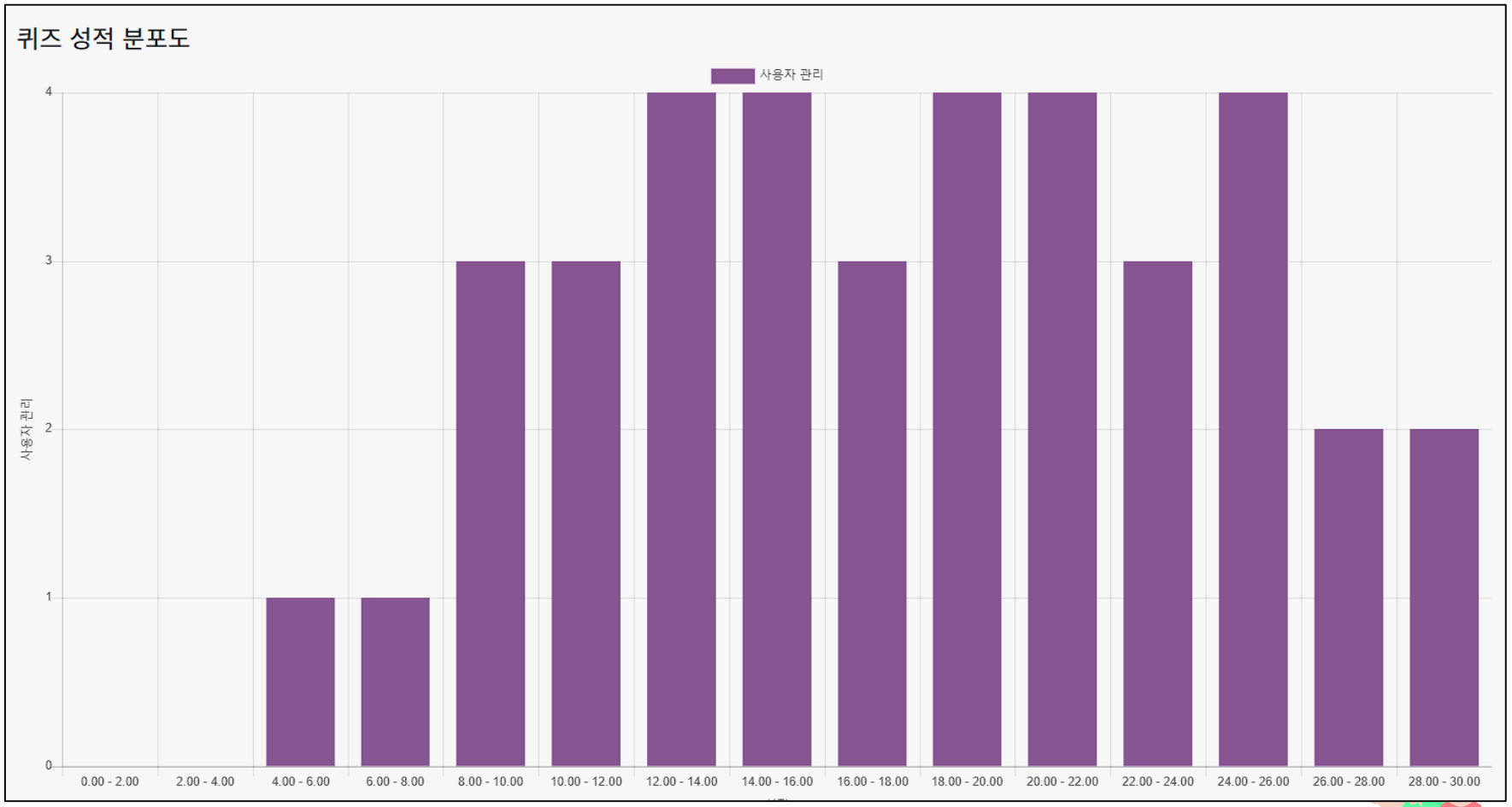
6장 연결 리스트 I

참고: 퀴즈 1 성적

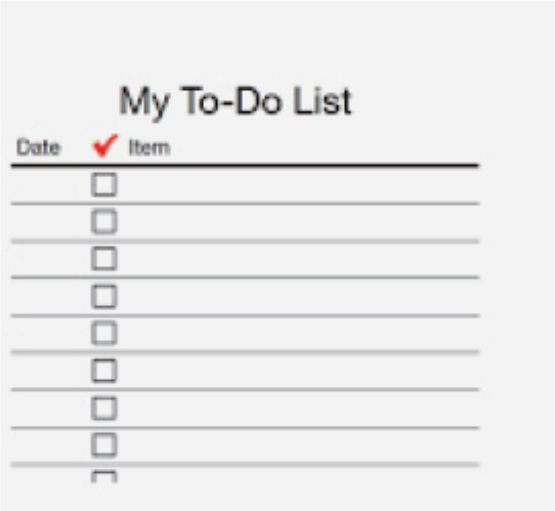


C로 쉽게 풀어쓴 자료구조

6.1 리스트 추상 데이터 타입

- 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)



리스트 정의 및 기본 연산

- 리스트: 순서(위치)에 따라 연속적으로 동일한 형태의 데이터들이 저장된 공간
  - 앞의 스택, 큐도 리스트의 일종

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

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

리스트 ADT

· 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.

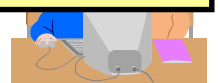
get_entry(list, pos) ::= pos 위치의 요소를 반환한다.

get_length(list) ::= 리스트의 길이를 구한다.

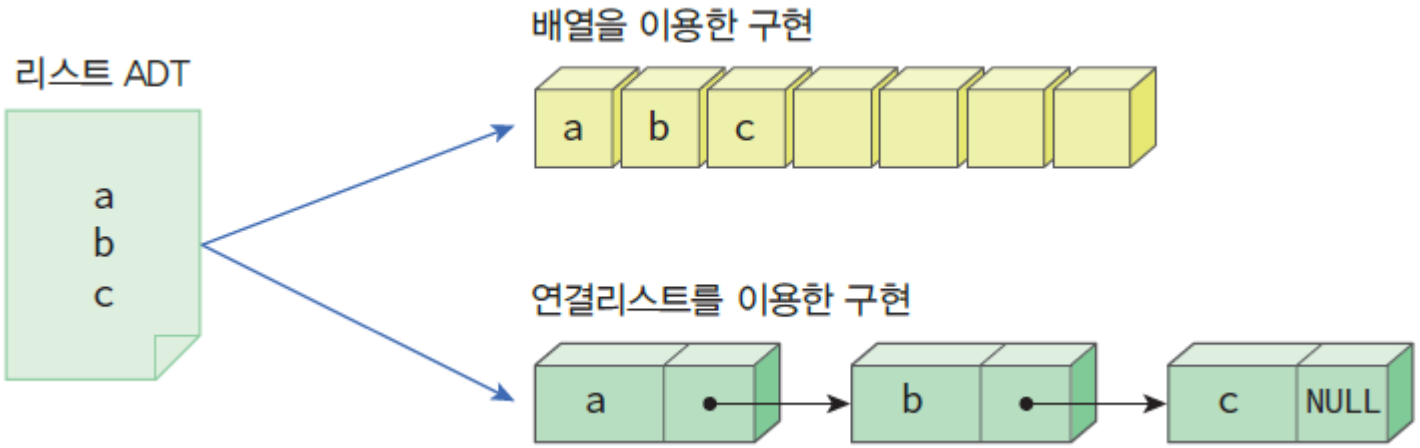
is_empty(list) ::= 리스트가 비었는지를 검사한다.

is_full(list) ::= 리스트가 꽉 찼는지를 검사한다.

print_list(list) ::= 리스트의 모든 요소를 표시한다.

