링크드 리스트 (Linked List)

- 동적 메모리 할당
- 단일 링크드 리스트
- 이중 링크드 리스트

본 자료는 Practical C Pointer, 김원선 저, 이한출판사의 자료에 기반해서 만들어졌습니다.

학습 목표

- **동적 메모리 할당**을 이용한 **링크드 리스트**에 대해 살펴본다.
- **자기 참조 구조체**를 활용한 **단일 링크드 리스트**에 대해 알아본다.
- 좀더 안정적인 구현을 위한 **이중 링크드 리스트**에 대해 살펴본다.

1. 단일 링크드 리스트

- 대량의 데이터나 복잡한 형태의 데이터를 다루기 위해, 자료를 체계적으로 조직화하기 위해 **자료 구조(Data Structure)**를 사용한다.
- 널리 사용되는 자료 구조로 **링크드 리스트(Linked List)** 가 있다. 이외도 **스택(Stack), 큐(Queue), 트리(Tree), 그래프(Graph)** 등 다양한 자료 구조가 있다.
- 자료 구조를 구현하기 위해 **동적 메모리 할당**을 사용하는 것이 일반적이며, 링크드 리스트에는 **단일 링크드와 이중 링크드 리스트가 있다.**
- **단일 링크드 리스트란** 이전 노드와 이후 노드를 연결하기 위해 구조체 포인터 멤버를 하나 두어 **다음 노드의 주소를 저장하여** 노드들을 연결하는 구조이다.

1.1 구조체 노드의 생성과 출력

➢ 단일 링크드 리스트를 구현할 구조체 선언문이다.

```
struct A {
        char name[20];
        int age;
        int salary;
        struct A *next;
} *head, *tail ;

head=tail=NULL;   // 노드 생성 전에 초기화
```

1.1 구조체 노드의 생성과 출력

- 노드를 연결하기 위하여 필요한 사항을 하나씩 살펴보자.

- 노드가 처음으로 생성되는 것인지, 혹은 추가되는 것인지에 따라 next 연결이 달라진다. 이를 위해 2개의 구조체 포인터가 필요하다.

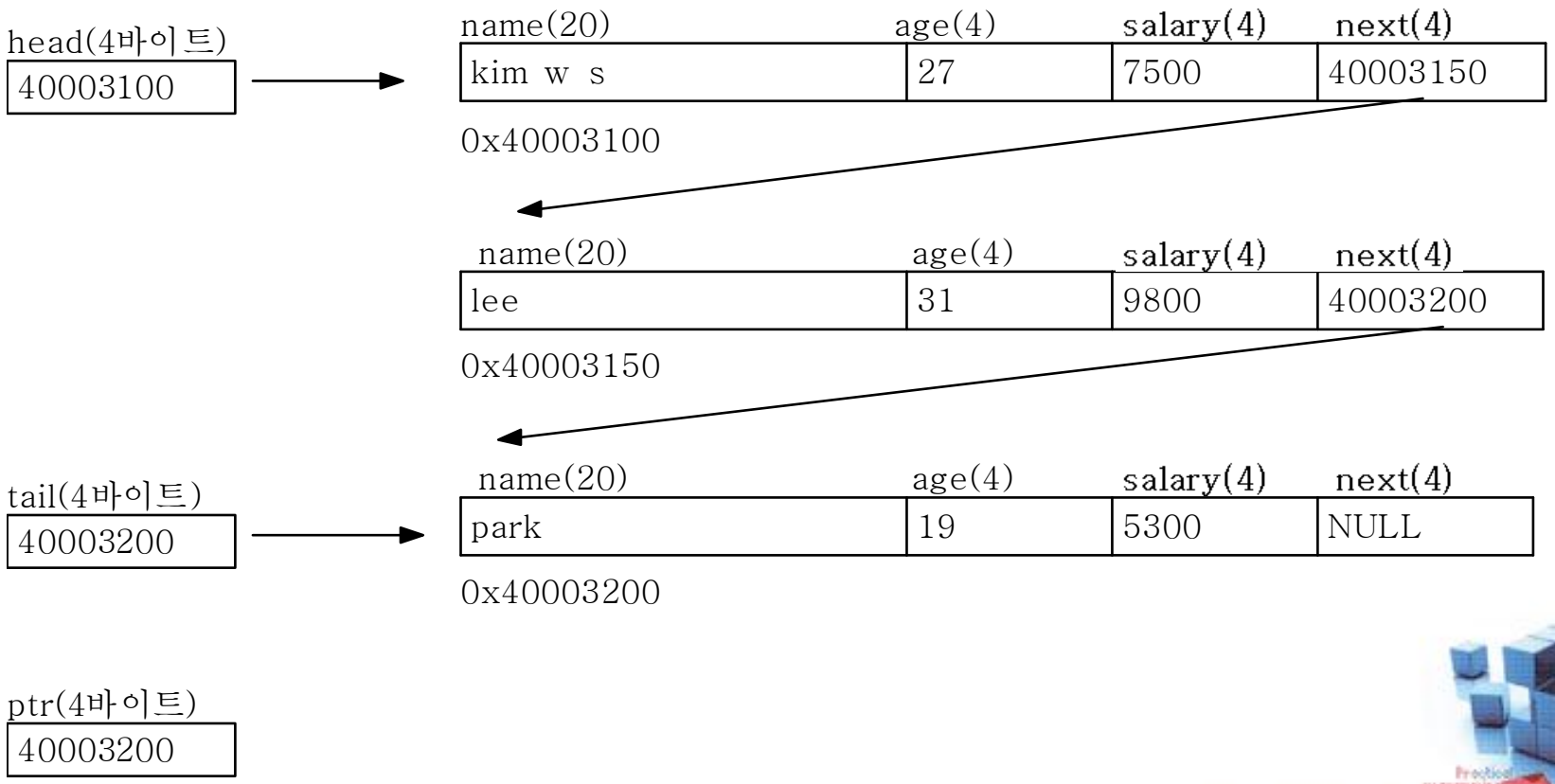
```
struct A *head;   //첫번째 노드의 주소를 저장할 포인터
struct A *tail;   //마지막 노드의 주소를 저장할 포인터

head=tail=NULL;
```

