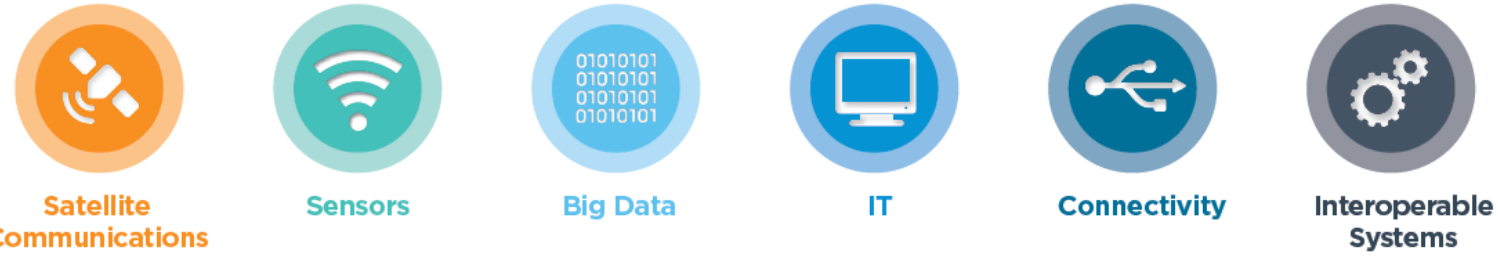
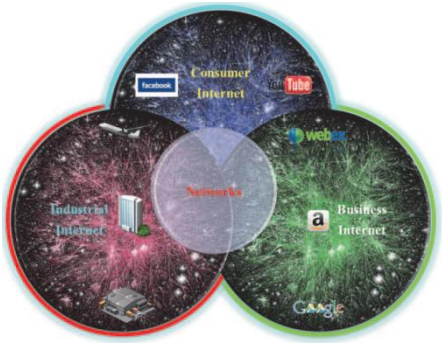
가천대학교
Gachon University

연결 리스트 #1

Dept. of Computer Engineering at Gachon Univ.

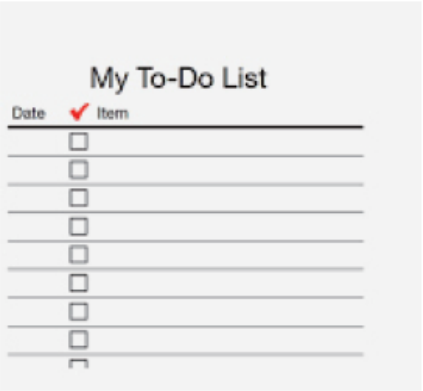
Prof. Chang Choi





리스트

- **일상생활에서의 리스트**
  - 오늘 해야 할 일: (청소, 쇼핑, 영화관람)
  - 버킷 리스트: (세계여행하기, 새로운 언어 배우기, 마라톤 뛰기)
  - 요일들: (일요일, 월요일, ... ,토요일)
  - 카드 한 벌의 값: (Ace, 2, 3,..., King)



$$L = (item_0, item_1, item_2, . . ., item_{n-1})$$

- **리스트에 새로운 항목을 추가한다(삽입 연산).**
- **리스트에서 항목을 삭제한다(삭제 연산).**
- **리스트에서 특정한 항목을 찾는다(탐색 연산).**

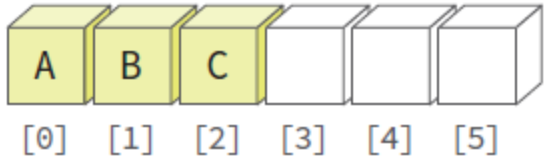
· 객체: n개의 element형으로 구성된 순서 있는 모임

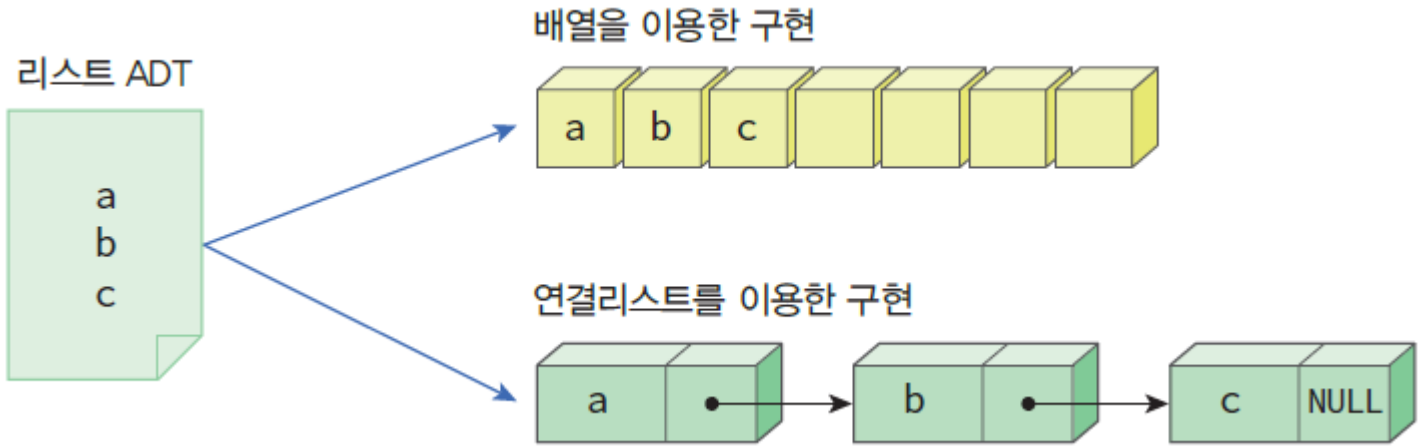
· 연산:

- insert(list, pos, item) ::= pos 위치에 요소를 추가한다.
- insert_last(list, item) ::= 맨 끝에 요소를 추가한다.
- insert_first(list, item) ::= 맨 처음에 요소를 추가한다.
- delete(list, pos) ::= pos 위치의 요소를 제거한다.
- clear(list) ::= 리스트의 모든 요소를 제거한다.
- get_entry(list, pos) ::= pos 위치의 요소를 반환한다.
- get_length(list) ::= 리스트의 길이를 구한다.
- is_empty(list) ::= 리스트가 비었는지를 검사한다.
- is_full(list) ::= 리스트가 꽉 찼는지를 검사한다.
- print_list(list) ::= 리스트의 모든 요소를 표시한다.