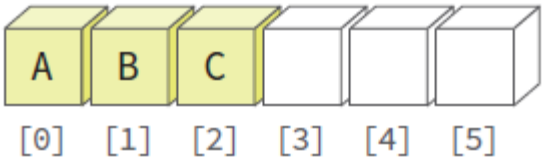


리스트 구현 방법



6.2 배열로 구현된 리스트

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



ArrayListType 구현: arraylist.c (1/4)


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

#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 1를 반환
// 그렇지 않으면 0를 반환
int is_empty(ArrayListType *L)
{
    return L->size == 0;
}
```

ArrayListType 구현: arraylist.c (2/4)


```
32  // 리스트가 가득 차 있으면 1을 반환
33  // 그렇지 않으면 1을 반환.
34  int is_full(ArrayListType *L)
35  {
36      return L->size == MAX_LIST_SIZE;
37  }
38
39  element get_entry(ArrayListType *L, int pos)
40  {
41      if (pos < 0 || pos >= L->size)
42          error("위치 오류");
43      return L->array[pos];
44  }
45
46  // 리스트 출력
47  void print_list(ArrayListType *L)
48  {
49      int i;
50      for (i=0; i<L->size; i++)
51          printf("%d->", L->array[i]);
52      printf("\n");
53  }
54
55  void insert_last(ArrayListType *L, element item)
56  {
57      if( L->size >= MAX_LIST_SIZE ) {
58          error("리스트 오버플로우");
59      }
60      L->array[L->size++] = item;
61  }
```

ArrayListType 구현: arraylist.c (3/4)


```
63  void insert(ArrayListType *L, int pos, element item)
64  {
65      int i;
66      if (!is_full(L) && (pos >= 0) && (pos <= L->size)) {
67          for (i = (L->size - 1); i >= pos; i--)
68              L->array[i + 1] = L->array[i];
69          L->array[pos] = item;
70          L->size++;
71      }
72  }
73
74  element delete(ArrayListType *L, int pos)
75  {
76      element item;
77      int i;
78
79      if (pos < 0 || pos >= L->size)
80          error("위 치  오 류 ");
81      item = L->array[pos];
82      for (i=pos; i<(L->size - 1); i++)
83          L->array[i] = L->array[i + 1];
84      L->size--;
85      return item;
86  }
```

ArrayListType 구현: arraylist.c (4/4)


```
89  int main(void)
90  {
91      // ArrayListType를 정적으로 생성하고 ArrayListType를
92      // 가리키는 포인터를 함수의 매개변수로 전달한다.
93      ArrayListType list;
94
95      init(&list);
96      insert(&list, 0, 10);   print_list(&list);  // 0번째 위치에 10 추가
97      insert(&list, 0, 20);   print_list(&list);  // 0번째 위치에 20 추가
98      insert(&list, 0, 30);   print_list(&list);  // 0번째 위치에 30 추가
99      insert_last(&list, 40); print_list(&list);  // 맨 끝에 40 추가
100     delete(&list, 0);       print_list(&list);  // 0번째 항목 삭제
101     return 0;
102 }
```

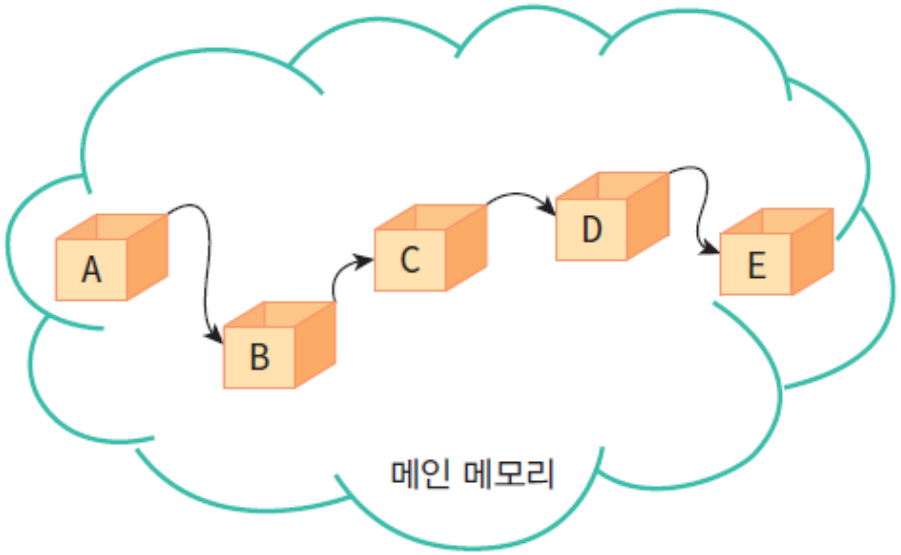
```
10->
20->10->
30->20->10->
30->20->10->40->
20->10->40->


--------------------------------
Process exited after 0.0205 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

6.3 연결 리스트

- 배열 형식의 리스트 구현 단점
  - 정적 변수로 선언 -> 크기를 정하기 힘들다.
    - 매우 큰 크기로 선언 -> 메모리 낭비가 심함
  - 삽입/삭제 비용이 크다.
    - 항상 연속적인 메모리 공간으로 구현되기 때문에 리스트 끝에서 삽입/삭제가 일어나지 않을 경우: 리스트 데이터의 많은 부분을 밀거나 당기는 연산이 필요하게 됨

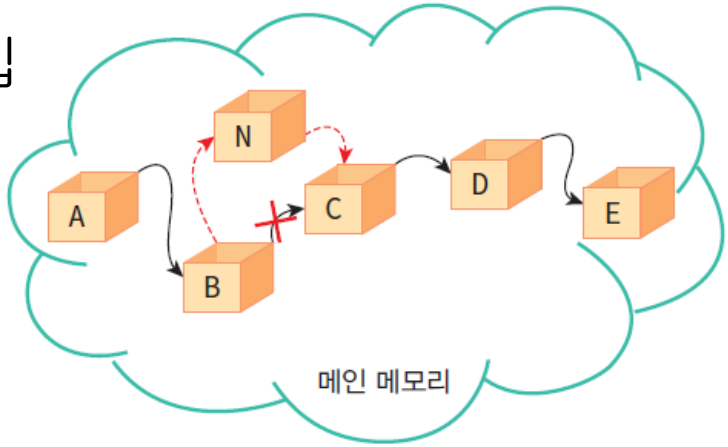
연결된 표현

- 장점: 동적으로 크기가 변할 수 있고, 삭제/삽입 시 데이터들을 이동할 필요가 없음
- 리스트의 항목들을 노드(node)라고 하는 곳에 분산하여 저장
- 노드는 데이타 필드와 링크 필드로 구성
  - 데이타 필드: 리스트의 원소, 즉 데이타 값을 저장하는 곳
  - 링크 필드: 다른 노드의 주소값을 저장하는 장소 (포인터)

