# 제 6 강의 . 연결 리스트

1. 포인터 타입

2. 단순 연결리스트

3. 연결리스트를 이용한 스택의 구현

4. 연결리스트를 이용한 큐의 구현

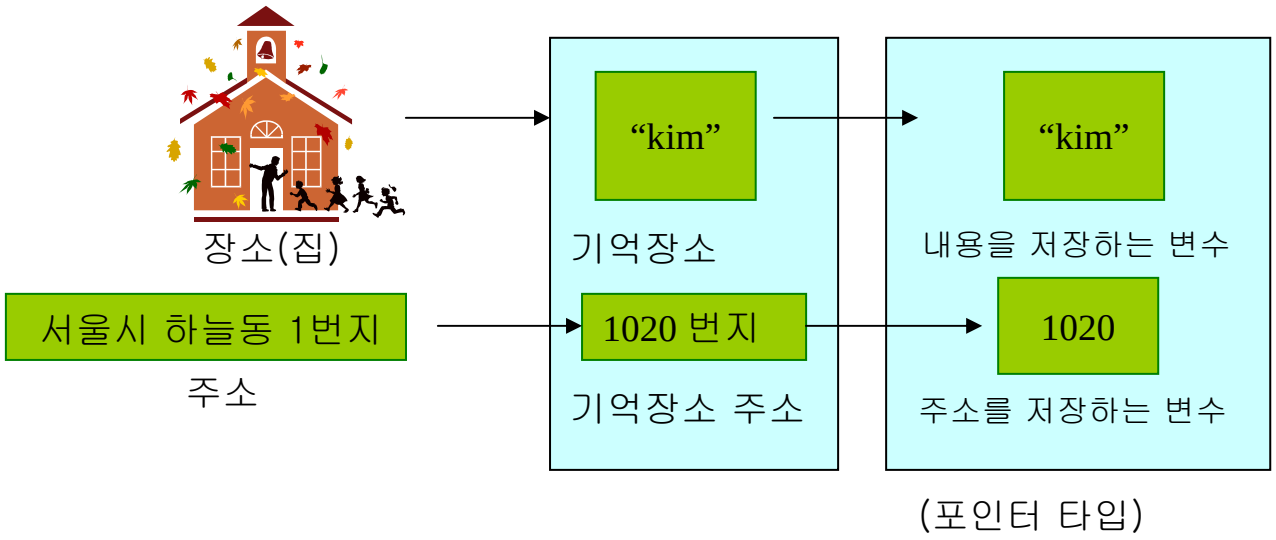
5. 리스트 구현방법의 정리

# 1. 포인터 타입(Pointers)

연결리스트는 기억장소의 임의의 구조체로 구현된 노드를 서로 연결하여 리스트를 만든다. 따라서 다음 데이터가 저장된 노드의 기억장소를 저장하는(가리키는) 값이 필요하다. 기억장소의 주소 값을 저장하는 데이터 타입을 "포인터(pointer) 타입"이라 한다