리스트 구현 방법

- 배열을 이용하여 리스트를 구현하면 순차적인 메모리 공간이 할당되므로, 이것을 리스트의 순차적 표현(sequential representation)이라고 함.





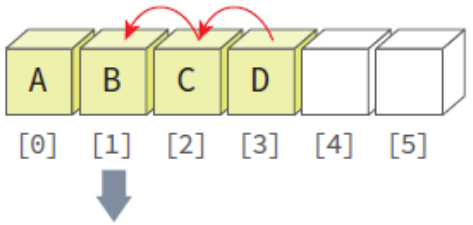
ArrayListType의 구현

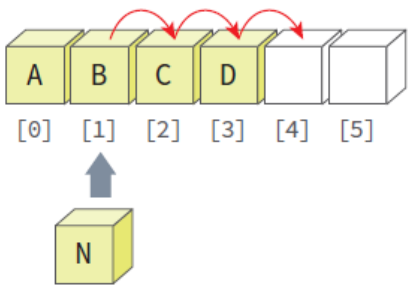
```c
#include <stdio.h>
#include <stdlib.h>
#define _CRT_SECURE_NO_WARNINGS
#define MAX_LIST_SIZE 100 // 리스트의 최대크기

typedef int element;// 항목의 정의
typedef struct {
   element array[MAX_LIST_SIZE];  // 배열 정의
   int size;  // 현재 리스트에 저장된 항목들의 개수
} ArrayListType;

// 오류 처리 함수
void error(const char* message)
{  fprintf(stderr, "%s\n", message);
   exit(1);                          }

// 리스트 초기화 함수
void init(ArrayListType* L)
{  L->size = 0;  }

// 리스트가 비어 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_empty(ArrayListType* L)
{   return L->size == 0;   }

// 리스트가 가득 차 있으면 1을 반환
// 그렇지 많으면 1을 반환
int is_full(ArrayListType *L)
{   return L->size == MAX_LIST_SIZE;   }
```

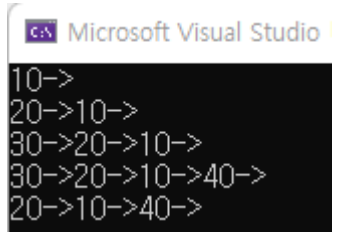
```c
element get_entry(ArrayListType* L, int pos)
{   if (pos < 0 || pos >= L->size)
    error("위치 오류");
    return L->array[pos];
}

// 리스트 출력
void print_list(ArrayListType* L)
{
   int i;
   for (i = 0; i < L->size; i++)
   printf("%d->", L->array[i]);
   printf("\n");
}

void insert_last(ArrayListType* L, element item)
{
   if (L->size >= MAX_LIST_SIZE) {
       error("리스트 오버플로우");
    }
   L->array[L->size++] = item;
}
```

```c
void insert(ArrayListType* L, int pos, element item)
{
   if (!is_full(L) && (pos >= 0) && (pos <= L->size)) {
   for (int i = (L->size - 1); i >= pos; i--)
     L->array[i + 1] = L->array[i];
     L->array[pos] = item;
     L->size++;
     }
}
```

```c
element delete1(ArrayListType* L, int pos)
{
   element item;

   if (pos < 0 || pos >= L->size) error("위치 오류");
   item = L->array[pos];
   for (int i = pos; i < (L->size - 1); i++)
      L->array[i] = L->array[i + 1];
   L->size--;
   return item;
}

int main(void)
{
// ArrayListType를 정적으로 생성하고 ArrayListType를
// 가리키는 포인터를 함수의 매개변수로 전달한다.
   ArrayListType list;

init(&list);
insert(&list, 0, 10);print_list(&list);// 0번째 위치에 10 추가
insert(&list, 0, 20);print_list(&list);// 0번째 위치에 20 추가
insert(&list, 0, 30);print_list(&list);// 0번째 위치에 30 추가
insert_last(&list, 40);print_list(&list);// 맨 끝에 40 추가
delete1(&list, 0);print_list(&list);// 0번째 항목 삭제
return 0;
}
```