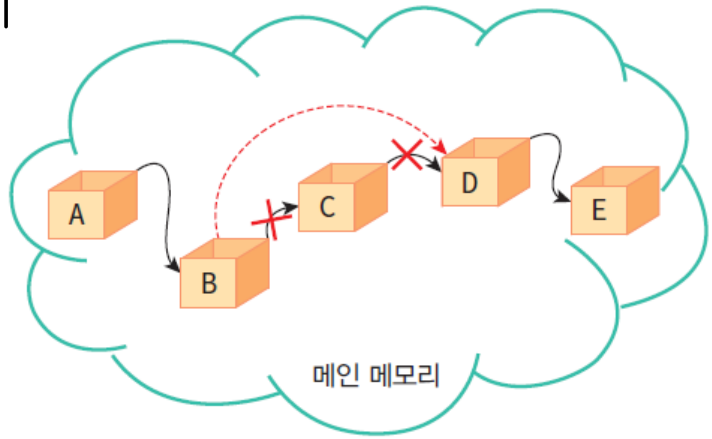


삽입과 삭제

- 삽입



- 삭제



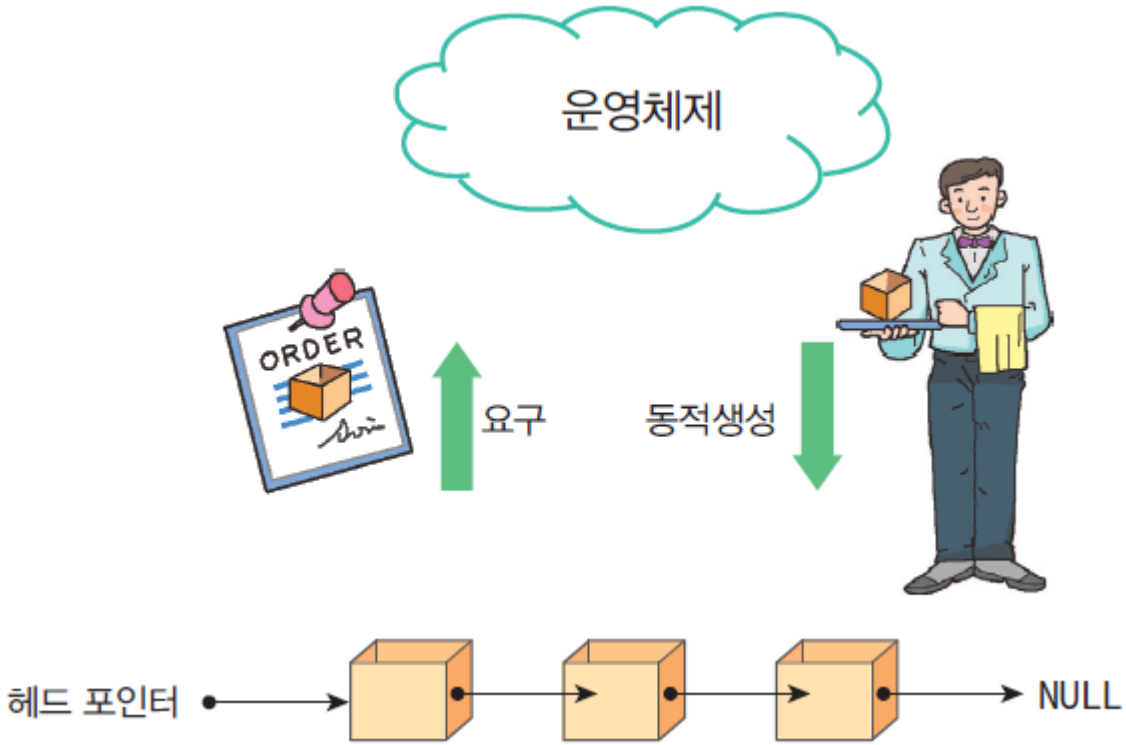
연결된 표현(연결 리스트)의 장단점

- 장점
  - 삽입, 삭제가 보다 용이하다. -> 연산 오버헤드가 적다
  - 연속된 메모리 공간이 필요 없다.
  - 크기 제한이 없다
- 단점
  - 구현이 어렵다.
  - 오류가 발생하기 쉽다.
  - 인덱스에 의한 access에 비해 access 시 오버헤드가 발생
    - 예) 리스트에 10번째 데이터 접근

