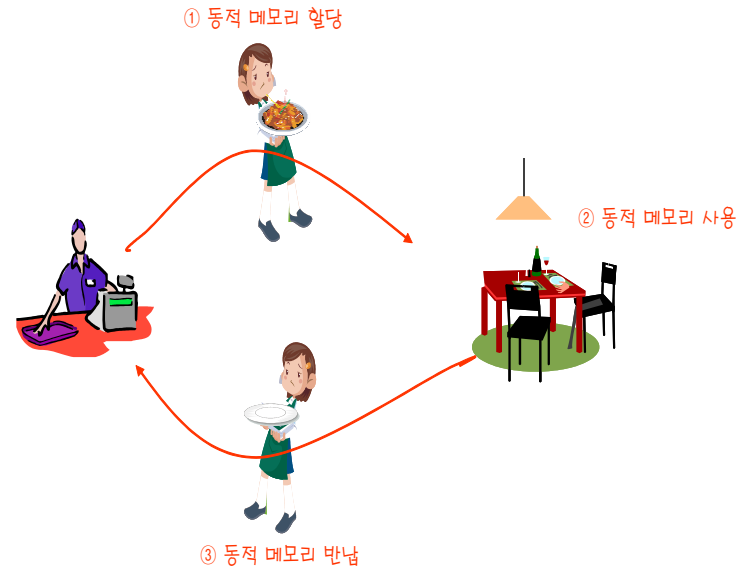
int main(void)
{
    int *pi;    // 동적 메모리를 가리키는 포인터

    pi = (int *)malloc(sizeof(int)); // ① 동적 메모리 할당

    if( pi == NULL )    // 반환값이 NULL인지 검사
    {
        printf("동적 메모리 할당 오류\n");
        exit(1);
    }

    *pi = 100; // ② 동적 메모리 사용
    printf("%d\n", *pi);

    free(pi); // ③ 동적 메모리 반납
    return 0;
}
```



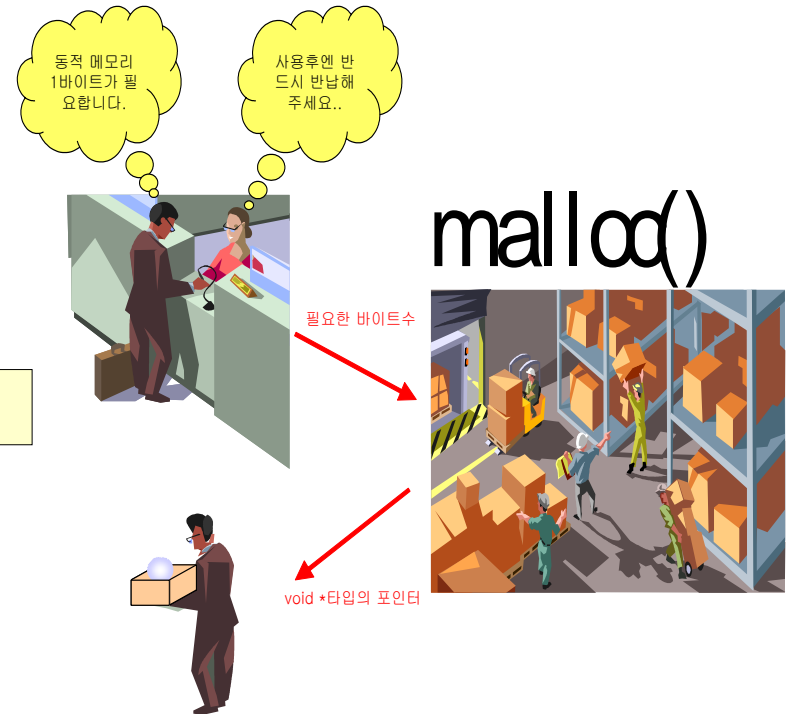
malloc()과 free()

```
void *malloc(size_t size);
```

- malloc()은 바이트 단위로 메모리를 할당
- size는 바이트의 수
- malloc()함수는 메모리 블록의 첫 번째 바이트에 대한 주소를 반환
- 만약 요청한 메모리 공간을 할당할 수 없는 경우에는 NULL값을 반환

```
void free(void *ptr);
```

- free()는 동적으로 할당되었던 메모리 블록을 시스템에 반납
- ptr은 malloc()을 이용하여 동적 할당된 메모리를 가리키는 포인터



malloc1.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  int main( void )
5.  {
6.      char *pc = NULL;
7.
8.      pc = (char *)malloc( sizeof(char) );
9.      if( pc == NULL )
10.     {
11.         printf( "메모리 할당 오류\n" );
12.         exit(1);
13.     }
14.     *pc = 'm';
15.     printf( "*pc = %c\n", *pc );
16.     free( pc );
17.
18.     return 0;
19. }
```

malloc2.c

```
1. // 메모리 동적 할당
2. #include <stdio.h>
3. #include <stdlib.h>

4. int main(void)
5. {
6.     char *pc = NULL;
7.     int i = 0;

8.     pc = (char *)malloc(100*sizeof(char));
9.     if( pc == NULL )
10.    {
11.        printf("메모리 할당 오류\n");
12.        exit(1);
13.    }
14.    for(i=0;i<26;i++)
15.    {
16.        *(pc+i) = 'a'+i;           // 알파벳 소문자를 순서대로 대입
17.    }
18.    *(pc+i) = 0; // NULL 문자 추가

19.    printf("%s\n", pc);
20.    free(pc);
21.    return 0;
22. }
```

abcdefghijklmnopqrstuvwxyz

malloc3.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>

3.  int main(void)
4.  {
5.      int *pi;