- **포인터 타입이란?**

**일상에서의 장소 -> 컴퓨터의 기억장소 -> 컴퓨터의 기억변수**

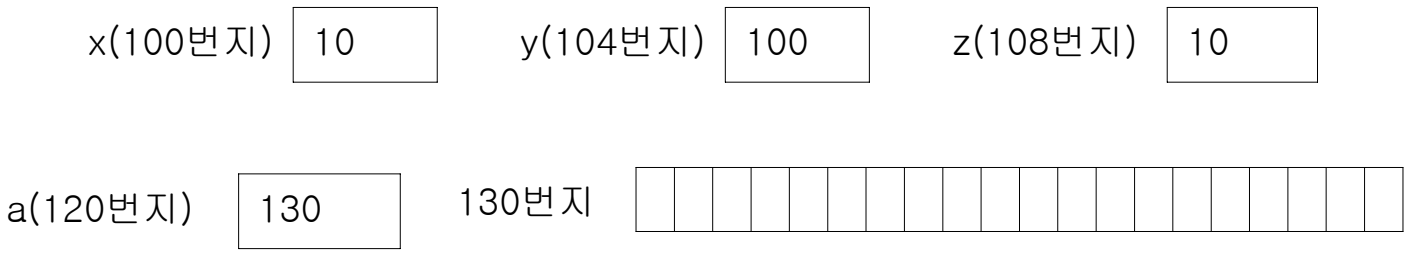


## 포인터 타입 사용 예 1

```
int x;          /* 내용을 저장하는 정수형 변수 */
int *y;         /* 주소를 저장하는 정수형 변수 */
int z;

x= 10;          /* 내용 10을 저장 */
y = &x;         /* x의  주소를 저장, x의 주소값을 알 필요 없다.*/
z = *y;         /* 주소 y가 가리키는 곳의 내용을 저장 */
                /* z = x와 같은 의미가 된다. */

int a[20];      /* 배열 변수 a는 주소 값으로 처리한다. */
                /* 그 이유는 배열의 내용 전체를 이동할 경우 보다
                   주소 값을 알려주면 편하기 때문이다. */
```

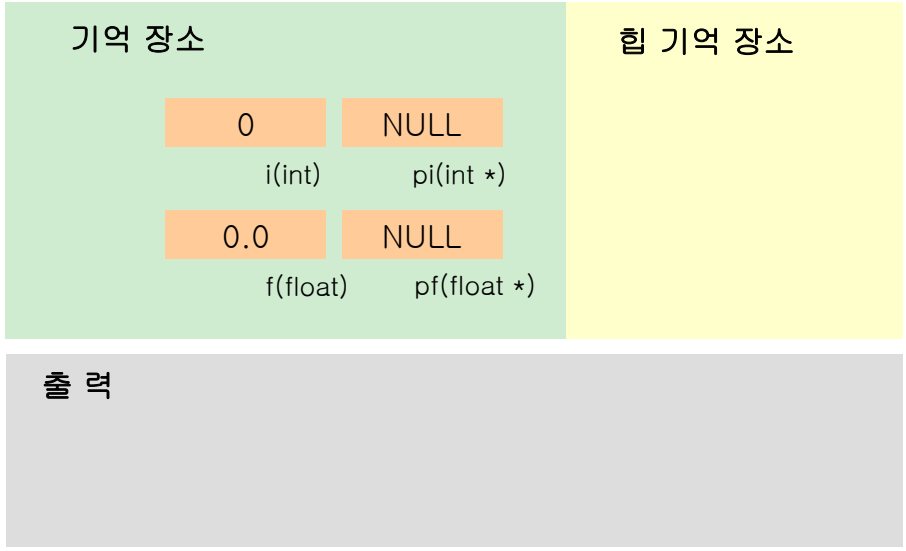


## 포인터 사용의 예 2

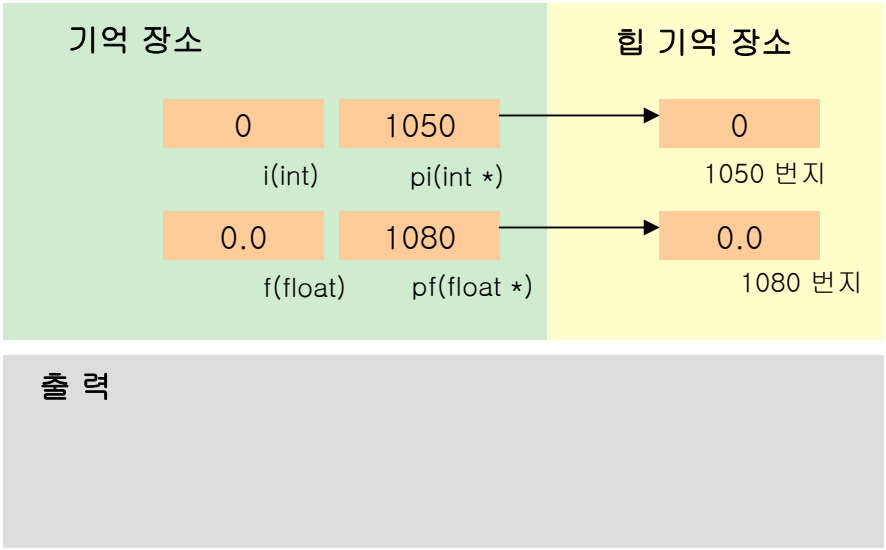
– 포인터 변수와 포인터가 가리키는 기억장소 확보하여 데이터를 저장하고 기억장소 반납

```
 0:  int main( )
 0:  {
 1:      int i,*pi;
 2:      float f,*pf;
 3:      pi = (int *)malloc(sizeof(int));
 4:      pf = (float *)malloc(sizeof(float));
 5:      *pi = 1024;
 6:      *pf = 3.14;
 7:      printf(“an integer=%d,a float=%f\n”,*pi,*pf);
 8:      free(pi);
 9:      free(pf);
10:   }
```