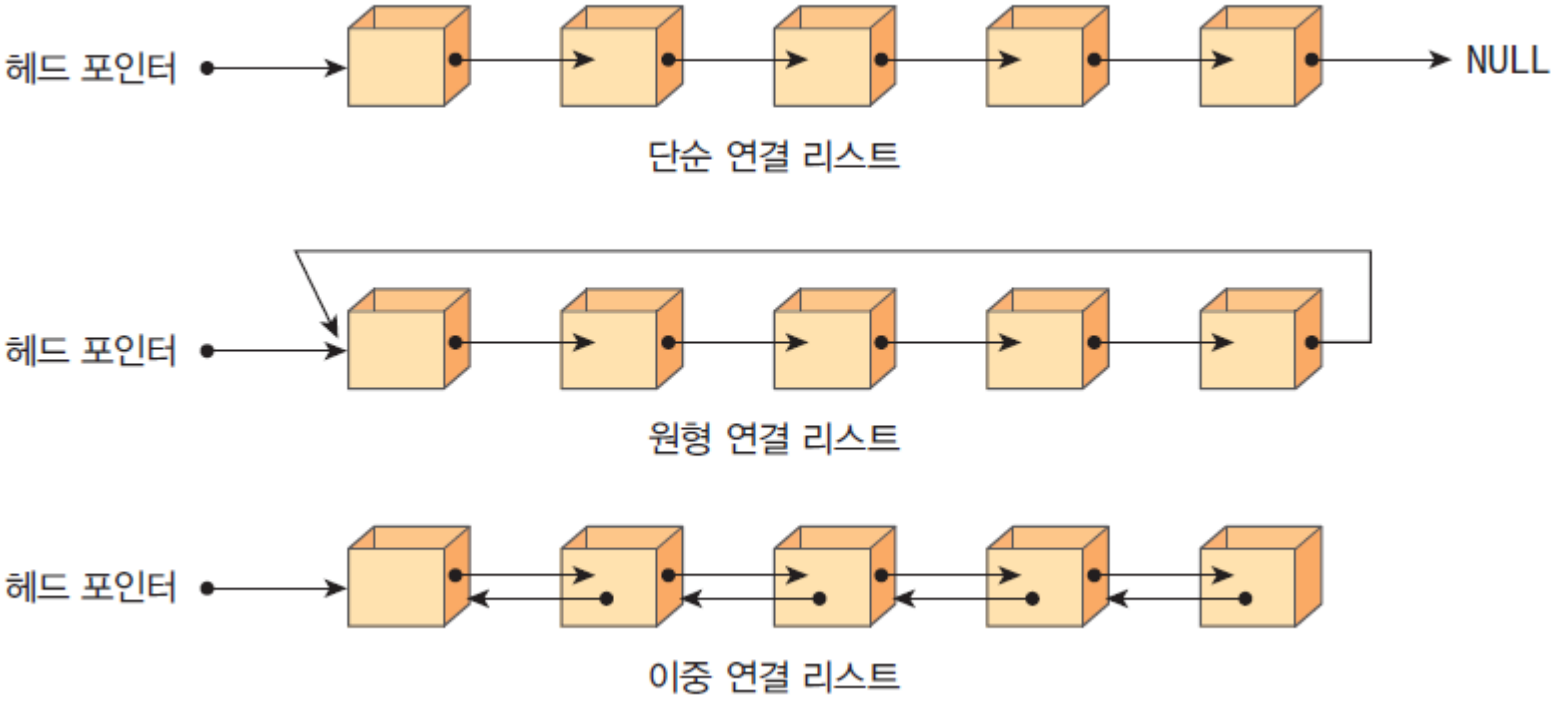
연결리스트의 구조

- 노드 = 데이터 필드 + 링크 필드

헤드 포인터와 노드의 생성

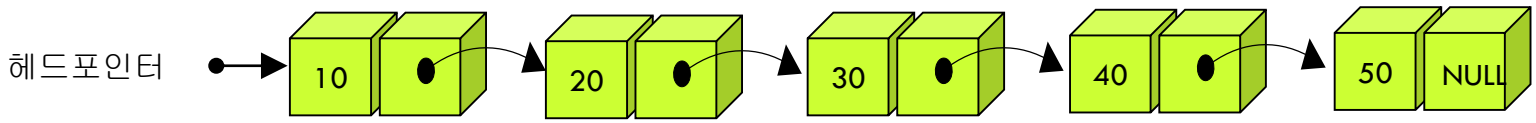


연결 리스트의 종류



6.4 단순 연결 리스트

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

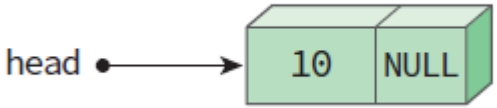


노드의 정의


```
typedef int element;

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

리스트의 생성


```
ListNode *head = NULL;

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

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

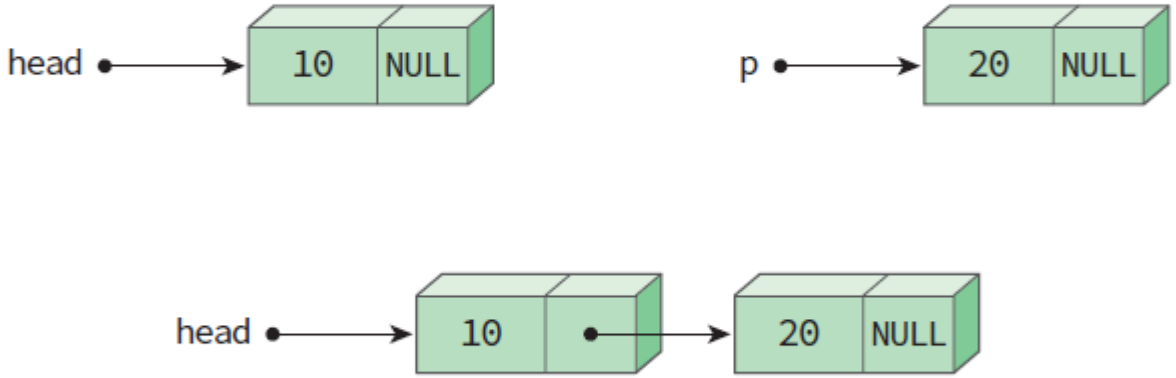


2번째 노드 생성


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

노드의 연결


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

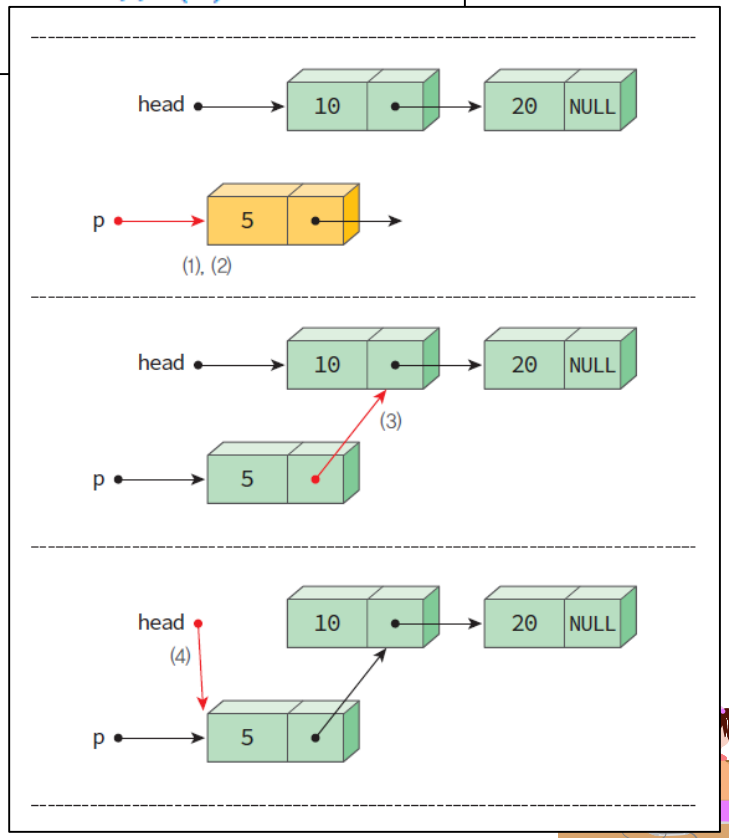


6.5 단순 연결 리스트의 연산

- 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

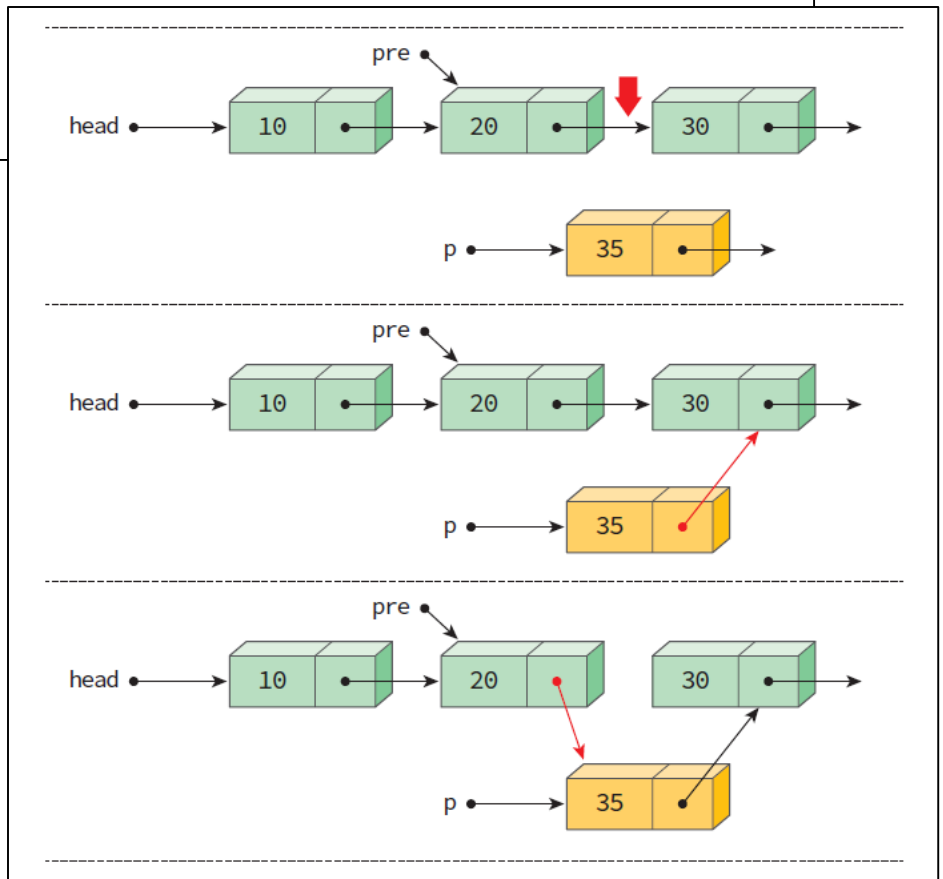
단순 연결 리스트(첫번째 노드 삽입): linklist1.c


```
17  // 리스트 제일 앞에 새로운 노드 삽입
18  ListNode* insert_first(ListNode *head, element value)
19  {
20      ListNode *p = (ListNode *)malloc(sizeof(ListNode));      // (1)
21      p->data = value;                                         // (2)
22      p->link = head; // 헤드 포인터의 값을 복사               // (3)
23      head = p;    // 헤드 포인터 변경                         // (4)
24      return head;     // 변경된 헤드 포인터 반환
25  }
```