6.      pi = (int *)malloc(5 * sizeof(int));

7.      if(pi == NULL){
8.          printf("메모리 할당 오류\n");
9.          exit(1);
10.     }

11.     pi[0] = 100;           // *(pi+0) = 100;와 같다.
12.     pi[1] = 200;           // *(pi+1) = 200;와 같다.
13.     pi[2] = 300;           // *(pi+2) = 300;와 같다.
14.     pi[3] = 400;           // *(pi+3) = 400;와 같다.
15.     pi[4] = 500;           // *(pi+4) = 500;와 같다.

16.     free(pi);
17.     return 0;
18. }
```

malloc4.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>

4.  struct Book {
5.      int number;
6.      char title[10];
7.  };

8.  int main(void)
9.  {
10.     struct Book *p;

11.     p = (struct Book *)malloc(2 * sizeof(struct Book));

12.     if(p == NULL){
13.         printf("메모리 할당 오류\n");
14.         exit(1);
15.     }

16.     p->number = 1;
17.     strcpy(p->title, "C Programming");

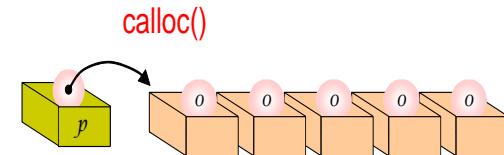
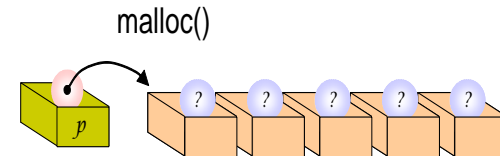
18.     (p+1)->number = 2;
19.     strcpy((p+1)->title, "Data Structure");

20.     free(p);
21.     return 0;
22. }
```

calloc()과 realloc()

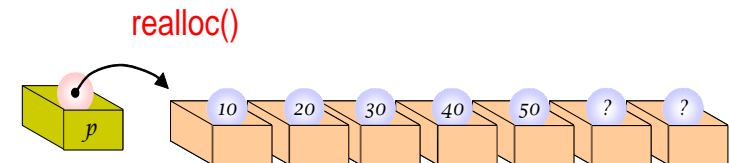
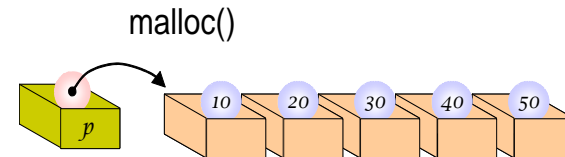
```
void *calloc(size_t n, size_t size);
```

- calloc()은 malloc()과는 다르게 0으로 초기화된 메모리 할당
- 항목 단위로 메모리를 할당(size크기의 항목을 n개 할당)
- (예) int *p;
p = (int *)calloc(5, sizeof(int));



```
void *realloc(void *memblock, size_t size);
```

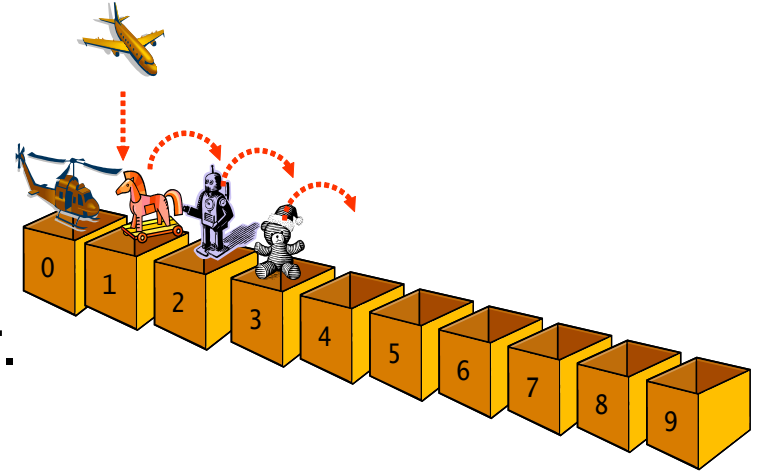
- realloc() 함수는 할당하였던 메모리 블록의 크기를 변경
- (예)
int *p;
p = (int *)malloc(5 * sizeof(int));
p = realloc(p, 7 * sizeof(int));



연결 리스트

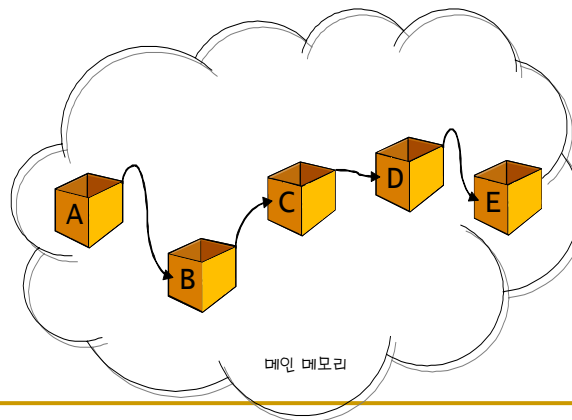
■ 배열(array)

- 장점: 구현이 간단하고 빠르다
- 단점: 크기가 고정된다.
- 중간에서 삽입, 삭제가 어렵다.