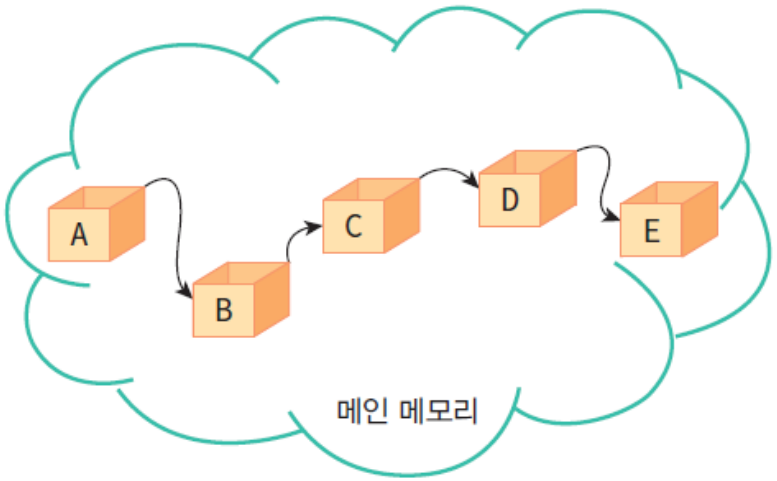


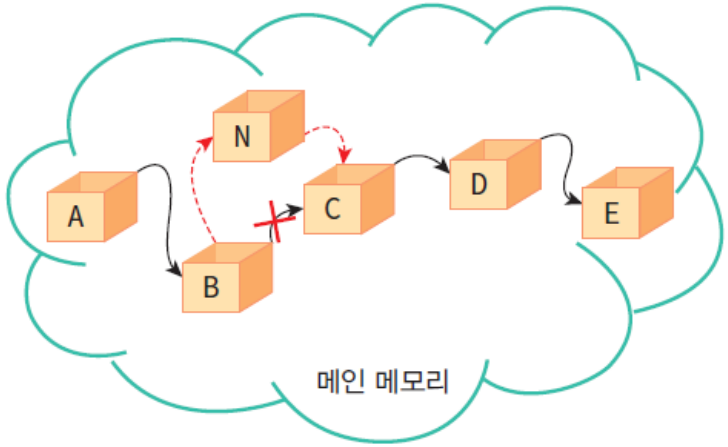


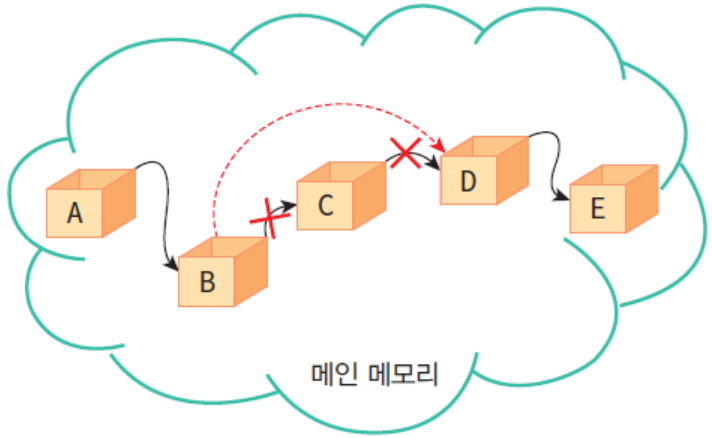
연결된 표현

- 리스트의 항목들을 노드(node)라고 하는 곳에 분산하여 저장
  - 노드는 데이터 필드와 링크 필드로 구성
  - 데이터 필드 – 리스트의 원소, 즉 데이터 값을 저장하는 곳
  - 링크 필드 – 다른 노드의 주소 값을 저장하는 장소 (포인터)

- 장점
  - 삽입, 삭제가 보다 용이하다.
  - 연속된 메모리 공간이 필요 없다.
  - 크기 제한이 없다
- 단점
  - 구현이 어렵다.
  - 오류가 발생하기 쉽다.

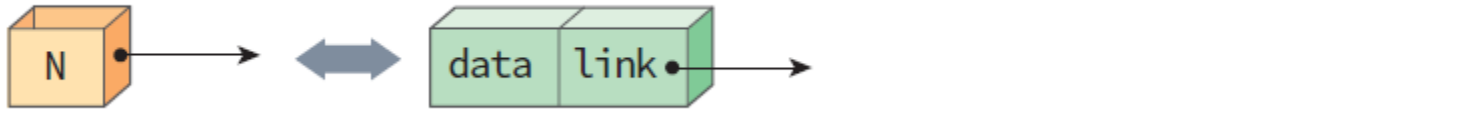




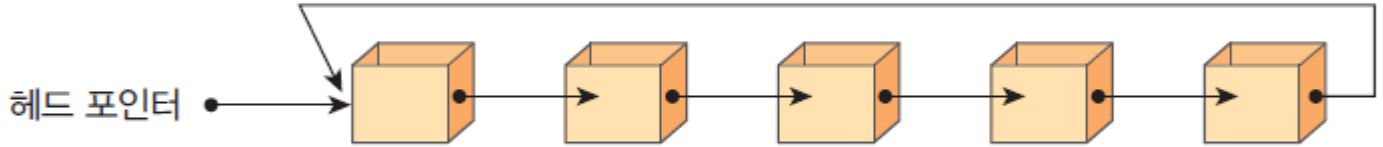


노드의 구조

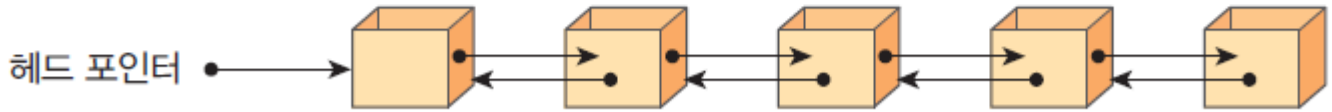
- 노드 = 데이터 필드 + 링크 필드



단순 연결 리스트

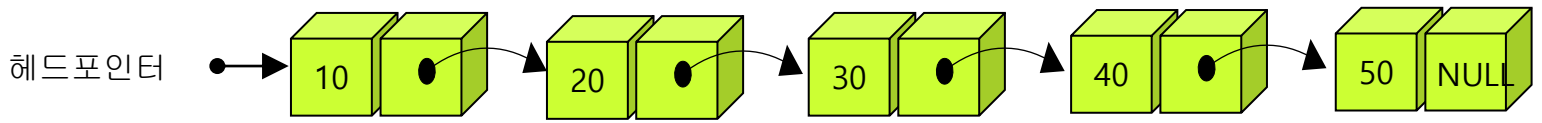


원형 연결 리스트



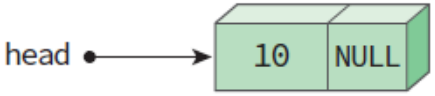
이중 연결 리스트

단순 연결 리스트



- 하나의 링크 필드를 이용하여 연결
- 마지막 노드의 링크 값은 NULL

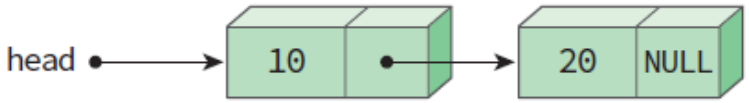

```
typedef int element;
// 노드 타입을 구조체로 정의한다.
typedef struct ListNode {
        element data;
        struct ListNode *link;
} ListNode;
```




```
ListNode *head = NULL;

head = (ListNode *)malloc(sizeof(ListNode));

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