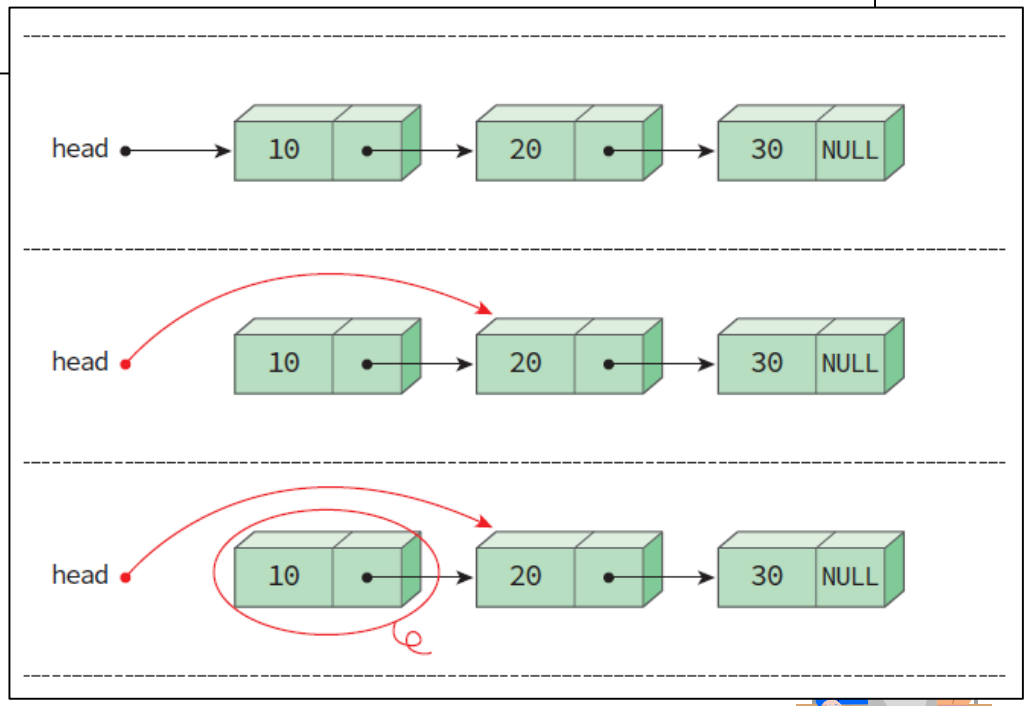
단순 연결 리스트(pre 뒤에 삽입): linklist1.c


```
27  // 노드 pre 뒤에 새로운 노드 삽입
28  ListNode*  insert(ListNode *head, ListNode *pre, element value)
29  {
30      ListNode *p = (ListNode *)malloc(sizeof(ListNode)); //(1)
31      p->data = value;          //(2)
32      p->link = pre->link;      //(3)
33      pre->link = p;            //(4)
34      return head;              //(5)
35  }
```



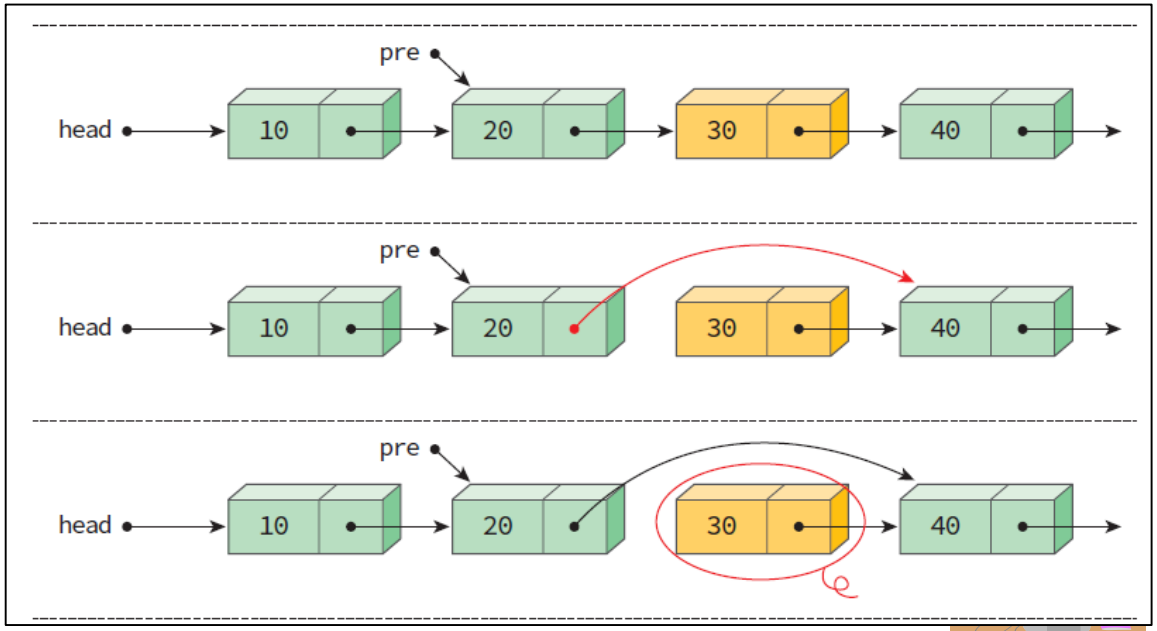
단순 연결 리스트(첫 노드 삭제): linklist1.c


```
37  // 첫 번째 노드 삭제
38  ListNode* delete_first(ListNode *head)
39  {
40      ListNode *removed;
41      if (head == NULL) return NULL;
42      removed = head;          // (1)
43      head = removed->link;    // (2)
44      free(removed);           // (3)
45      return head;             // (4)
46  }
```



단순 연결 리스트(pre 다음 노드 삭제): linklist1.c


```
48  // pre가 가리키는 노드의 다음 노드를 삭제한다.
49  ListNode* delete(ListNode *head, ListNode *pre)
50  {
51      ListNode *removed;
52      removed = pre->link;          // (1)
53      pre->link = removed->link;    // (2)
54      free(removed);                // (3)
55      return head;                  // (4)
56  }
```