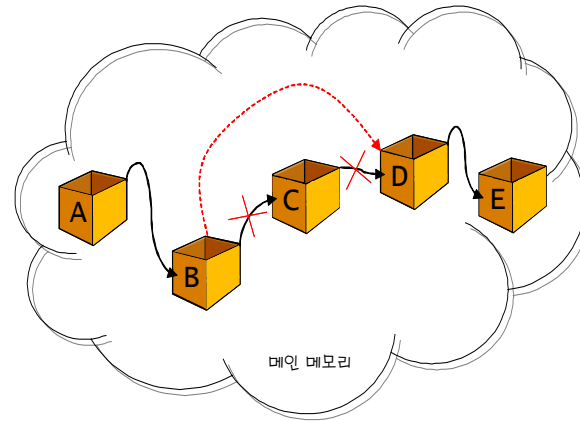
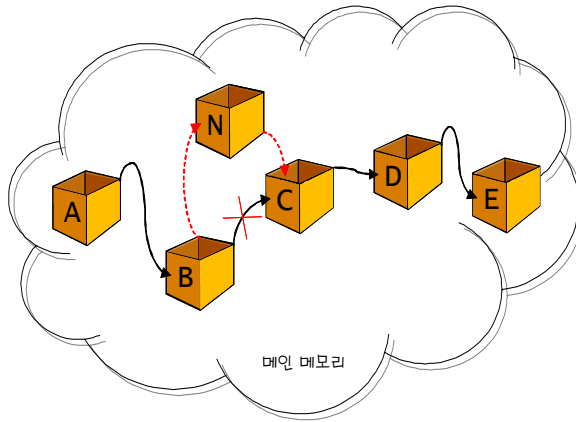
■ 연결 리스트(linked list)

- 각각의 원소가 포인터를 사용하여 다음 원소의 위치를 가리킨다.



연결 리스트의 장단점

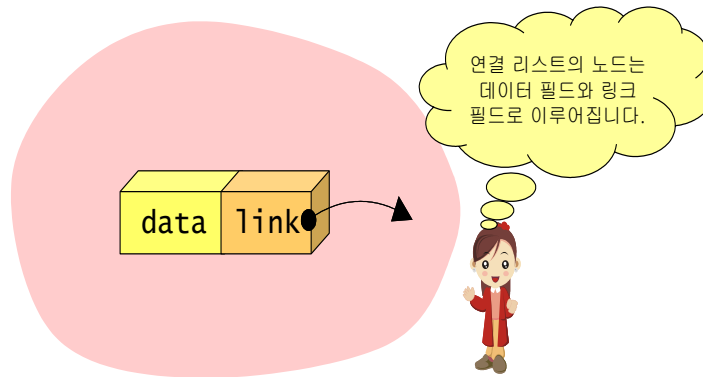
- 중간에 데이터를 삽입, 삭제하는 경우



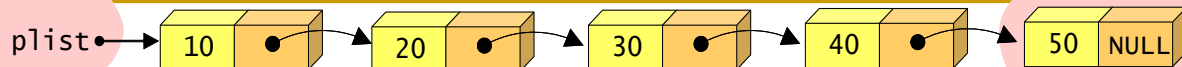
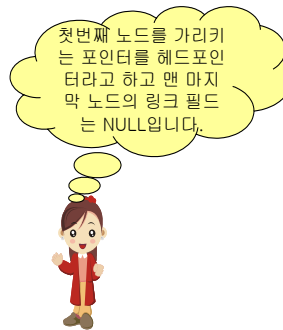
- 데이터를 저장할 공간이 필요할 때마다 동적으로 공간을 만들어서 쉽게 추가
- 구현이 어렵고 오류가 나기 쉽다.

연결 리스트의 구조

- 노드(node) = 데이터 필드(data field)+ 링크 필드(link field)



- 헤드 포인터(head pointer): 첫번째 노드를 가리키는 포인터



자기 참조 구조체

- 자기 참조 구조체(self-referential structure)는 특별한 구조체로서 구성 멤버 중에 같은 타입의 구조체를 가리키는 포인터가 존재하는 구조체

```
// 데이터의 정의
typedef struct data {
    int id;
    char name[20];
    char phone[12];
} DATA;

// 노드의 정의
typedef struct NODE {
    DATA data;
    struct NODE *link;
} NODE;
```

간단한 연결 리스트 생성

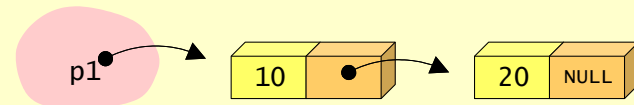
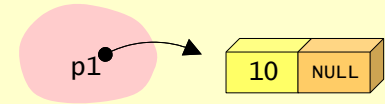
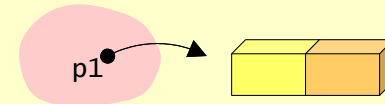
```
NODE *p1;  
p1 = (NODE *)malloc(sizeof(NODE));
```

```
p1->data = 10;  
p1->link = NULL;
```

```
NODE *p2;  
p2 = (NODE *)malloc(sizeof(NODE));  
p2->data = 20;
```

```
p2->link = NULL;  
p1->link = p2;
```

```
free(p1);  
free(p2);
```