```
ListNode *p;
p = (ListNode *)malloc(sizeof(ListNode));
p->data = 20;
p->link = NULL;
```

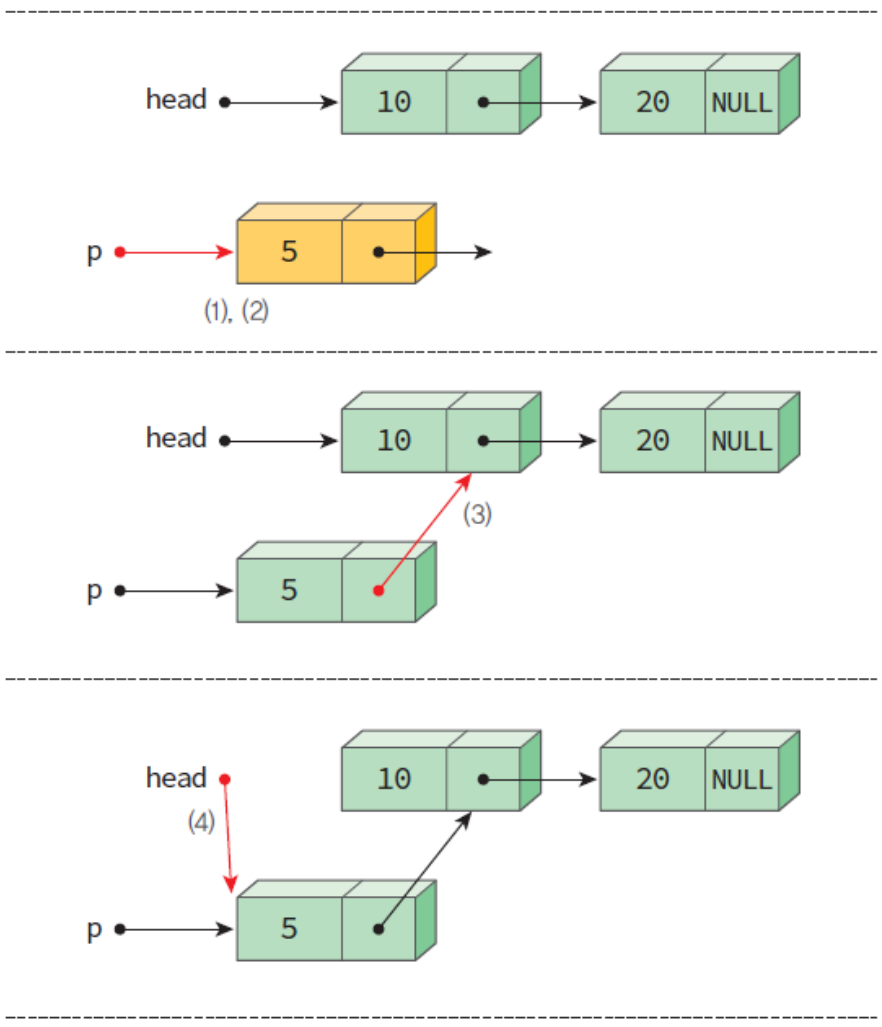
```
head->link = p;
```



단순 연결 리스트의 연산 #1

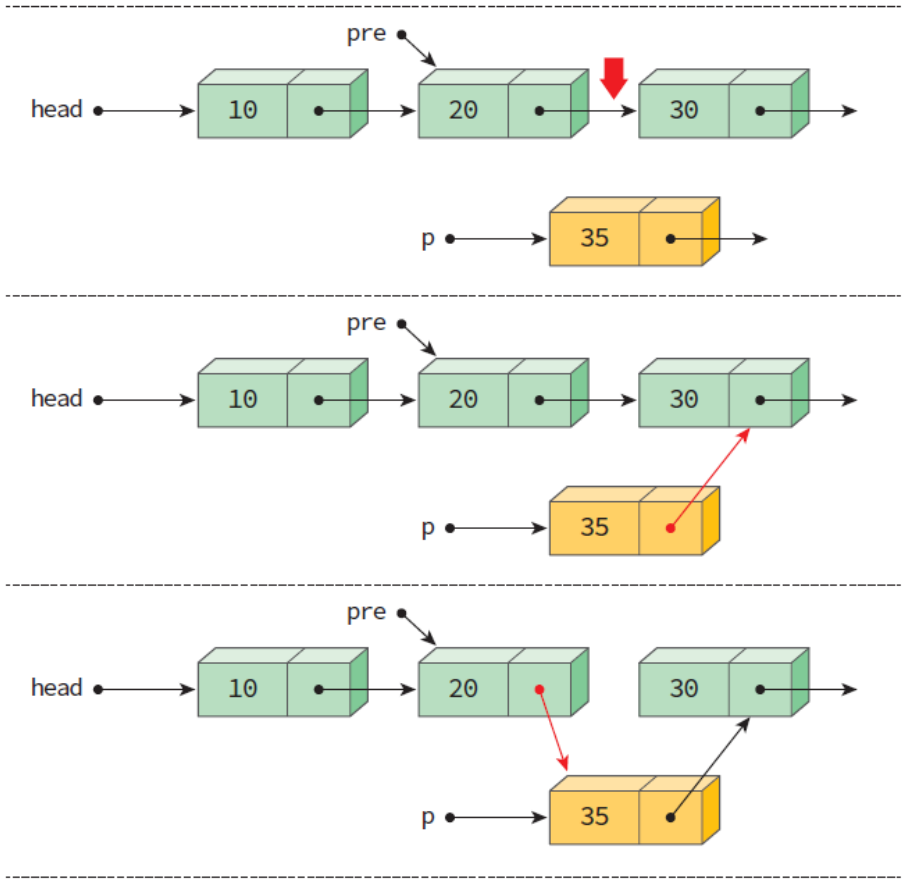
- insert_first(): 리스트의 시작 부분에 항목을 삽입하는 함수
- insert(): 리스트의 중간 부분에 항목을 삽입하는 함수
- delete_first(): 리스트의 첫 번째 항목을 삭제하는 함수
- delete(): 리스트의 중간 항목을 삭제하는 함수(도전 문제)
- print_list(): 리스트를 방문하여 모든 항목을 출력하는 함수