방문해서 (출력) 연산: linklist1.c


```
58  void print_list(ListNode *head)
59  {
60      ListNode *p;
61      for (p = head; p != NULL; p = p->link)
62          printf("%d->", p->data);
63      printf("NULL \n");
64  }
```

main 함수: linkedlist1.c


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

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

```
0->NULL
1->0->NULL
2->1->0->NULL
3->2->1->0->NULL
4->3->2->1->0->NULL
3->2->1->0->NULL
2->1->0->NULL
1->0->NULL
0->NULL
NULL
```

Lab: 단어들을 저장 연결리스트 구현 (Lab1.c)


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

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

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

Lab: 단어들을 저장 연결리스트 구현 (Lab1.c)


```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;
}
```

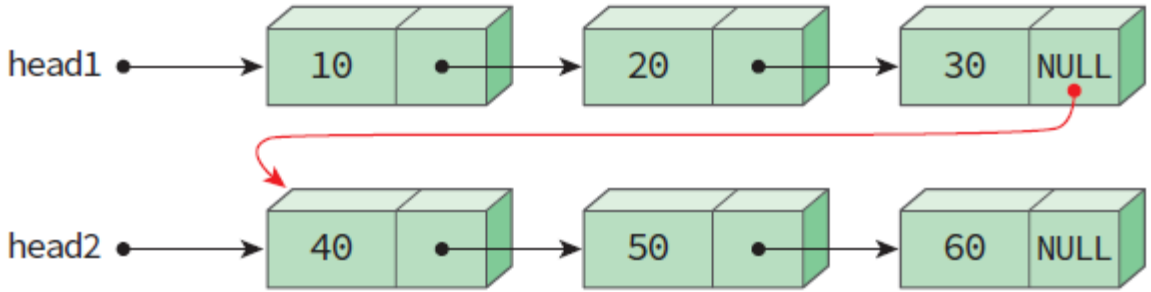
```
APPLE->NULL
KIWI->APPLE->NULL
BANANA->KIWI->APPLE->NULL
```

특정한 값을 탐색하는 함수 (Lab2.c)


```c
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;
}
```

```
10->NULL
20->10->NULL
30->20->10->NULL
리스트에서 30을 찾았습니다.
```

Lab3: 2개의 리스트를 연결 (lab3.c)



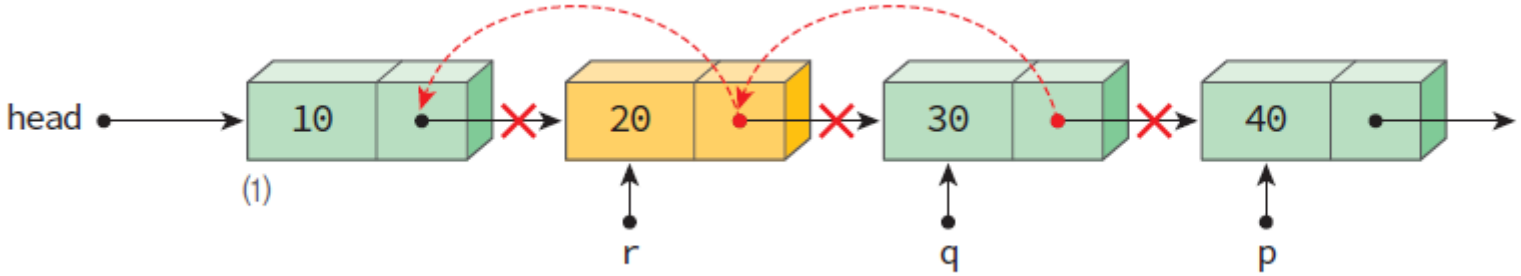

```
30  ListNode* concat_list(ListNode *head1, ListNode *head2)
31  {
32      if (head1 == NULL) return head2;
33      else if (head2 == NULL) return head2;
34      else {
35          ListNode *p;
36          p = head1;
37          while (p->link != NULL)
38              p = p->link;
39          p->link = head2;
40          return head1;
41      }
42  }
```

Lab3: 2개의 리스트를 연결 (lab3.c)


```c
// 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;
}
```

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

Lab: 리스트를 역순으로 만드는 연산 (lab4.c)




```
27  ListNode* reverse(ListNode *head)
28  {
29      // 순회 포인터로 p, q, r를 사용
30      ListNode *p, *q, *r;
31
32      p = head;           // p는 역순으로 만들 리스트
33      q = NULL;           // q는 역순으로 만들 노드
34      while (p != NULL)
35      {
36          r = q;          // r은 역순으로 된 리스트.
37                          // r은 q, q는 p를 차례로 따라간다.
38          q = p;
39          p = p->link;
40          q->link = r;    // q의 링크 방향을 바꾼다.
41      }
42      return q;
43  }
```

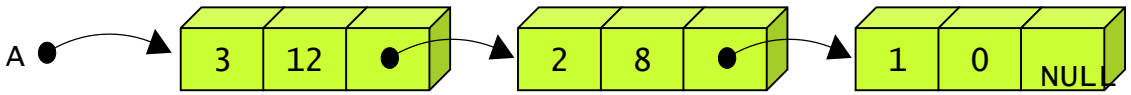
Lab: 리스트를 역순으로 만드는 연산 (lab4.c)


```
45  // 테스트 프로그램
46  int main(void)
47  {
48      ListNode* head1 = NULL;
49      ListNode* head2 = NULL;
50
51      head1 = insert_first(head1, 10);
52      head1 = insert_first(head1, 20);
53      head1 = insert_first(head1, 30);
54      print_list(head1);
55
56      head2 = reverse(head1);
57      print_list(head2);
58      return 0;
59  }
```

```
30->20->10->NULL
10->20->30->NULL
```

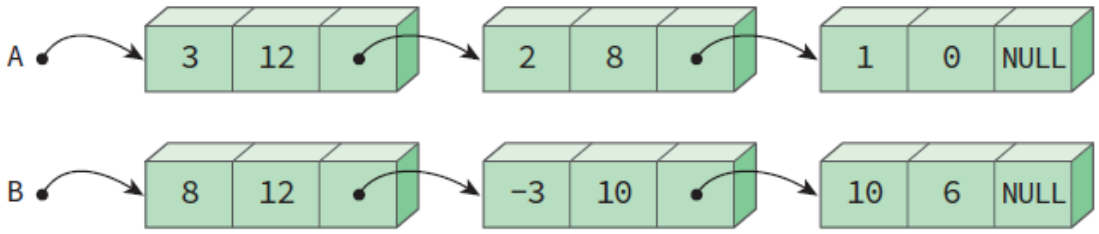
6.6 연결리스트의 응용: 다항식

- 다항식을 컴퓨터로 처리하기 위한 자료구조
  - 다항식의 덧셈, 뺄셈...
- 하나의 다항식을 하나의 연결리스트로 표현
  - $A = 3x^{12} + 2x^{8} + 1$



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.



