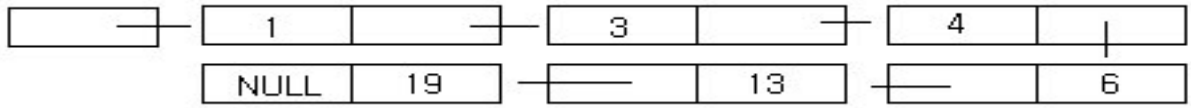
# [ R E P O R T ]

정보통신공학전공
**200301582**
김성태

# 1. 링크드 리스트 **(Linked List)**

링크드 리스트의 개념도를 표현하면  구현하면 다음과 같다.




시작 노드를 가리키며 head가 있고 그 head를 따라 가면 시작 노드가 나온다.
시작 노드의 value는 1이고 다음 노드를 가리키는 next를 따라 가면 2번째 노드가 나온다.
2번째 노드의 value는 3이고 다음 노드를 가리키는 next를 따라가면 3번째 노드가 나온다.
3번째의 노드의 value는 4이고 다음 노드를 가리키는 next를 따라가면 4번째 노드가 나온다.
4번째 노드의 value는 6이고 다음 노드를 가리키는 next를 따라가면 5번 째 노드가 나온다.
value는 13이고 다음 노드를 가리키는 노드 next를 따라가면 6번째 노드가 나온다.
6번째 노드의 value는 19이고 다음 노드를 가리키는 next가 NULL이므로 더 이상 노드가 없다.

**① 노드 삽입**

비어 있는 링크드 리스트에 새로운 원소를 삽입하려면 새 노드를 생성하고 head에 연결 한다.

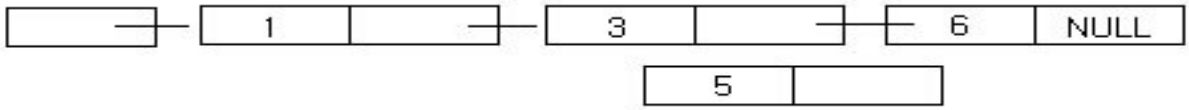
```
if (head == NULL)
{
	head = (struct Node *) malloc(sizeof(struct Node));
	head->value = value;
	head->next = NULL;
}
```

비어있지 않은 링크드 리스트에 새로운 원소를 삽입할 경우에는 새로운 원소가 삽입될 위치를 찾아야 한다.
새로운 원소가 삽입될 위치는 노드를 순회하면 값이 입력보다 큰노드를 찾는 것이다.

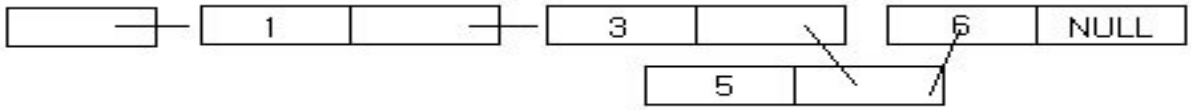
```
struct Node* node = head;
struct Node* prev = head;

while (node!= NULL)
{
	if (node->value > value)
		break;
	prev = node;
	node = node->next;
}
```

위치를 찾았으면 새 노드를 삽입한다. 먼저 새 노드 temp를 생성하고 그것의 value를 설정한다. 다음은 next들을 수정해야 한다.

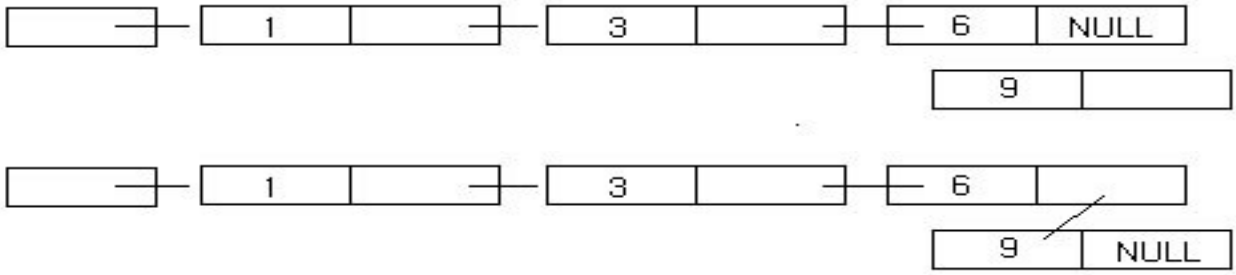
(1) 다음의 그림처럼 삽입할 위치가 노드의 중간일 때 앞 노드를 prev, 뒤 노드를 node라 하자.