```
ListNode* insert_first(ListNode *head, int value)
{
        ListNode *p =
         (ListNode *)malloc(sizeof(ListNode));        //(1)
        p->data = value;                              //(2)
        p->link = head;                               //(3)
        head = p;    //(4)
        return head;
}
```



단순 연결 리스트의 연산 #2

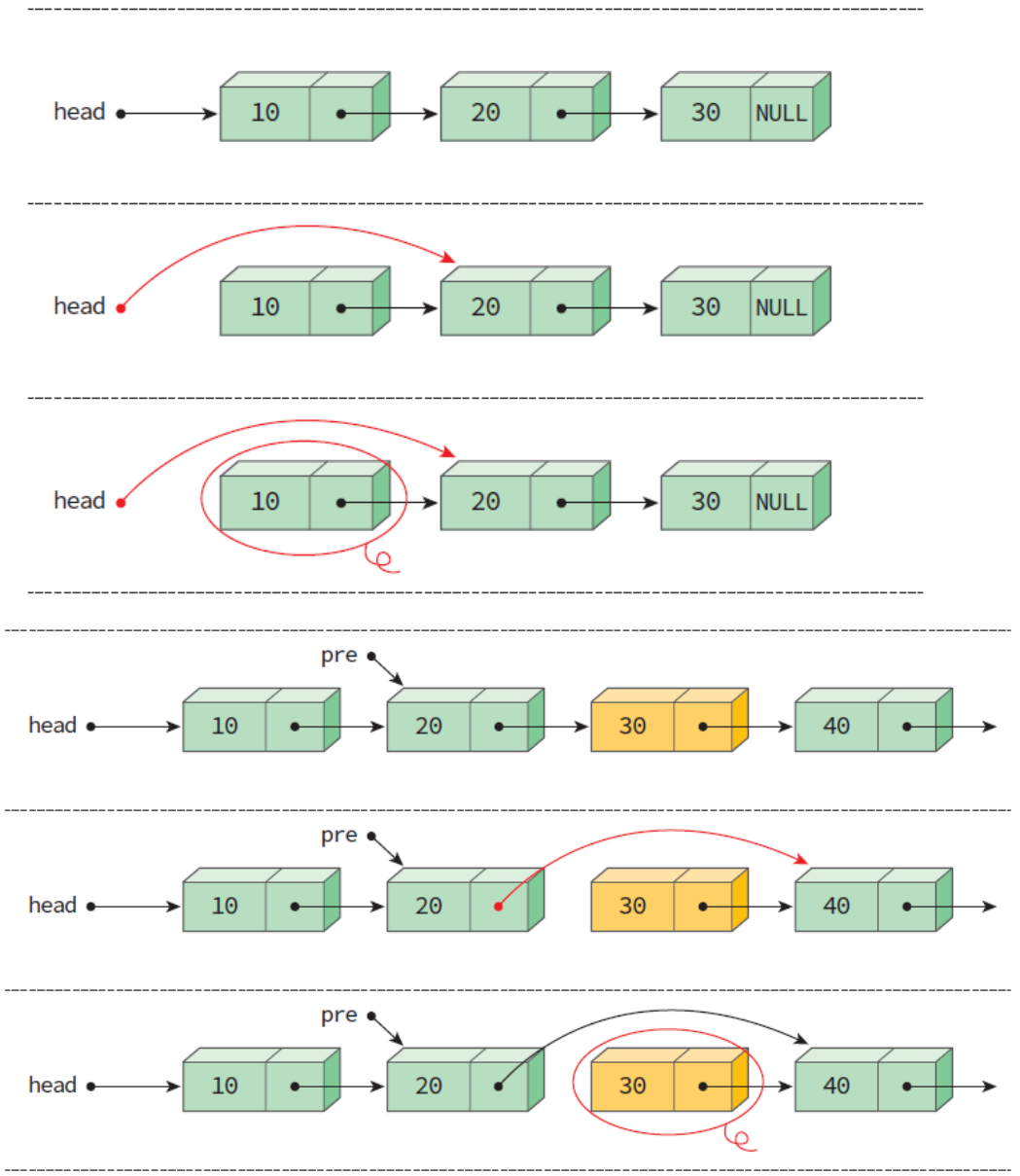
```
// 노드 pre 뒤에 새로운 노드 삽입
ListNode*  insert(ListNode *head, ListNode *pre, element value)
{
        ListNode *p = (ListNode *)malloc(sizeof(ListNode));      //(1)
        p->data = value;                                         //(2)
        p->link = pre->link;                                     //(3)

        pre->link = p;                                           //(4)

        return head;                                             //(5)

}
```



단순 연결 리스트의 연산 #3

```
ListNode* delete_first(ListNode *head)
{
        ListNode *removed;
        if (head == NULL) return NULL;
        removed = head;                 // (1)
        head = removed->link;           // (2)
        free(removed);                  // (3)
        return head;                    // (4)
}
```

```
// pre가 가리키는 노드의 다음 노드를 삭제한다.
ListNode* delete(ListNode *head, ListNode *pre)
{
        ListNode *removed;
        removed = pre->link;
        pre->link = removed->link;          // (2)
        free(removed);                      // (3)
        return head;                        // (4)
}
```



단순 연결 리스트의 연산 #4

```
void print_list(ListNode *head)
{
        for (ListNode *p = head; p != NULL; p = p->link)
                printf("%d->", p->data);
        printf("NULL \n");
}
```

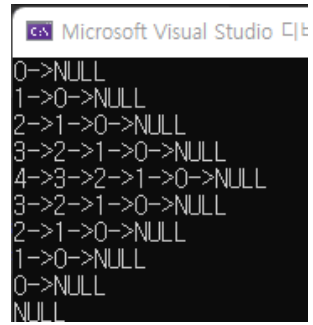
```
// 테스트 프로그램
int main(void)
{
        ListNode *head = NULL;

        for (int i = 0; i < 5; i++) {
                head = insert_first(head, i);
                print_list(head);
        }
        for (int i = 0; i < 5; i++) {
                head = delete_first(head);
                print_list(head);
        }
        return 0;
}
```

단순 연결 리스트의 연산 #5

```c
#include <stdio.h>
#include <stdlib.h>
#define _CRT_SECURE_NO_WARNINGS

typedef int element;
typedef struct ListNode { // 노드 타입
   element data;
   struct ListNode* link;
} ListNode;