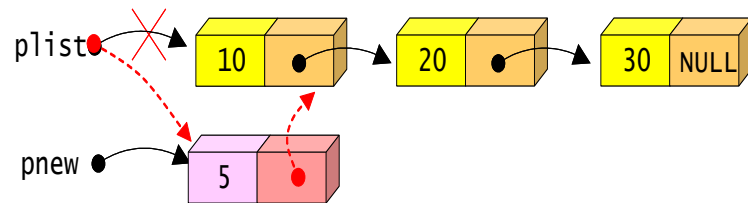


연결 리스트의 삽입 연산

```
NODE *insert_NODE(NODE *plist, NODE *pprev, DATA item);
```

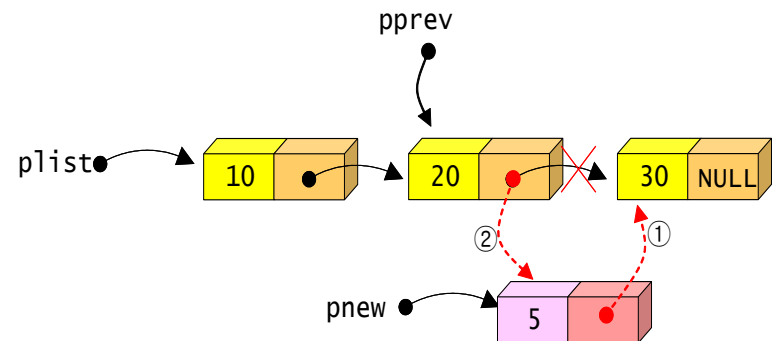
1. 리스트의 처음에 삽입하는 경우

`pnew->link = plist;`
`plist = pnew;`



2. 리스트의 중간에 삽입하는 경우 (순서가 중요)

`pnew->link = pprev->link; // ①`
`pprev->link = pnew; // ②`



연결 리스트의 삽입 연산

```
NODE *insert_node(NODE *plist, NODE *pprev, DATA item)
{
    NODE *pnew = NULL;

    if( !(pnew = (NODE *)malloc(sizeof(NODE))) )
    {
        printf("메모리 동적 할당 오류\n");
        exit(1);
    }

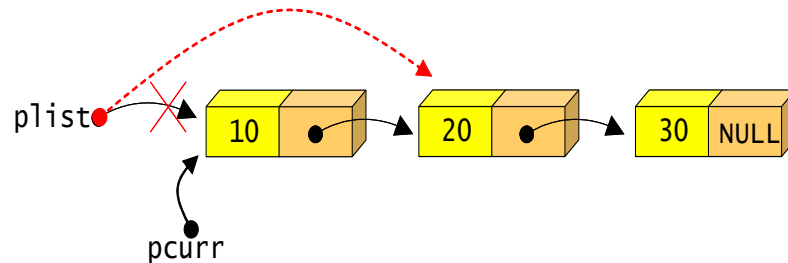
    pnew->data = data;
    if( pprev == NULL )    // 연결 리스트의 처음에 삽입
    {
        pnew->link = plist;
        plist = pnew;
    }
    else                  // 연결 리스트의 중간에 삽입
    {
        pnew->link = pprev->link;
        pprev->link = pnew;
    }
    return plist;
}
```

연결 리스트의 삭제 연산

```
NODE *delete_node(NODE *plist, NODE *pprev, NODE *pcurr);
```

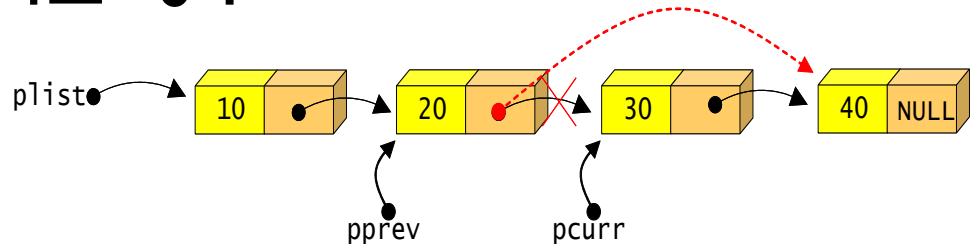
1. 리스트의 처음을 삭제하는 경우

```
plist = pcurr->link;  
free(pcurr);
```



2. 리스트의 중간을 삭제하는 경우

```
pprev->link = pcurr->link;  
free(pcurr);
```



연결 리스트의 삭제 연산

```
NODE *delete_node(NODE *plist, NODE *pprev, NODE *pcurr)
{
    if( pprev == NULL )
        plist = pcurr->link;
    else
        pprev->link = pcurr->link;

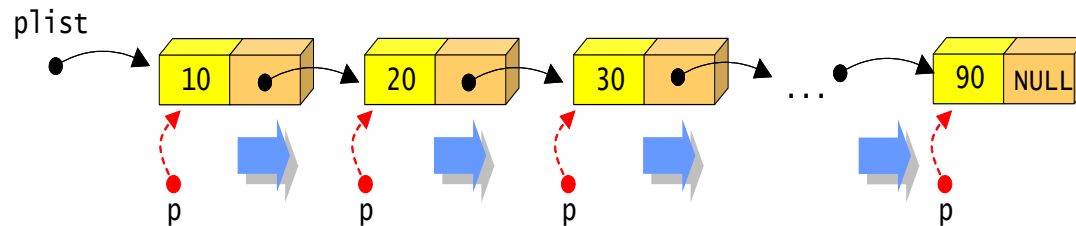
    free(pcurr);
    return plist;
}
```