- 포인터 변수를 프로그램 시작시 NULL로 초기화하여 링크드 리스트가 비었음을 알린다. 노드가 생성되고 추가되면 head와 tail은 서로 다른 노드 주소를 갖게 되며, 이를 이용하여 노드간의 주소 연결을 갖게 된다.

1.1 구조체 노드의 생성과 출력

➢ 구조체 노드 3개가 할당된 메모리 구조이다.



1.1 구조체 노드의 생성과 출력

- 단일 링크드 리스트는 head부터 next 포인터로 노드 하나씩 연결되어 tail까지 연결된다.

➢ 결과

```
성명 ? kim w s
나이 ? 27
월급 ? 7500

성명 ? lee
나이 ? 31
월급 ? 9800

성명 ? park
나이 ? 19
월급 ? 5300

성명 ? sun
나이 ? 43
월급 ? 12000

성명 ? end

Node List
name:kim w s , age: 27, salary: 7500
name:lee , age: 31, salary: 9800
name:park , age: 19, salary: 5300
name:sun , age: 43, salary: 12000
```

```
void input(void) //노드 추가
{
    struct A *ptr;

    while(1)
    {
        if((ptr=(struct A *)malloc(sizeof(struct A)))==NULL) //메모리 요청
        {
            printf("Memory Allocation Error ₩n");
            exit(1);
        }

        printf("₩n성명 ? ");
        gets(ptr->name);
        if(!strcmp(ptr->name, "end"))
            break;
        printf("나이 ? ");
        scanf("%d", &ptr->age);
        printf("월급 ? ");
        scanf("%d%*c", &ptr->salary);
        ptr->next=NULL;
```

```
            if(head==NULL)
                head=tail=ptr; // 첫번째 노드인 경우
            else
            {
                tail->next=ptr; // 노드가 추가되는 경우
                tail=ptr;
            }
        }

        free(ptr);
}
```

1.2 구조체 노드의 검색

```
void find(void) // 노드 검색
{
    struct A *ptr;
    int t_salary, found =1 ;

    printf("입력한 월급보다 큰 값 검색 ? ");
    scanf("%d", &t_salary);

    ptr=head;
    while(ptr)
    {
        if(ptr->salary>=t_salary)
        { //참이면 출력
            printf("name:%s , age: %d, salary: %d ₩n",
            ptr->name, ptr->age, ptr->salary);
        }
        ptr=ptr->next; //다음 노드 주소로 이동
    }
}
```

1.3 구조체 노드의 삭제

- 삭제하려는 노드가 head, tail 또는 **중간 노드**일 수 있다. 각각 처리하는 방법이 다르다.
- 노드를 삭제하기 위해 **삭제할 노드를 검색**한다. 검색된 노드는 free() 함수로 해제한다.
- 다음과 같은 포인터 변수를 추가할 것이다.