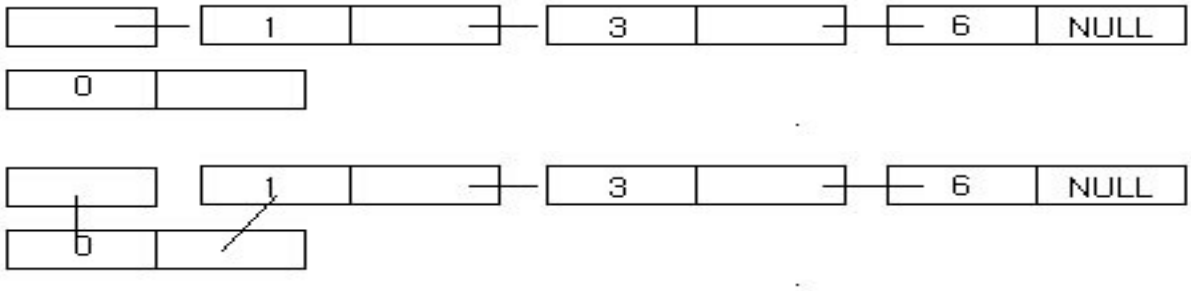

이때 next의 갱신은 다음의 그림처럼 되어야 한다.




(2) 다음 그림처럼 링크드 리스트의 마지막에 추가될 때는 prev의 next를 수정해야 한다.




(3) 다음의 그림처럼 링크드 리스트의 시작에 추가된다면 head의 값을 갱신해야 한다.




최종적으로 삽입하는 코드는 다음과 같다.

```
temp = (struct node *) malloc(sizeof(struct node));
temp->value = value;
temp->next = node;
if (prev == node)
        head = temp;
else
        prev->next = temp;
```

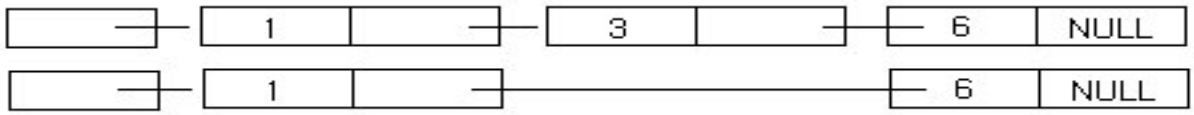
**② 노드 삭제**

링크드 리스트에 원소를 삭제하려면 원소의 위치를 찾아야 한다. 노드를 순회하면서 입력한 값을 갖는 노드

를 찾는다.

```
struct node* node = head;
struct node* prev = head;

while (node != NULL)
{
        if ( node->value == value)
                break;
        prev = node;
        node = node -> next;
}
```

(1) 다음의 그림처럼 삭제될 원소가 링크드 리스트의 중간일 때는 (원소의 값이 3) 앞 노드를 prev, 뒤 노드를 node라 하면 prev의 next를 node->next로 설정하여야 한다.




(2) 다음의 그림처럼 삭제될 원소가 링크드 리스트의 마지막이면 prev의 next를 NULL로 설정한다.


(3) 다음의 그림처럼 삭제될 원소가 링크드 리스트의 시작일 때는 head를 수정하여야 한다.최종적으로 삭제하는 코드는 다음과 같다.


```
if (prev == node)
        head = head->next;
else
        prev->next = node->next;
```

**③ 링크드 리스트의 확장**

(1) 이중 연결 리스트(double linked list)

노드에 앞의 노드를 가리키는 prev를 저장한다. head와 마찬가지로 tail이라는 마지막 노드를 가리키는 포인터 변수를 둔다.

(2) 원형 연결 리스트(ciccular linked list)

마지막 노드에 시작 노드를 연결한다.

**④ 링크드 리스트의 장단점**

(1) 장점
노드의 연결만으로 구성하므로 리스트의 연결(Combine)과 분리(Split)가 용이함.
포인터의 조작만 필요함으로 원소의 이동이 필요치 않아 삽입, 삭제가 용이함.
필요없는 노드는 메모리에서 삭제하므로 사용 후 기억장소의 재사용이 가능함.

(2) 단점
포인터(Link값)으로 인해 밀도가 낮다
알고리즘이 복잡함.

## 2. 크루스칼 알고리즘(Kruskal Algorithms)

**① 합집합을 수행하는 연산자**

merge(p, q)