// 오류 처리 함수
void error(const char* message)
{
   fprintf(stderr, "%s\n", message);
   exit(1);
}

ListNode* insert_first(ListNode* head, int value)
{
   ListNode* p = (ListNode*)malloc(sizeof(ListNode));//(1)
   p->data = value;// (2)
   p->link = head;// 헤드 포인터의 값을 복사//(3)
   head = p;// 헤드 포인터 변경//(4)
   return head;// 변경된 헤드 포인터 반환
}
```

```c
// 노드 pre 뒤에 새로운 노드 삽입
ListNode* insert(ListNode* head, ListNode* pre, element value)
{
   ListNode* p = (ListNode*)malloc(sizeof(ListNode));//(1)
   p->data = value;//(2)
   p->link = pre->link;//(3)
   pre->link = p;//(4)
   return head;//(5)
}

ListNode* delete_first(ListNode* head)
{
   ListNode* removed;
   if (head == NULL) return NULL;
   removed = head;// (1)
   head = removed->link;// (2)
   free(removed);// (3)
   return head;// (4)
}

// pre가 가리키는 노드의 다음 노드를 삭제한다.
ListNode* delete1(ListNode* head, ListNode* pre)
{
   ListNode* removed;
   removed = pre->link;
   pre->link = removed->link;// (2)
   free(removed);// (3)
   return head;// (4)
}
```

```c
void print_list(ListNode* head)
{
   for (ListNode* p = head; p != NULL; p = p->link)
   printf("%d->", p->data);
   printf("NULL \n");
}

// 테스트 프로그램
int main(void)
{
   ListNode* head = NULL;

   for (int i = 0; i < 5; i++) {
   head = insert_first(head, i);
   print_list(head);
   }

   for (int i = 0; i < 5; i++) {
   head = delete_first(head);
   print_list(head);
   }
   return 0;
}
```

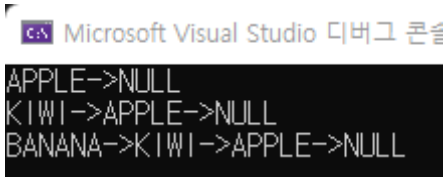


단어들을 저장하는 연결 리스트 구현

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define _CRT_SECURE_NO_WARNINGS
#pragma warning(disable:4996)

typedef struct {
   char name[100];
} element;

typedef struct ListNode { // 노드 타입
   element data;
   struct ListNode* link;
} ListNode;

// 오류 처리 함수
void error(const char* message)
{
   fprintf(stderr, "%s\n", message);
   exit(1);
}
```

```c
ListNode* insert_first(ListNode* head, element value)
{
   ListNode* p = (ListNode*)malloc(sizeof(ListNode));//(1)
   p->data = value;// (2)
   p->link = head;// 헤드 포인터의 값을 복사//(3)
   head = p;// 헤드 포인터 변경//(4)
   return head;
}

void print_list(ListNode* head)
{
   for (ListNode* p = head; p != NULL; p = p->link)
   printf("%s->", p->data.name);
   printf("NULL \n");
}
```

```c
// 테스트 프로그램
int main(void)
{
   ListNode* head = NULL;
   element data;
   strcpy(data.name, "APPLE");
   head = insert_first(head, data);
   print_list(head);

   strcpy(data.name, "KIWI");
   head = insert_first(head, data);
   print_list(head);

   strcpy(data.name, "BANANA");
   head = insert_first(head, data);
   print_list(head);

   return 0;
}
```



특정 값을 탐색하는 함수 구현

```
#include <stdio.h>
#include <stdlib.h>
#define _CRT_SECURE_NO_WARNINGS
#pragma warning(disable:4996)

typedef int element;

typedef struct ListNode { // 노드 타입
   element data;
   struct ListNode* link;
} ListNode;

ListNode* insert_first(ListNode* head, element value)
{
   ListNode* p = (ListNode*)malloc(sizeof(ListNode));//(1)
   p->data = value;// (2)
   p->link = head;// 헤드 포인터의 값을 복사//(3)
   head = p;// 헤드 포인터 변경//(4)
   return head;
}
```

```
void print_list(ListNode* head)
{
   for (ListNode* p = head; p != NULL; p = p->link)
   printf("%d->", p->data);
   printf("NULL \n");
}

ListNode* search_list(ListNode* head, element x)
{
   ListNode* p = head;

   while (p != NULL) {
   if (p->data == x) return p;
   p = p->link;
     }
   return NULL;// 탐색 실패
}
```

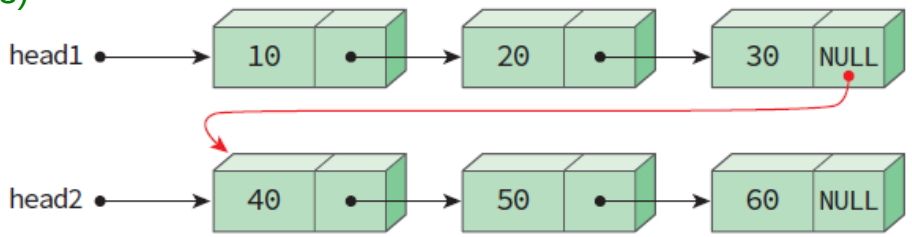
```
// 테스트 프로그램
int main(void)
{
   ListNode* head = NULL;

   head = insert_first(head, 10);
   print_list(head);
   head = insert_first(head, 20);
   print_list(head);
   head = insert_first(head, 30);
   print_list(head);

   if (search_list(head, 30) != NULL)
      printf("리스트에서 30을 찾았습니다. \n");
   else
      printf("리스트에서 30을 찾지 못했습니다. \n");

   return 0;
}
```

```
Microsoft Visual Studio 디버그 콘솔
10->NULL
20->10->NULL
30->20->10->NULL
리스트에서 30을 찾았습니다.
```

