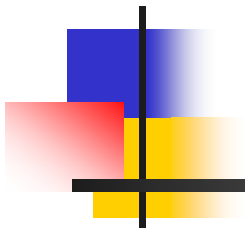
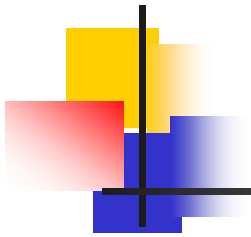


두번째 강의

연결 리스트

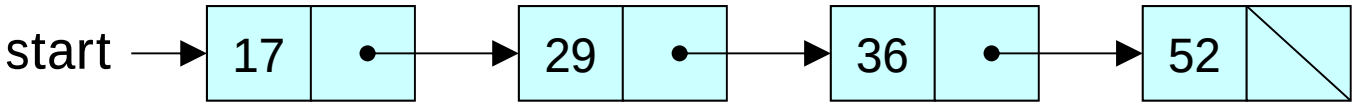


목차

- 연결 리스트**(Linked List)**의 정의
- 연결 리스트에 적용 가능한 연산
- 이중 연결 리스트
- 실습
- 스스로 프로그래밍
- 첨삭 지도

1. 연결 리스트의 정의

- 정의
  - 자기 참조 구조체(노드)들의 연결
  - 첫 번째 노드에 대한 포인터만 유지
  - 이후 노드들은 구조체의 next 포인터를 통하여 참조
  - 마지막 노드의 next 포인터는 NULL로 설정



- 자기 참조 구조체의 구성

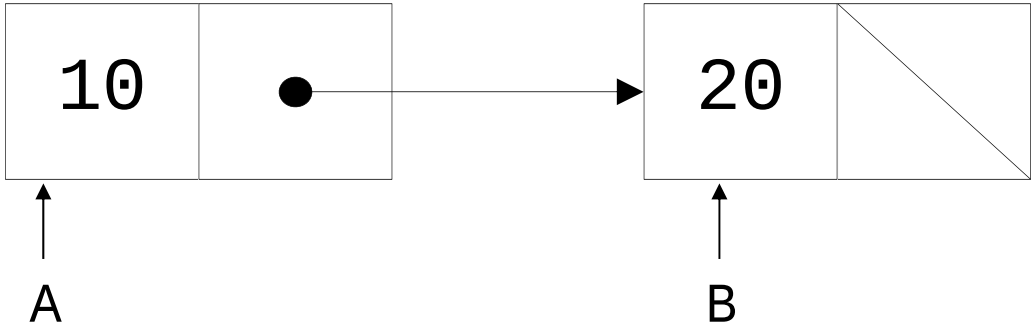
```
struct node {
        int data;
        struct node *next;
};
```

예: 두 개의 노드를 연결

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

struct node *A, *B;

A = (struct node *) malloc(sizeof(struct node));
A->data = 10;
B = (struct node *) malloc(sizeof(struct node));
B->data = 20;

A->next = B;
B->next = NULL;
```

At <stdio.h>

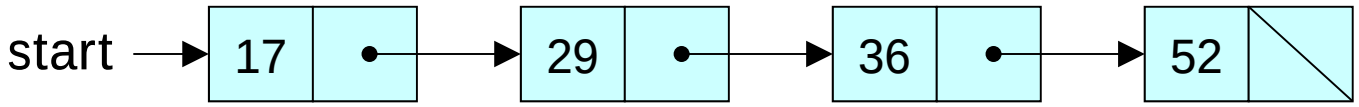
#define NULL ((void *) 0)



2. 연결 리스트에 적용 가능한 연산

- 연결 리스트의 순회
- 연결 리스트에서 노드 추가
  - 앞에 추가: Stack 방식
  - 뒤에 추가: Queue 방식
  - 중간에 추가
- 연결 리스트에서 노드 삭제
  - 제일 앞의 노드 삭제: Stack/Queue 방식
  - 다른 위치의 노드 삭제

2.1 연결 리스트의 순회



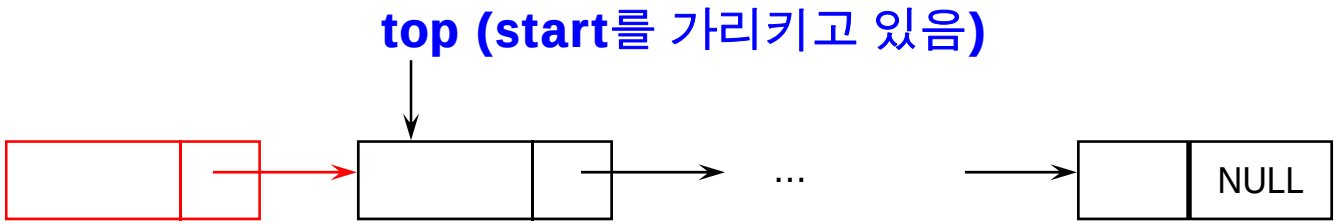