연결 리스트의 순회 연산

```
void print_list(NODE *plist)
{
    NODE *p;

    p = plist;
    printf("( ");

    while( p )
    {
        printf("%d ", p->data);
        p = p->link;
    }
    printf("\n");
}
```



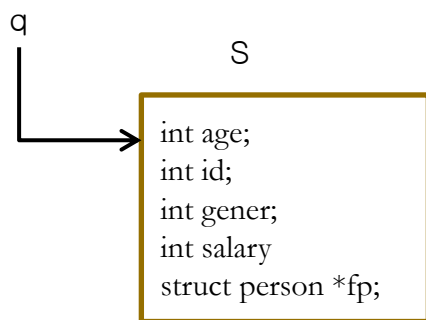
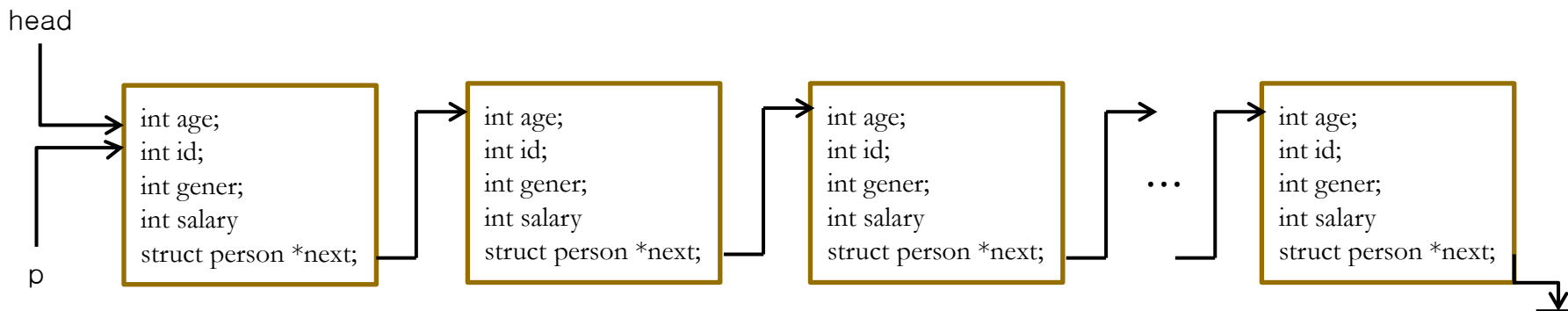
노드의 개수 세기

```
1. int get_length(NODE *plist)
2. {
3.     NODE *p;
4.     int length = 0;
5.
6.     p = plist;
7.
8.     while( p )
9.     {
10.         length++;
11.         p = p->link;
12.     }
13.     printf("리스트의 길이는 %d\n", length);
14.     return length;
15. }
```


합계 구하기

```
1. int get_sum(NODE *plist)
2. {
3.     NODE *p;
4.     int sum = 0;
5.
6.     p = plist;
7.
8.     while( p )
9.     {
10.         sum += p->data;
11.         p = p->link;
12.     }
13.     printf("리스트의 합계는 %d\n", sum);
14.     return sum;
15. }
```