-> Set_union 함수는 집합 set1과 set2의 합집합인 setu를 만든다. 우선 setu가 set_init으로 초기화된다. 다음에 set1의 원소들에 대해 list_ins_next를 반복 호출해서 set1의 원소들을 setu에 넣는다. 마지막으로 set2의 원소들이 비슷한 방식으로 setu에 넣어지는데, setu에 같은 원소가 중복되지 않도록 삽입하기 전에 set_is_member를 호출한다.

```
int set_union(Set *setu, const Set *set1, const Set *set2)
{

/* 링크드 리스트 포인터 생성 */
        ListElmt *member;
        void *data;

/* 합집합 초기화. */
        set_init(setu, set1->setmatch, NULL);

/* 첫번째 집합의 원소들 삽입. */
/* member 변수에 set1 의 head 를 입력하고 리스트의 끝까지 이동하면서 내용을 새로운 합집합의 주소로 넘겨 준다. 만약, 데이터 삽입시 오류가 발생할 경우, 새로운 합집합(setu)를 제거하고 오류값(-1)을 리턴한다. */
        for(member = list_head(set1); member != NULL; member = list_next(member)) {
                data = list_data(member);

                if(list_inst_next(setu, list_tail(setu), data) != 0) {
                        set_destroy(setu);
                        return -1;
                }
        }

/* 두번째 집합의 원소들 삽입. */
/* 첫번째 집합의 원소들 삽입과 마찬가지로 member의 변수에 set2의 head 를 입력하고 리스트의 끝까지 이동하면서 해당 원소들을 삽입한다. 하지만, 첫번째 삽입과는 달리 원소들을 삽입할 때 마다 같은 원소의 포함 유무를 검색하여 중복된 데이터의 삽입을 방지한다. */
        for(member = list_head(set2); member != NULL; member = list_next(member)) {
                if(set_is_member(set1, list_data(member)) {
                        /* 중복 삽입을 방지 */
                        continue;
```

```
                }
                else {
                        data = list_data(member);
                        if(list_ins_next(setu, list_tail(setu), data) != 0) {
                                set_destroy(setu);
                                return -1;
                        }
                }
        }

        return 0;
}
```

**② 원소가 어떤 집합에 속해 있는 지를 알 수 있는** find(i) **프로시져**.

find(i)
-> Set_is_member 함수는 특정 원소가 집합에 속하는지 확인한다. 일치하는 원소를 찾거나 모든 원소를 순회할 때까지 list_next를 사용해서 집합을 순회한다.

```
int set_is_member(const Set *set, const void *data)
{
        ListElmt *member;

/* 원소가 집합에 속하는지 확인 */
/* member 포인터에 set의 head를 입력후 리스트의 끝까지 이동하면서 리스트 속의 데이터를 검색을 원하는 데이터와 비교한다. 만약 일치하는 데이터가 있을 경우 -1을 리턴하고 없는 경우 0을 리턴한다. */
        for(member = list_head(set); member != NULL; member = list_next(member)) {

        if(set->match(data, list_data(member))
                return -1;
        }

        return 0;
}
```

**③ 두 집합이 같은 집합인지를 비교하는 연산자** : equal(p, q)

equal(p, q)
-> set_is_equal 함수는 집합 set1과 집합 set2가 같은지 확인하는 함수이다. 두 집합이 같으면 크기도 같으므로 먼저 크기를 비교한다. 두 집합의 크기가 다르면 두 집합은 다르다. 두 집합의 크기가 같으면 set1이 set2의 부분집합인지에 대한 결과를 리턴하면 된다. 이것은 set_is_subset을 호출해서 결정된다.

Set_is_subset(set1, set2)
-> Set_is_subset 함수는 집합 set1이 집합 set2의 부분집합인지 확인하는 함수이다. 어떤 집합의 부분집합은 크기가 작거나 같아야 하므로 먼저 크기를 비교한다. Set1의 크기가 set2의 크기보다 크면 set1은 set2의 부분집합이 아니다. 작다면 set1에 속하면서 set2에 속하지 않는 원소를 찾거나 모든 원소들을 순회할 때까지 list_next를 사용해서 set1의 원소들을 순회한다. Set1에 속하면서 set2에 속하지 않는 원소를 찾으면 set1은 set2의 부분집합이 아니다. Set1의 모든 원소를 순회하면 set1은 set2의 부분집합이다.

```
int set_is_equal(const Set *set1, const Set *set2)
{
/* 규칙에 어긋나는 경우들에 대한 빠른 테스트 */
/* 크기를 비교하여 크기가 같지 않는 경우에는 0(거짓)을 리턴한다.
        if(set_size(set1) != set_size(set2))
                return 0;

/* 크기가 같은 집합의 경우에 부분집합이면 서로 같다.
        return set_is_subset(set1, set2);
}
```

```
/* 부분집합의 여부를 확인하는 set_is_subset 함수.*/
/* 부분집합일 경우 1을 리턴하고 부분집합이 아닐경우 0을 리턴한다.*/
set_is_subset

int set_is_subset(const Set *set1, const Set *set2)
{
        ListElmt *member;

        /* 규칙에 어긋나는 경우들에 대한 빠른 테스트
        if(set_size(set1) > set_size(set2))
                return 0;

/* set1 이 set2의 부분 집합인지 확인. */
/* set1 리스트의 head 에서 tail 까지 모든 데이터들을 set2 의 멤버와 비교한다. 만약 하나라도 set2에
속하지 않은 데이터가 있다면, 부분 집합이 아니므로 0을 리턴한다. */
        for(member = list_head(set1); member != NULL; member = list_next(member)) {
                if(!set_is_member(set2, list_data(member)))
                        return 0;
        }

        return 1;
}
```