리스트를 하나로 결합하는 함수 구현

```
#include <stdio.h>
#include <stdlib.h>
#define _CRT_SECURE_NO_WARNINGS
#pragma warning(disable:4996)

typedef int element;

typedef struct ListNode { // 노드 타입
   element data;
   struct ListNode* link;
} ListNode;

ListNode* insert_first(ListNode* head, element value)
{
   ListNode* p = (ListNode*)malloc(sizeof(ListNode));//(1)
   p->data = value;// (2)
   p->link = head;// 헤드 포인터의 값을 복사//(3)
   head = p;// 헤드 포인터 변경//(4)
   return head;
}
```

```
void print_list(ListNode* head)
{
   for (ListNode* p = head; p != NULL; p = p->link)
   printf("%d->", p->data);
   printf("NULL \n");
}

ListNode* concat_list(ListNode* head1, ListNode* head2)
{
   if (head1 == NULL) return head2;
   else if (head2 == NULL) return head2;
        else {
              ListNode* p;
              p = head1;
              while (p->link != NULL)
                    p = p->link;
              p->link = head2;
              return head1;
             }
}
```

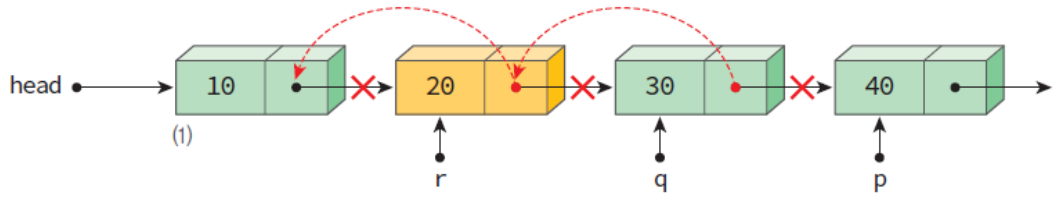


Microsoft Visual Studio 디버그 콘

```
30->20->10->NULL
50->40->NULL
30->20->10->50->40->NULL
```

```
// 테스트 프로그램
int main(void)
{
   ListNode* head1 = NULL;
   ListNode* head2 = NULL;

   head1 = insert_first(head1, 10);
   head1 = insert_first(head1, 20);
   head1 = insert_first(head1, 30);
   print_list(head1);

   head2 = insert_first(head2, 40);
   head2 = insert_first(head2, 50);
   print_list(head2);

   ListNode* total = concat_list(head1, head2);
   print_list(total);
   return 0;
}
```

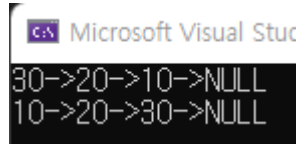
리스트를 역순으로 만드는 연산 구현




```
#include <stdio.h>
#include <stdlib.h>
#define _CRT_SECURE_NO_WARNINGS
#pragma warning(disable:4996)

typedef int element;

typedef struct ListNode { // 노드 타입
   element data;
   struct ListNode* link;
} ListNode;

ListNode* insert_first(ListNode* head, element value)
{
   ListNode* p = (ListNode*)malloc(sizeof(ListNode));//(1)
   p->data = value;// (2)
   p->link = head;// 헤드 포인터의 값을 복사//(3)
   head = p;// 헤드 포인터 변경//(4)
   return head;
}
```

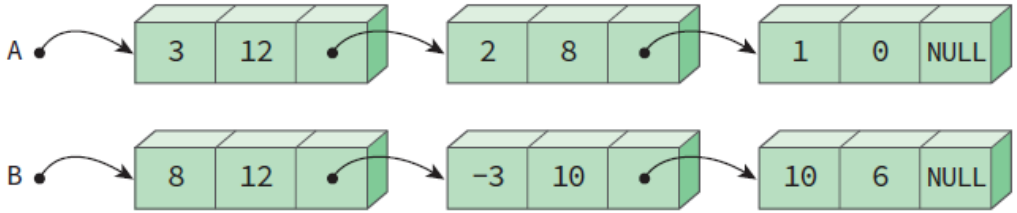
```
void print_list(ListNode* head)
{
   for (ListNode* p = head; p != NULL; p = p->link)
   printf("%d->", p->data);
   printf("NULL \n");
}

ListNode* reverse(ListNode* head)
{
   // 순회 포인터로 p, q, r을 사용
   ListNode* p, * q, * r;

   p = head;        // p는 역순으로 만들 리스트
   q = NULL;        // q는 역순으로 만들 노드
   while (p != NULL) {
      r = q;        // r은 역순으로 된 리스트.
                    // r은 q, q는 p를 차례로 따라간다.
      q = p;
      p = p->link;
      q->link = r;      // q의 링크 방향을 바꾼다.
      }
   return q;
}
```

```
// 테스트 프로그램
int main(void)
{
   ListNode* head1 = NULL;
   ListNode* head2 = NULL;

   head1 = insert_first(head1, 10);
   head1 = insert_first(head1, 20);
   head1 = insert_first(head1, 30);
   print_list(head1);

   head2 = reverse(head1);
   print_list(head2);
   return 0;
}
```

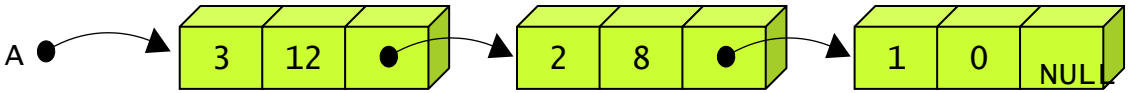


연결리스트의 응용: 다항식

예를 들면 다항식 $A(x)=3x^{12}+2x^{8}+1$과 $B(x)=8x^{12}-3x^{10}+10x^{6}$은 다음과 같이 표현된다.