```
struct node *ptr;
int length = 0;

for (ptr = start; ptr != NULL; ptr = ptr->next)
    printf("%d₩n", ptr->data);

for (ptr = start; ptr != NULL; ptr = ptr->next)
    length += 1;

printf("노드 수 = %d₩n", length);
```

배열과 연결 리스트의 비교

```
int A[20], i;
int sum = 0;

for (i = 0; i < 20; i++)
    sum += A[i];
```

```
struct node *A, *ptr;
int sum = 0;

for (ptr = A; ptr != NULL;
    ptr = ptr->next)
        sum += ptr->data;
```

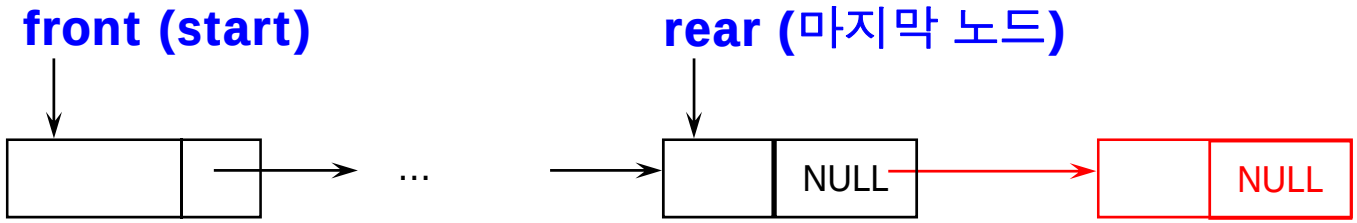
2.2 제일 앞에 노드 추가(Stack 방식)




```
void push(int data)
{  // 스택 top에 새로운 item 추가
   struct node *temp  =
          (struct node *) malloc(sizeof(struct node));

   temp→data =  data;
   temp→next  =  top;
   top = temp;
}
```

top은 전역 변수이며,
초기값은 NULL로 설정

제일 뒤에 노드 추가(Queue 방식)

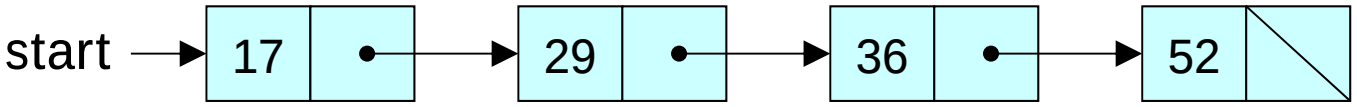



```
void addq(int data)
{  // 큐의 rear에 새로운 element 추가
   struct node *temp =
         (struct node *) malloc(sizeof(struct node));
   temp→data =  data;
   temp→next  =  NULL;
   if (front != NULL)   rear→next  =  temp;
   else front = temp;
   rear = temp;
}
```

front와 rear는 전역 변수이며,
front의 초기값은 NULL로 설정

연결 리스트 연산 – 리스트 생성

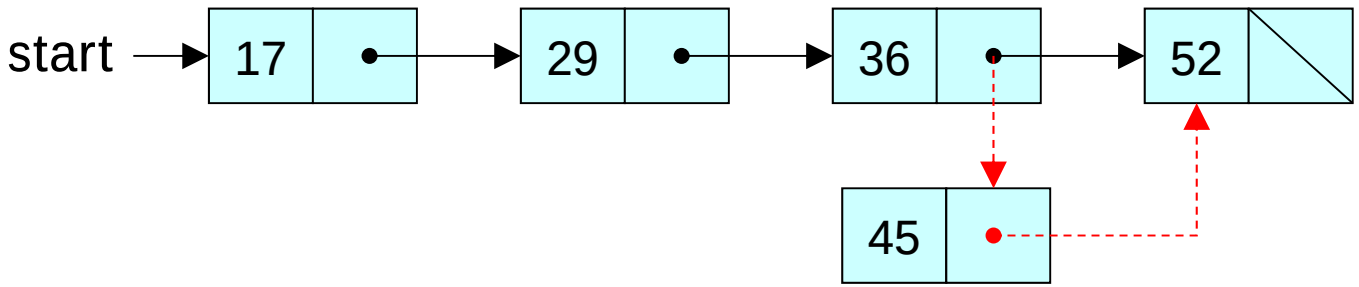
```
struct node *start = NULL, *temp;
int digit;