연결리스트 생성



```
struct person {  
    int age;  
    int id;  
    int gener;  
    int salary;  
    struct person *next;  
};
```

```
int main()
{
    struct person {
        int    age;
        int    id;
        int    gender;
        int    salary;
        structperson *next;
    };

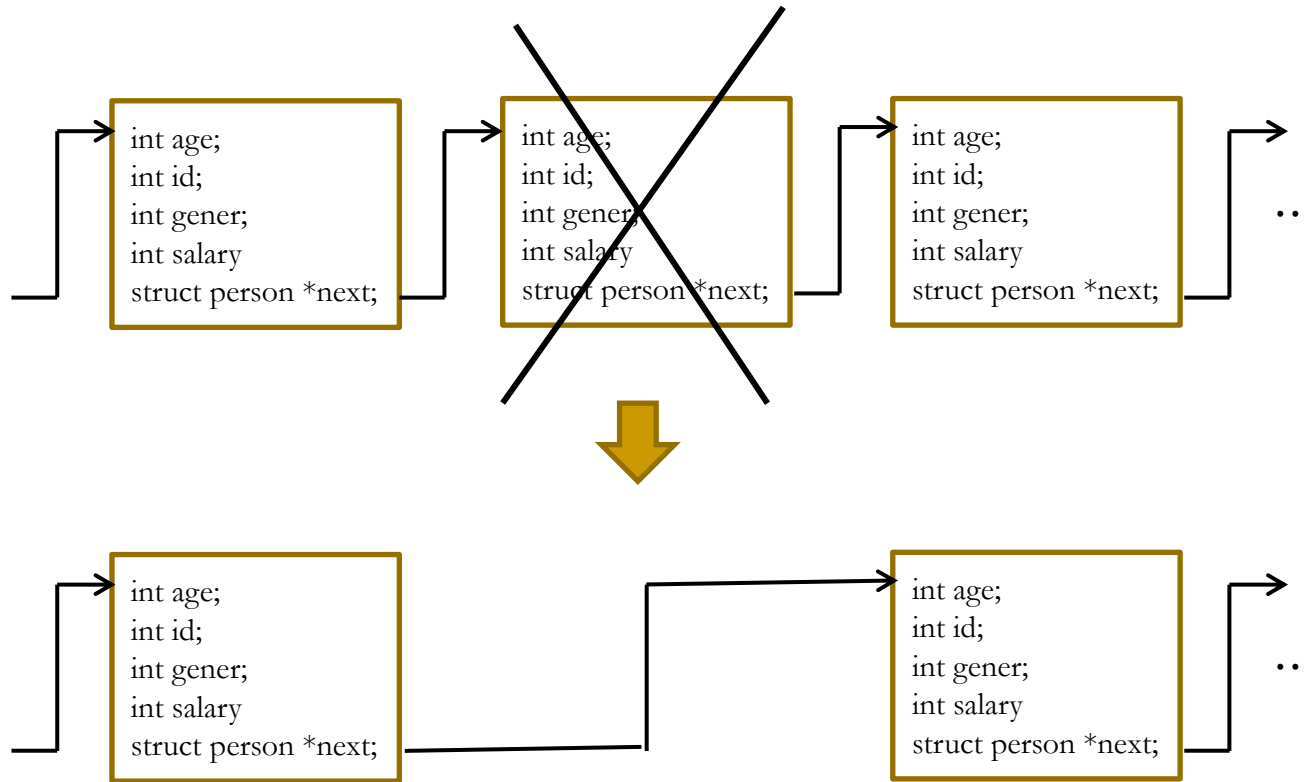
    typedef struct person NODE;

    NODE *head;
    NODE *p;

    head = (NODE *)malloc(sizeof(NODE));
    p = head;

    for(i=1; i<SIZE; i++) {
        p -> next = (NODE *) malloc(sizeof(NODE));
        p = p -> next;
    }
    P -> next = NULL;
    Return 0;
}
```

특정 N 번째 노드 삭제

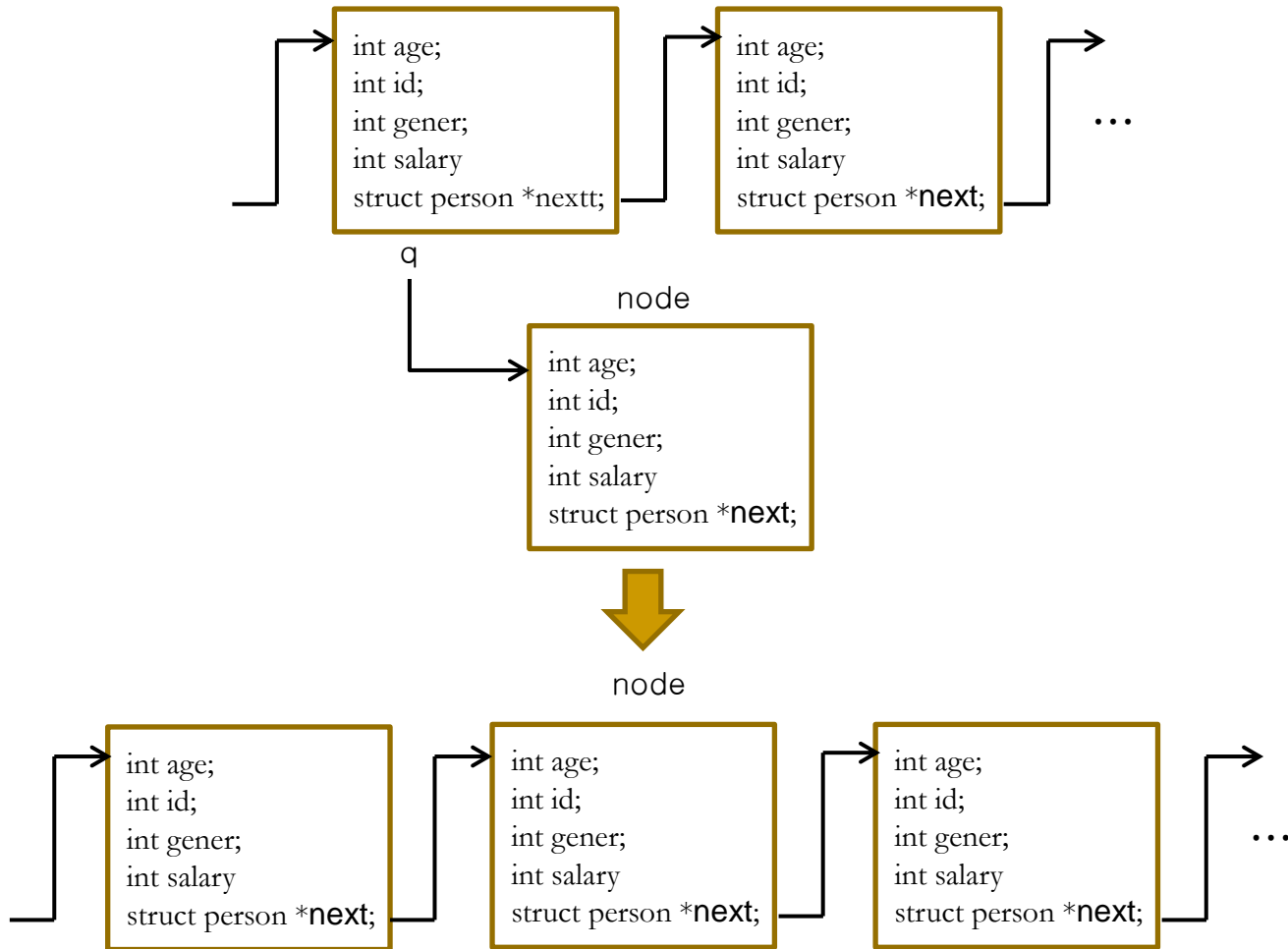


```
int main()
{
    struct person {
        int    age;
        int    id;
        int    gender;
        int    salary;
        structperson *next;
    };

    typedef struct person NODE;

    NODE *head;
    NODE *p;
    // SIZE 개의 node가 lined list로 구성되어 있다고 가정
    p = head;
    for(i=0; i<SIZE; i++) {
        p = p -> next;
        if( i == N-1) break;
    }
    P -> next = p -> next -> next;
    Return 0;
}
```