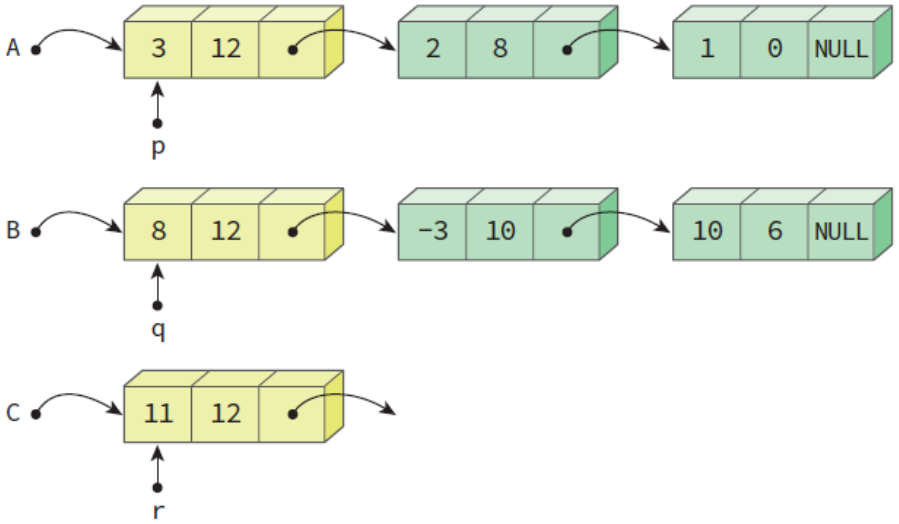
- 다항식을 컴퓨터로 처리하기 위한 자료구조
  - 다항식의 덧셈, 뺄셈...
- 하나의 다항식을 하나의 연결리스트로 표현
  - A=3x12+2x8+1



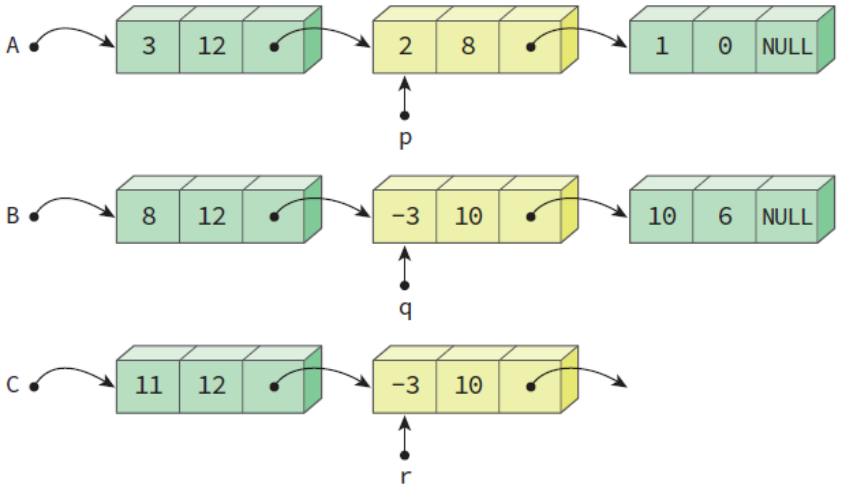

```
typedef struct ListNode {                // 노드 타입
        int coef;
        int expon;
        struct ListNode *link;
} ListNode;
```

다항식의 덧셈 구현

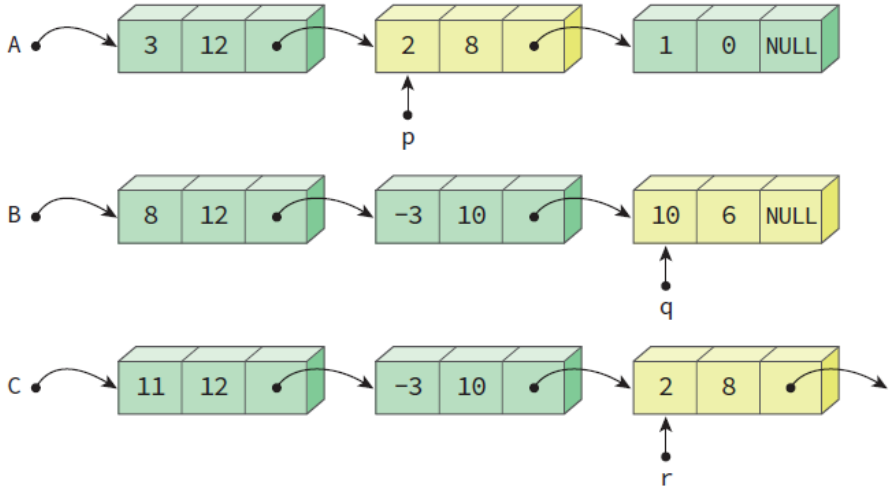
- 2개의 다항식을 더하는 덧셈 연산을 구현
  - $A=3x^{12}+2x^{8}+1$, $B=8x^{12}-3x^{10}+10x^{6}$이면
  - $A+B=11x^{12}-3x^{10}+2x^{8}+10x^{6}+1$



(a) p와 q가 가리키는 항들의 지수가 같으면 계수를 더한다.

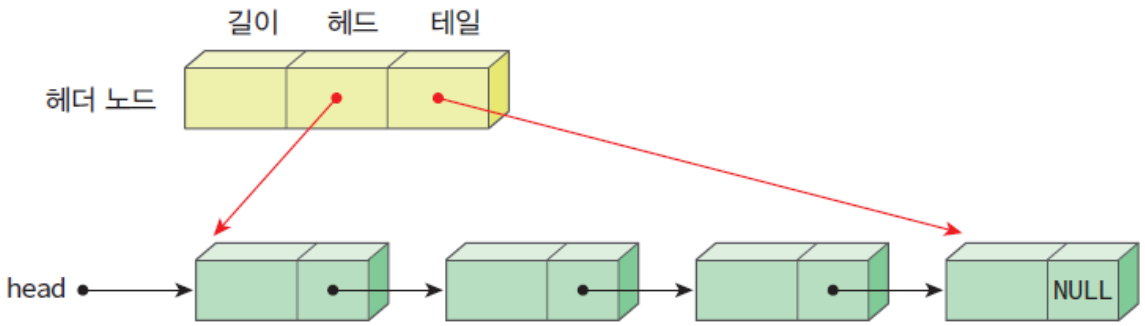


(b) q가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.



(c) p가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.

다항식 프로그램



- [Ch 6] polynomial.c파일 참조

```
Microsoft Visual Studio 디버그 콘솔
polynomial = 3^12 + 2^8 + 1^0 +
polynomial = 8^12 + -3^10 + 10^6 +
polynomial = 11^12 + -3^10 + 2^8 + 10^6 + 1^0 +
```

Thank you

Q & A