scanf("%d", &digit);
while (digit != -1) {
    push(&start, digit);        // push에서 start를 인자로 받음
    scanf("%d", &digit);
}
```



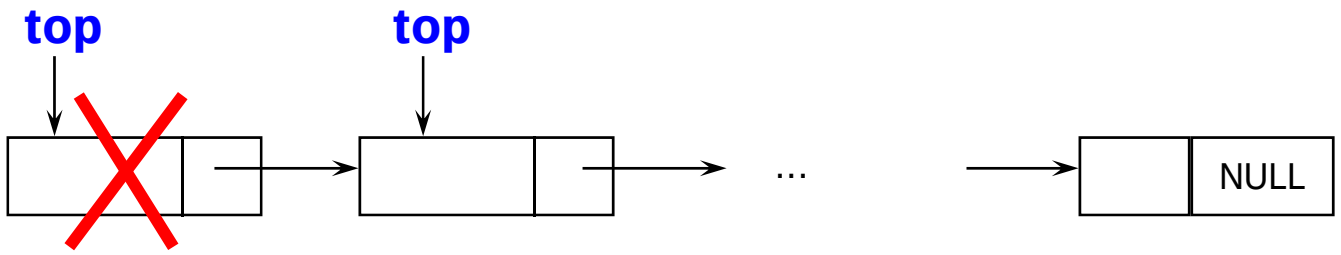
리스트 중간에 노드 추가

- 추가할 위치의 앞 노드를 알아야 함: **36** 다음에 **45**를 추가

```
struct node *ptr, *temp;

temp = (struct node *) malloc(sizeof(struct node));
temp->data = 45;
for (ptr = start; ptr->data != 36; ptr = ptr->next) ;

temp->next = ptr->next;
ptr->next = temp;
```



2.3 리스트에서 노드 삭제



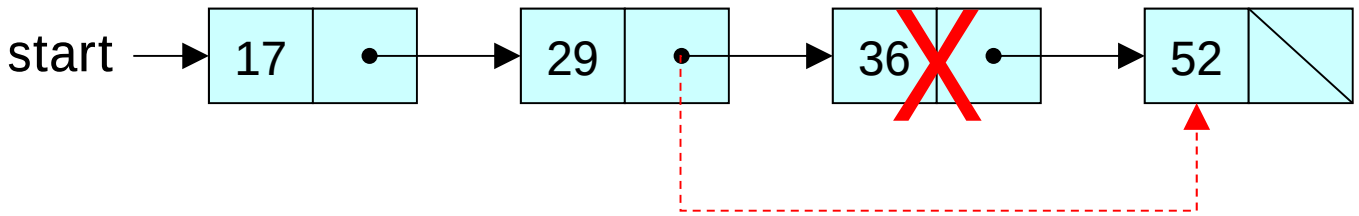

```
// 경우 1: 리스트의 첫번째 노드를 삭제하고 값을 return
int pop()
{
    struct node *temp = top;

   int item = temp→data;
    top = temp→link;
    free(temp);
    return item;
}
```

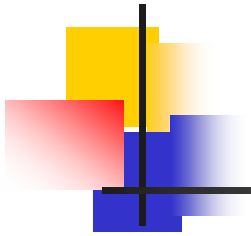
top이 NULL인 경우도 고려할 것!
Queue일 경우, top을 front로 변경.

리스트의 중간 노드를 삭제

- 삭제할 노드의 앞 노드까지 이동: **36** 노드를 삭제

```
struct node *ptr, *target;

for (ptr = start; ptr->next->data != 36; ptr = ptr->next) ;