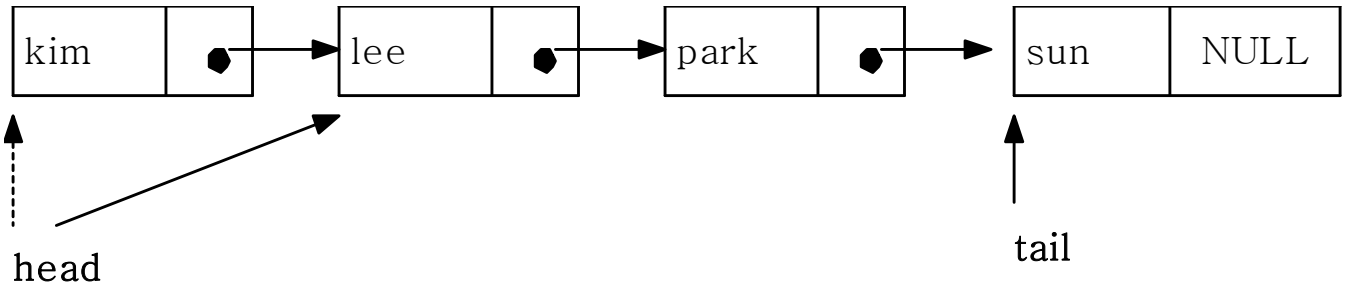
  temp : 삭제할 노드를 가리키는 포인터 변수
  prev : temp 의 이전 노드를 가리키는 포인터 변수

1.3 구조체 노드의 삭제

➢ 삭제하려는 노드가 head인 경우

① head를 바꿔줘야 한다.

```
if (temp==head) {
    head=temp->next;    // 다음 노드를 head로 설정
}
```

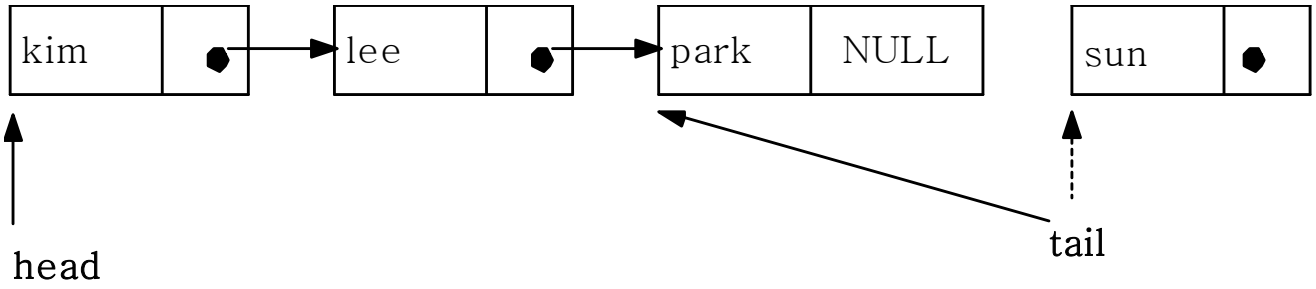


1.3 구조체 노드의 삭제

➢ 삭제하려는 노드가 tail인 경우

② tail을 바꿔줘야 한다.

```
else if(temp==tail) {
    prev->next=NULL;
    tail=prev;
}
```