특정 N 번째 노드와 N+1 번째 노드 사이에 새로운 노드 삽입

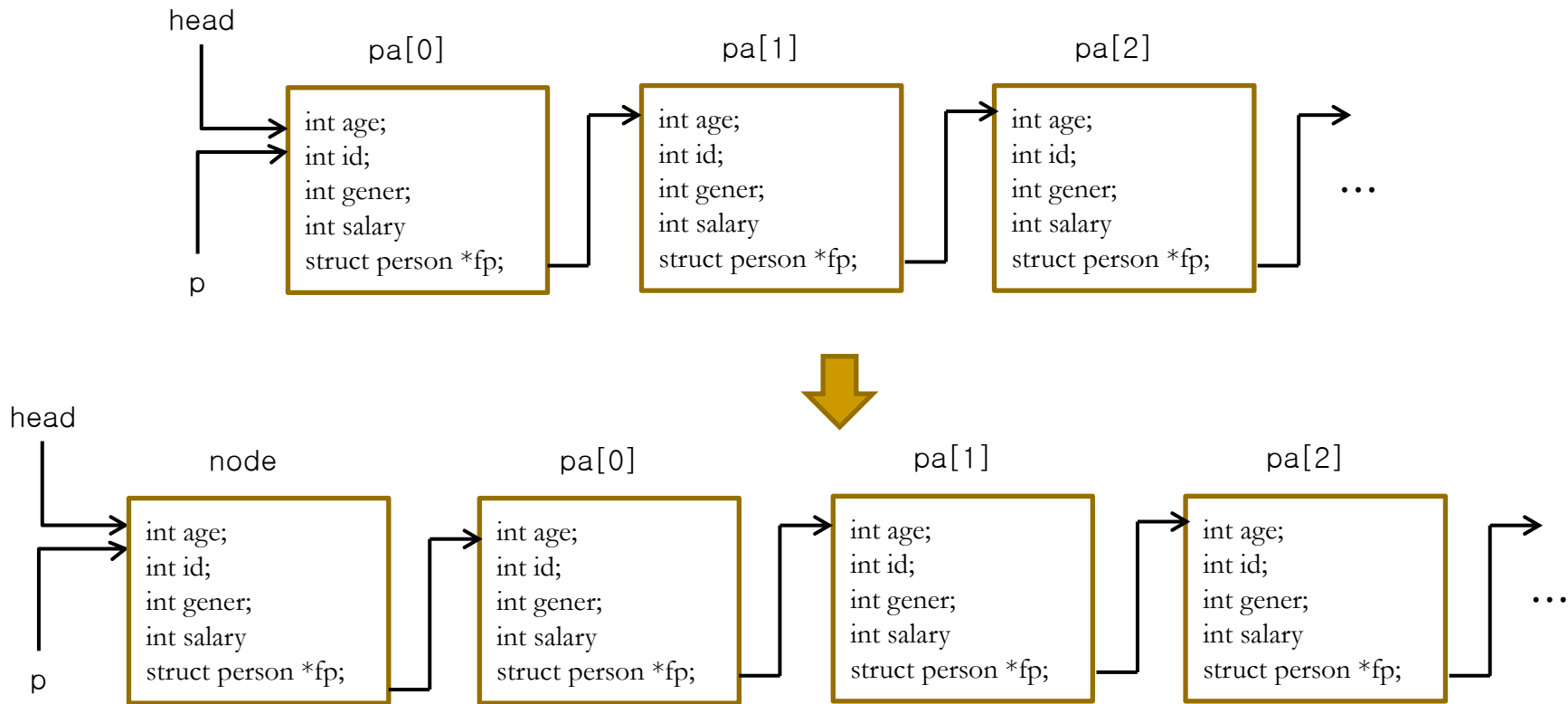


```
int main()
{
    struct person {
        int    age;
        int    id;
        int    gender;
        int    salary;
        structperson *next;
    };

    typedef struct person NODE;

    NODE *head;
    NODE *p;
    NODE *q;
    // SIZE 개의 node가 lined list로 구성되어 있다고 가정
    p = head;
    q = (NODE *) malloc(sizeof(NODE));
    for(i=0; i<SIZE; i++) {
        p = p -> next;
        if( i == N) break;
    }
    q -> next = p -> next;
    P -> next = q;
    Return 0;
}
```