target = ptr->next;
ptr->next = target->next;
free(target);
```





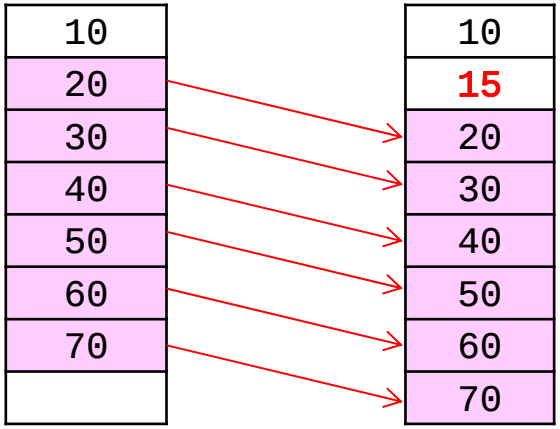
배열과 연결 리스트의 비교(1)

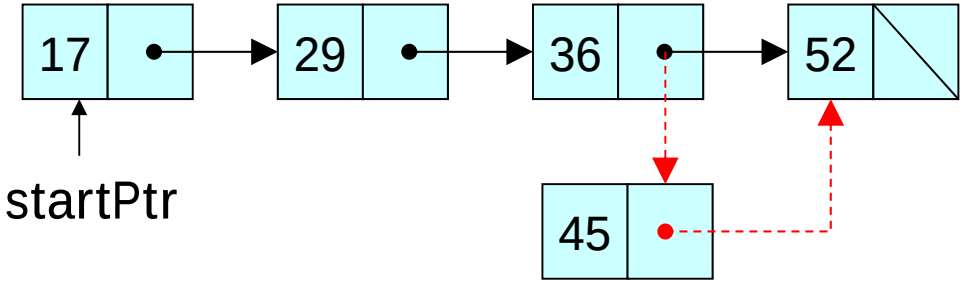
- 저장 방식의 차이
  - 배열: int A[4]; ← 메모리의 인접한 곳에 저장
    - 다음 데이터에 대한 주소를 알 필요 없음
  - 연결 리스트: struct node에 대한 네 번의 malloc
    - 각 노드들은 메모리의 여러 곳에 나누어 저장
    - next 포인터를 이용하여 다음 노드의 주소 유지

- 메모리 사용 측면
  - 저장될 데이터의 수를 안다면 배열이 효과적
  - 데이터의 수를 모를 경우, 연결 리스트가 유리
    - 새로 데이터가 입력될 때마다 malloc 실행 후 연결

배열과 연결 리스트의 비교(2)

- 정렬된 데이터의 유지
  - 배열:
    - 데이터가 추가될 때 기존 데이터의 위치 변경 가능
    - 이진 검색 가능
  - 연결 리스트
    - 기존 데이터의 위치 변경은 발생하지 않음
    - 이진 검색은 불가능





3. 이중 연결 리스트

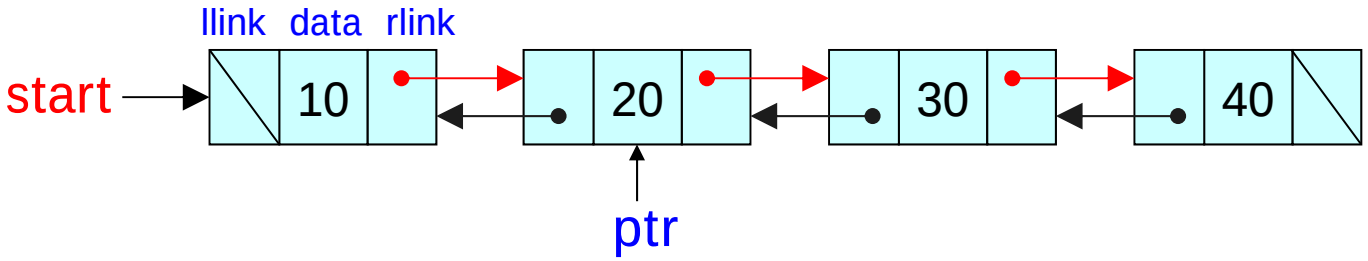
- 이중 연결 리스트(Doubly linked list)란?
  - 한 노드에 두 개의 link가 저장

```
struct node {
        struct node *llink;          // 이전 노드를 포인트
        int          data;
        struct node  *rlink;         // 다음 노드를 포인트
};
```

- 이중 연결 리스트는 양 방향으로 이동 가능
  - 단일 연결 리스트의 경우, 한 방향으로만 이동 가능

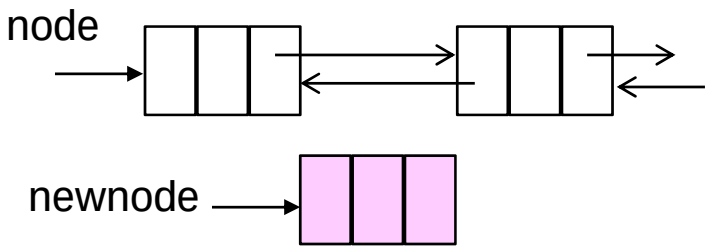
이중 연결 리스트의 종류

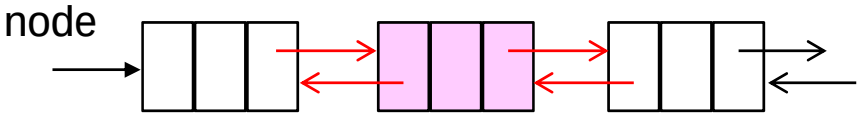
- 처음 노드의 **llink**와 마지막 노드의 **rlink**는 **NULL**



- ptr = ptr→llink→rlink = ptr→rlink→llink

이중 연결 리스트에 노드 추가

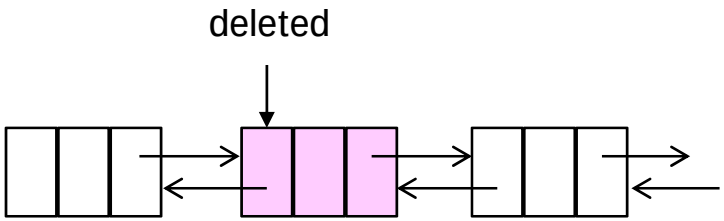




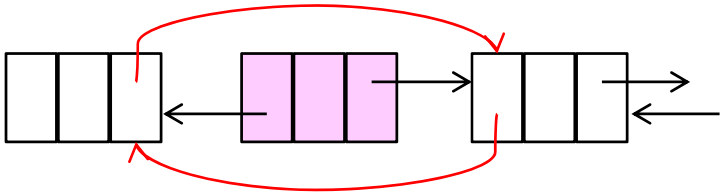