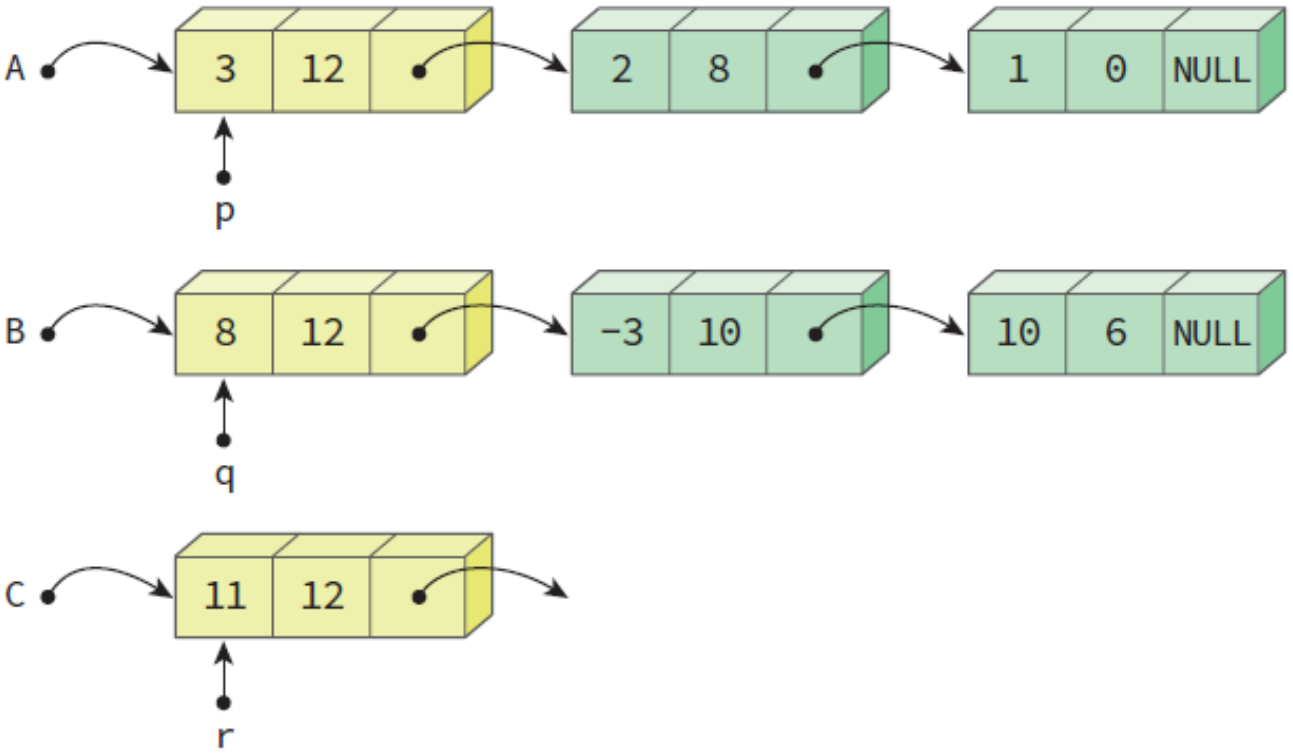

```c
#include <stdio.h>
#include <stdlib.h>

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

다항식의 덧셈

- 2개의 다항식을 더하는 덧셈 연산을 구현

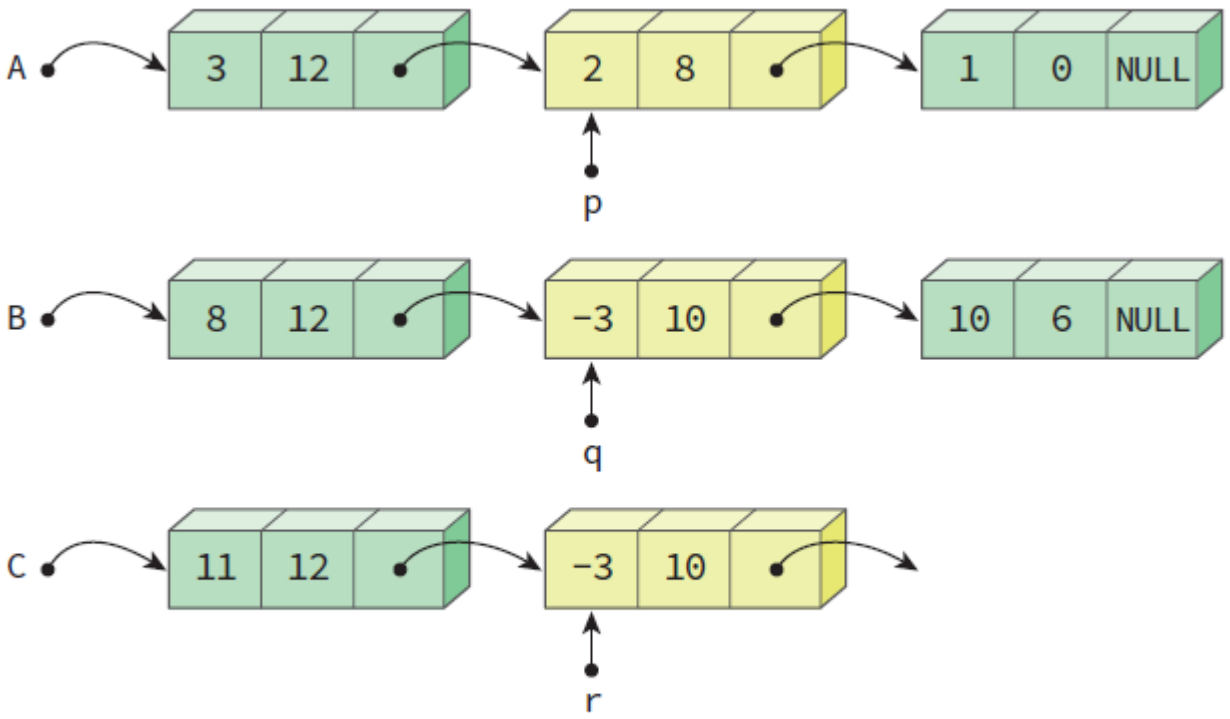
- $A = 3x^{12} + 2x^{8} + 1$, $B = 8x^{12} - 3x^{10} + 10x^{6}$이면
- $A + B = 11x^{12} - 3x^{10} + 2x^{8} + 10x^{6} + 1$

다항식의 덧셈 연산 (1/3)



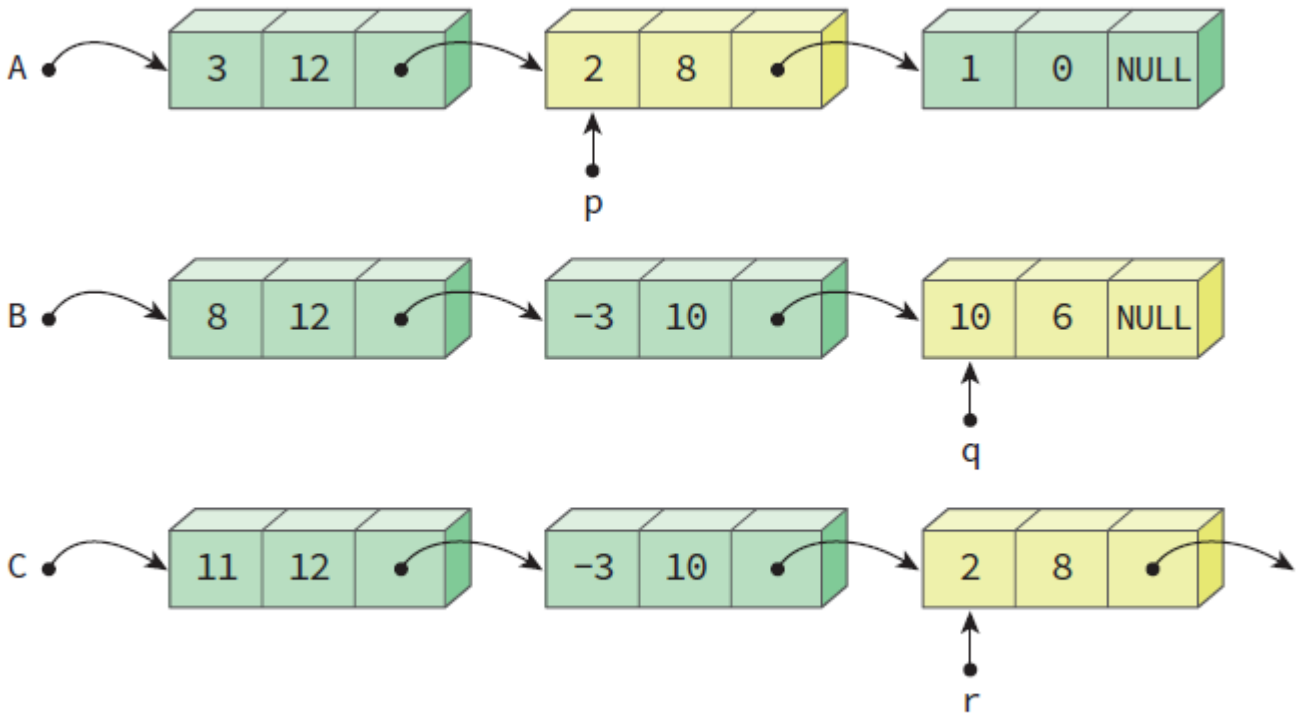
(a) p와 q가 가리키는 항들의 지수가 같으면 계수를 더한다.

다항식의 덧셈 연산 (2/3)



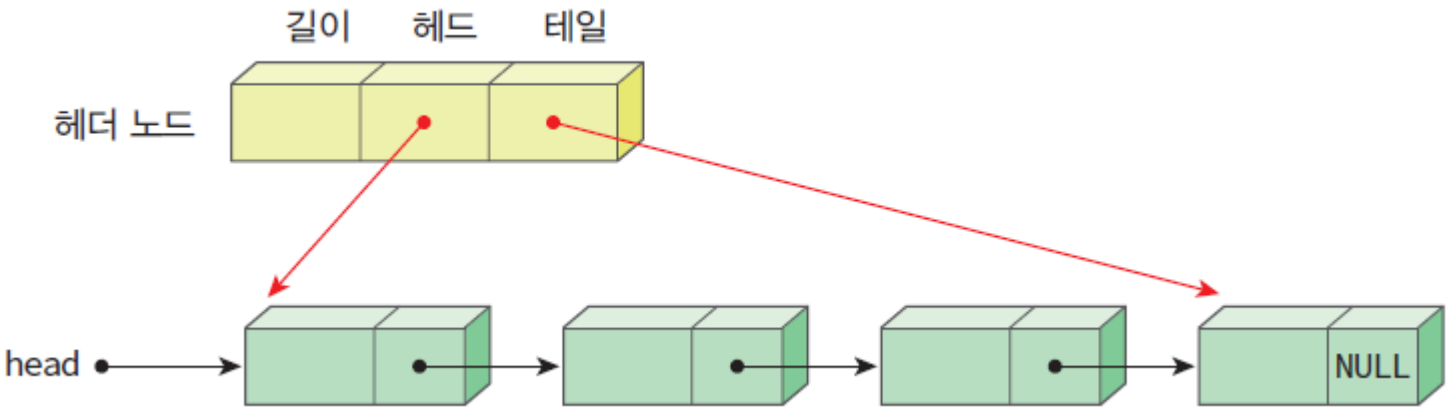
(b) q가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.

다항식의 덧셈 연산 (3/3)



(c) p가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.

헤더 노드의 기능 확장



다항식 프로그램: polynomial.c (1/4)