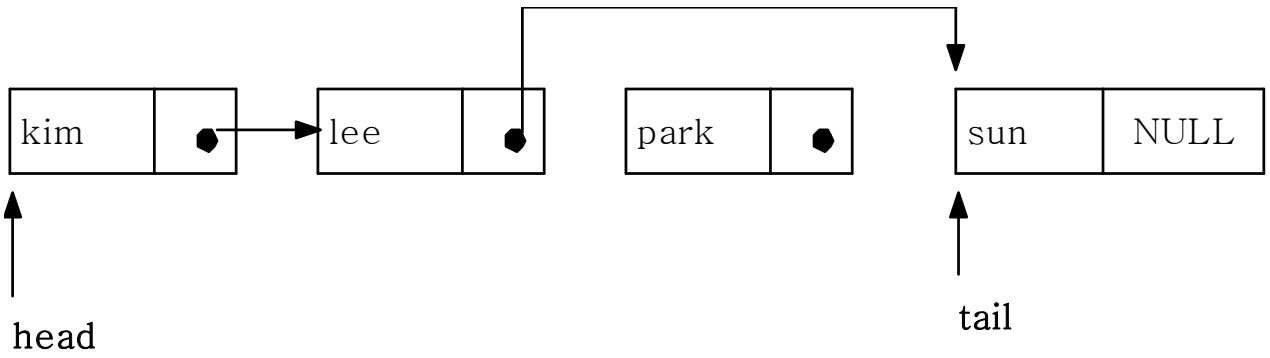


1.3 구조체 노드의 삭제

➢ 삭제하려는 노드가 중간 노드인 경우

③ 삭제하려는 노드의 이전 노드와 다음 노드를 연결해줘야 한다.

```
else
   prev->next=temp->next;
```



1.3 구조체 노드의 삭제

- 다음은 4개의 노드가 존재하는 경우 중간 노드를 삭제하는 경우이다.
- 검색된 노드는 삭제되고 남겨진 노드를 출력하고 있다.

➢ 결과

```
삭제할 성명 ? park
name:park , age: 19, salary: 5300
출력된 노드를 삭제할까요 ? (y/n) y
노드 삭제...

Node List
name:kim w s , age: 27, salary: 7500
name:lee , age: 31, salary: 9800
name:sun , age: 43, salary: 12000
```

1.4 할당된 구조체 노드 모두 해제

- Heap에 할당된 노드를 모두 해제하려고 한다.
- head부터 시작하여 next 포인터가 NULL일 때까지 노드를 하나씩 해제한다.
- 링크드 리스트에 연결된 모든 노드 제거

```
void node_free(void) // 전체 노드 삭제 및 해제
{
    struct A *ptr, *x;

    ptr=head;
    if(ptr==NULL)
    {
        printf("삭제될 노드가 없습니다. ₩n");
        return ;
    }
```

1.4 할당된 구조체 노드 모두 해제

```
        printf("₩n모든 node가 메모리에서 제거됩니다. ₩n");
        while(ptr)
        {
            x=ptr;
            ptr=ptr->next; //다음 노드 주소로 이동
            free(x); //노드 영역 해제
        }

        head=tail=NULL;
        output(); //노드 List가 비어 있음을 확인한다.
}
```

2. 이중 링크드 리스트

- 단일 링크드 리스트로 구성한 경우 **포인터 멤버의 값이 손상되면** 그 이후의 모든 노드를 참조할 수 없다.
- 따라서 자료 구조를 좀 더 안정적으로 구현하기 위해 **이중 링크드 리스트(Doubly-Linked List)**를 사용할 수 있다.
- 이중 링크드 리스트는 포인터 멤버를 하나 더 사용하여 **이전 노드의 주소까지 저장하는 구조**이다.
- 따라서 다음 노드를 가리키는 포인터가 손상을 입더라도, 이전 노드를 가리키는 포인터가 존재하므로 **역으로 노드를 참조할 수 있게 된다.**