노드를 리스트 맨 앞에 삽입



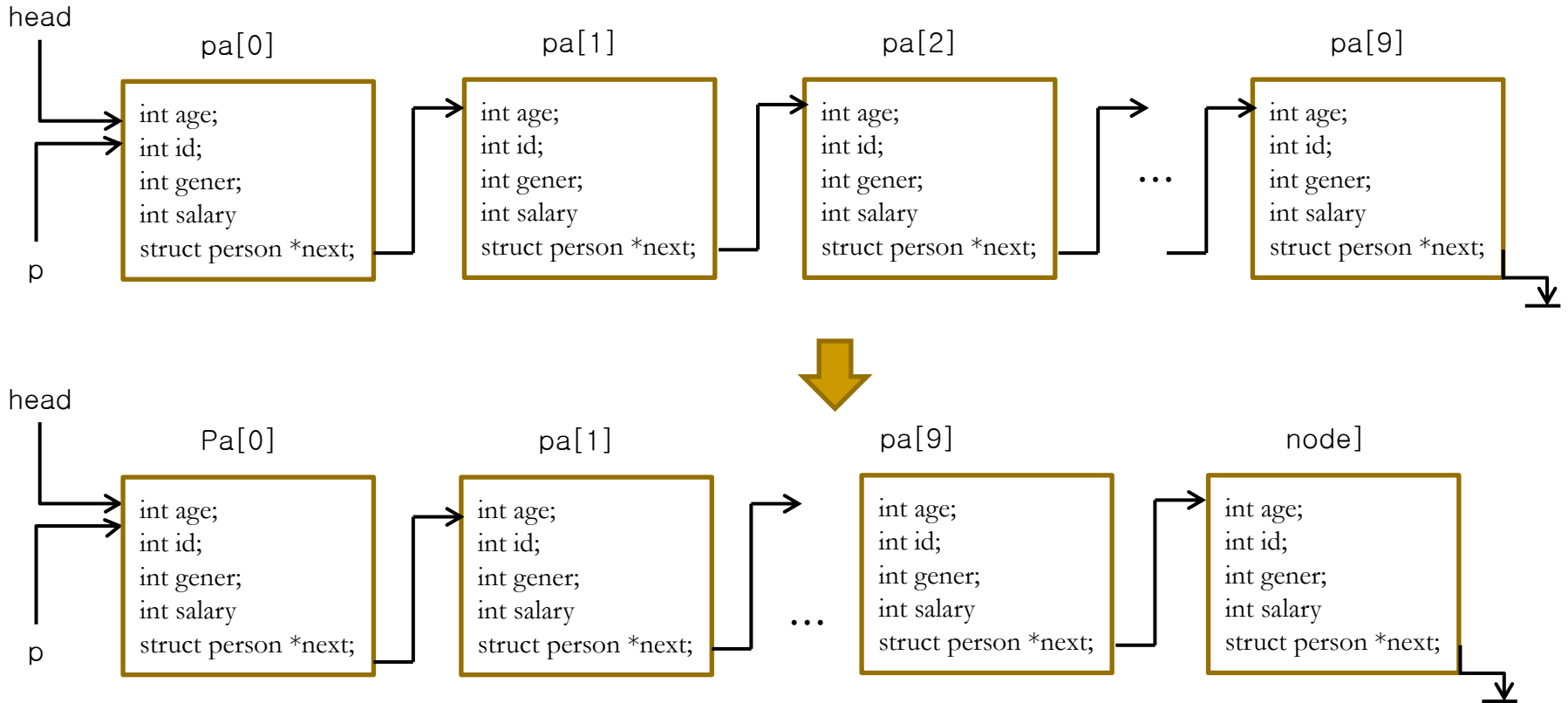

```
int main()
{
    struct person {
        int    age;
        int    id;
        int    gender;
        int    salary;
        structperson *next;
    };

    typedef struct person NODE;

    NODE *head;
    NODE *p;
    NODE *q;
    // SIZE 개의 node가 lined list로 구성되어 있다고 가정

    q = (NODE *) malloc(sizeof(NODE));
    q -> next = head;
    head = q;
    P = head;
    return 0;
}
```

노드를 리스트 맨 뒤에 추가



```
int main()
{
    struct person {
        int    age;
        int    id;
        int    gender;
        int    salary;
        structperson *next;
    };

    typedef struct person NODE;

    NODE *head;
    NODE *p;
    NODE *q;
    // SIZE 개의 node가 lined list로 구성되어 있다고 가정

    q = (NODE *) malloc(sizeof(NODE));
    while(p != NULL) {
        p = p -> next;
    }
    P -> next = q;
    q -> next = NULL;
    return 0;
}
```

질문 ??