```
 4  typedef struct ListNode { // 노드 타입
 5      int coef;
 6      int expon;
 7      struct ListNode *link;
 8  } ListNode;
 9
10  // 연결 리스트 헤더
11  typedef struct ListType { // 리스트 헤더 타입
12      int size;
13      ListNode *head;
14      ListNode *tail;
15  } ListType;
```

```
24  // 리스트 헤더 생성 함수:
25  ListType* create()
26  {
27      ListType *plist = (ListType *)malloc(sizeof(ListType));
28      plist->size = 0;
29      plist->head = plist->tail = NULL;
30      return plist;
31  }
```

다항식 프로그램: polynomial.c (2/4)


```c
// plist는 연결 리스트의 헤더를 가리키는 포인터, coef는 계수,
// expon는 지수
void insert_last(ListType* plist, int coef, int expon)
{
    ListNode* temp =
        (ListNode *)malloc(sizeof(ListNode));
    if (temp == NULL) error("메모리 할당 에러");
    temp->coef = coef;
    temp->expon = expon;
    temp->link = NULL;
    if (plist->tail == NULL) {
        plist->head = plist->tail = temp;
    }
    else {
        plist->tail->link = temp;
        plist->tail = temp;
    }
    plist->size++;
}
```

```c
// 다항식 리스트 출력
void poly_print(ListType* plist)
{
    ListNode* p = plist->head;

    printf("polynomial = ");
    for (; p; p = p->link) {
        printf("%d^%d + ", p->coef, p->expon);
    }
    printf("\n");
}
```

다항식 프로그램: polynomial.c (3/4)


```
53  // list3 = list1 + list2
54  void poly_add(ListType* plist1, ListType* plist2, ListType* plist3)
55  {
56      ListNode* a = plist1->head;
57      ListNode* b = plist2->head;
58      int sum;
59
60      while (a && b) {
61          if (a->expon == b->expon) {   // a의 차수 > b의 차수
62              sum = a->coef + b->coef;
63              if (sum != 0) insert_last(plist3, sum, a->expon);
64              a = a->link; b = b->link;
65          }
66          else if (a->expon > b->expon) {  // a의 차수 == b의 차수
67              insert_last(plist3, a->coef, a->expon);
68              a = a->link;
69          }
70          else {                    // a의 차수 < b의 차수
71              insert_last(plist3, b->coef, b->expon);
72              b = b->link;
73          }
74      }
75
76      // a나 b중의 하나가 먼저 끝나게 되면 남아있는 항들을 모두
77      // 결과 다항식으로 복사
78      for (; a != NULL; a = a->link)
79          insert_last(plist3, a->coef, a->expon);
80      for (; b != NULL; b = b->link)
81          insert_last(plist3, b->coef, b->expon);
82  }
```

다항식 프로그램: polynomial.c (4/4)


```c
// 다항식 덧셈 테스트 프로그램
int main(void)
{
    ListType *list1, *list2, *list3;

    // 연결 리스트 헤더 생성
    list1 = create();
    list2 = create();
    list3 = create();

    // 다항식 1을 생성
    insert_last(list1, 3, 12);
    insert_last(list1, 2, 8);
    insert_last(list1, 1, 0);

    // 다항식 2를 생성
    insert_last(list2, 8, 12);
    insert_last(list2, -3, 10);
    insert_last(list2, 10, 6);

    poly_print(list1);
    poly_print(list2);

    // 다항식 3 = 다항식 1 + 다항식 2
    poly_add(list1, list2, list3);
    poly_print(list3);

    free(list1); free(list2); free(list3);
}
```

```
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 +
```