2. 이중 링크드 리스트

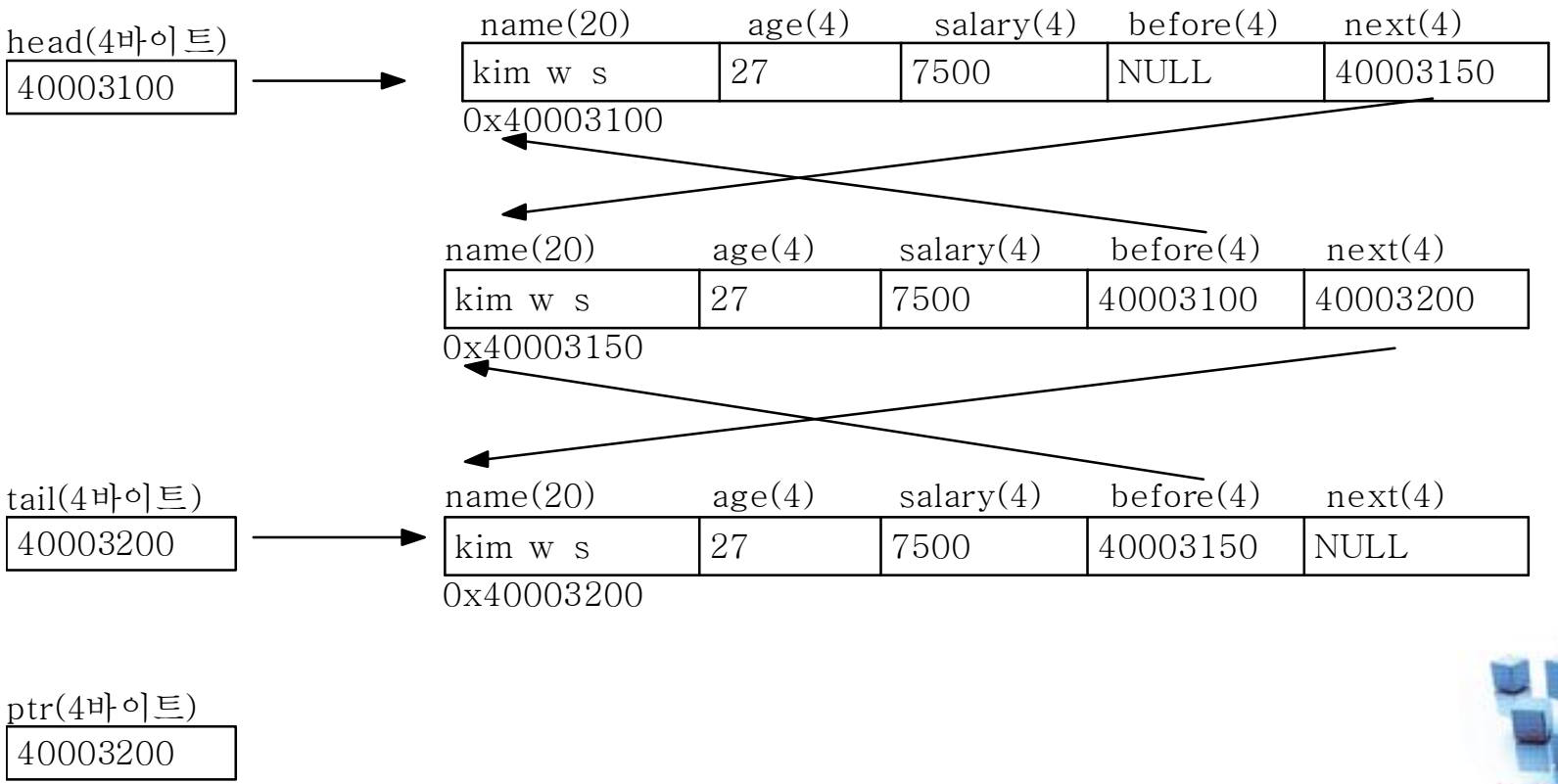
➢ 이중 링크드 리스트를 구현할 구조체 선언문이다.

```
struct A {
        char name[20];
        int age;
        int salary;
        struct A *before, *next;
} *head, *tail ;

head=tail=NULL;   // 노드 생성 전에 초기화
```

2. 이중 링크드 리스트

➢ 이중 링크드 리스트를 구현할 구조체 노드의 메모리 할당 구조이다.



2. 이중 링크드 리스트

➢ 이중 링크드 리스트는 노드 추가 시 이전 노드의 주소를 저장해준다.

```
if (head==NULL) {
        head=tail=ptr;          // 첫번째 노드인 경우
}
else {
        ptr->before=tail;       // 이전 노드 연결
        tail->next=ptr;         // 이후 노드 연결
        tail=ptr;
}
```

2. 이중 링크드 리스트

- 결과에서 보듯이
- head에서 tail까지, 또는
- tail에서 head까지 양방향 접근이 된다.

```
성 명 ? kim w s
나이 ? 27
월 급 ? 7500

성 명 ? lee
나이 ? 31
월 급 ? 9800

성 명 ? park
나이 ? 19
월 급 ? 5700

성 명 ? sun
나이 ? 43
월 급 ? 12000

성 명 ? end

Node List head -> tail
name:kim w s , age: 27, salary: 7500
name:lee , age: 31, salary: 9800
name:park , age: 19, salary: 5700
name:sun , age: 43, salary: 12000

Node List tail -> head
name:sun , age: 43, salary: 12000
name:park , age: 19, salary: 5700
name:lee , age: 31, salary: 9800
name:kim w s , age: 27, salary: 7500
Press any key to continue_
```

요약

- 구조체(노드)간 연결을 위해 자기 참조 구조체를 사용한다. 이전 노드와 이후 노드의 연결을 위한 구조체 포인터 멤버를 하나 두어 다음 노드의 주소를 저장하는 구조를 **단일 링크드 리스트라** 한다.

- **이중 링크드 리스트는** 단일 링크드 리스트에 포인터 멤버를 하나 더 사용하여 이전 노드의 주소까지 저장한다. 따라서 다음 노드를 가리키는 포인터가 손상되라도, 이전 노드를 가리키는 포인터가 존재하므로, 역방향으로 노드를 참조할 수 있게 된다.