```
void dinsert(struct node *node,
struct node *newnode)
{
// newnode를 node의 오른쪽에
   추가
   newnode→llink = node;
   newnode→rlink = node→rlink;
   node→rlink→llink = newnode;
   node→rlink = newnode;
}
```

다른 문제들:
(1) newnode를 node의 왼쪽에 추가
(2) 원형 리스트가 아닌 이중 연결 chain의 왼쪽과 오른쪽에 노드 추가

이중 연결 리스트에서 노드 삭제



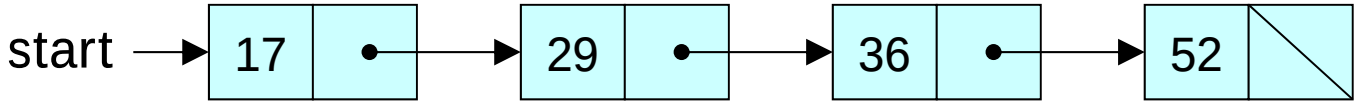

```
void ddelete(struct node *deleted)
{
    deleted→llink→rlink = deleted→rlink;
    deleted→rlink→llink = deleted→llink;
    free(deleted);
}
```



다른 문제들:
(1) deleted 노드의 이웃 노드 삭제
(2) 원형 리스트가 아닌 이중 연결 chain 에서 노드 삭제

연결 리스트 – 실습(1)

- 네 개의 노드를 동적으로 할당받은 다음, 아래와 같은 연결 리스트를 만들어 보라.



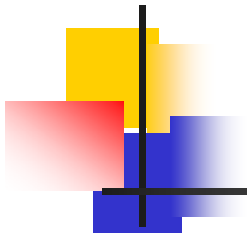
- **start**를 입력으로 받아, 연결 리스트의 모든 원소들을 출력하는 함수 **void print_list(struct node *)**를 작성하라.

연결 리스트 – 실습(2)

- 연결 리스트의 앞에 데이터를 추가하는 함수 **void insert_front(struct node **, int)**를 작성하라.

- **rand()** 함수를 이용하여 **1**부터 **1000**사이의 정수를 무작위로 **30**개 생성한 다음, **insert_front()** 함수를 이용하여 연결 리스트의 앞에 차례대로 추가하라.

- **print_list()**를 이용하여 연결 리스트의 내용을 출력하라.



연결 리스트 – 실습(3)

- 연결 리스트의 뒤에 데이터를 추가하는 함수 **void insert_rear(struct node **, struct node **, int)**를 작성하라.

- **rand()** 함수를 이용하여 **1**부터 **1000**사이의 정수를 무작위로 **30**개 생성한 다음, **insert_rear()** 함수를 이용하여 연결 리스트의 뒤에 차례대로 추가하라.

- **print_list()**를 이용하여 연결 리스트의 내용을 출력하라.

연결 리스트 – 실습(4)

- 오름차순으로 데이터를 추가하는 함수 **void sort_order(struct node **, int)**를 작성하라.

- **rand()** 함수를 이용하여 **1**부터 **1000**사이의 정수를 무작위로 **30**개 생성한 다음, **sort_order()** 함수를 이용하여 연결 리스트에 차례대로 추가하라.

- **print_list()**를 이용하여 연결 리스트의 내용을 출력하라.

- 이후 사용자에게 정수 하나를 입력받은 다음, 그 수를 연결 리스트에서 삭제하라. 다시 **print_list()**를 이용하여 연결 리스트의 내용을 출력하라.