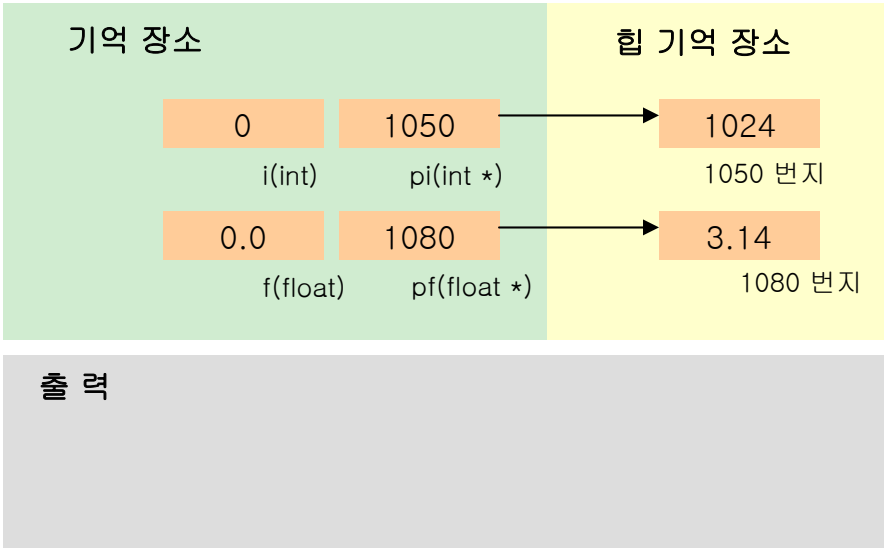
```
실행문장 =>    1:      int i,*pi;
               2:      float f,*pf;
                       /* 변수 i, pi, f, pf의 기억장소가 할당된다. */
```



```
실행문장 => 3:      pi = (int *)malloc(sizeof(int));
            4:      pf = (float *)malloc(sizeof(float));
                      /* 변수 pi, pf에 힙에서 임의의 기억장소가 할당되어
                       각각 정수형, 실수형 포인터로 변환되어 주소값을 저장한다 */
```

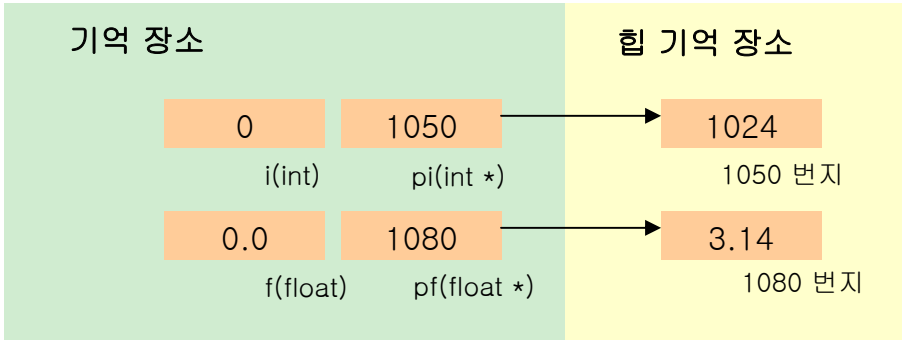


```
실행문장 => 5:        *pi = 1024;
            6:        *pf = 3.14;
                        /* 변수 pi, pf가 가리키는 곳에 1024와 3.14를 저장한다.*/
```



실행문장 => 7:

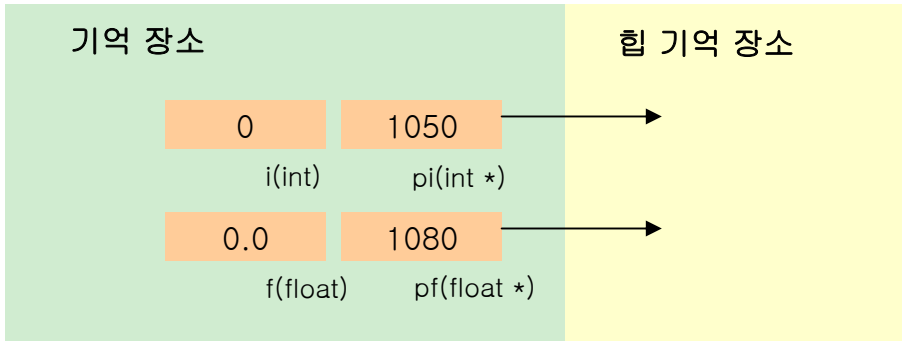
```
printf("an integer=%d,a float=%f\n",*pi,*pf);
/* 변수 pi, pf가 가리키는 곳의 값을 출력한다.*/
```




**출 력**

```
an integer=1024,a float=3.14
```

```
실행문장 => 8:      free(pi);
            9:      free(pf);
                      /* 변수 pi, pf가 가리키는 기억장소를 반환한다.
                      pi, pf가 가리키는 값은 남아있지만 사용하면 error 이다.*/
```



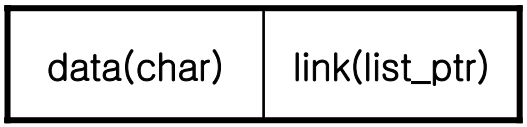

**출 력**

```
an integer=1024,a float=3.14
```

## 연결리스트를 만들기 위한 구조체 선언

- (자기참조구조체, self referential structure, **a pointer to itself**)
- (동적인 자료의 할당과 해지(allocation/deallocation)를 위한 자료구조)

```
struct node {
        char data;
        struct node * link;
};
typedef struct node list_node;
typedef list_node * list_ptr;
```

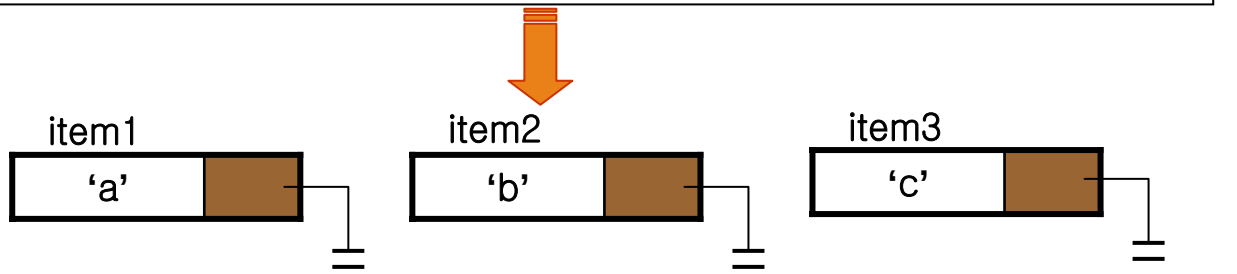



**list** 타입

## 리스트 자료의 예

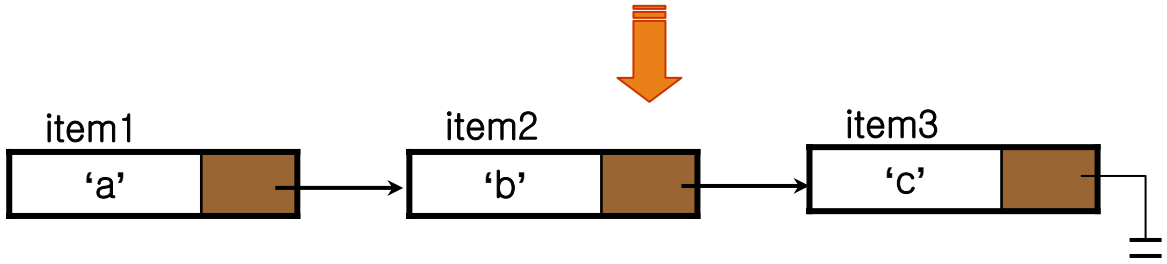
```
list item1, item2, item3;

item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```




## 리스트의 연결 예

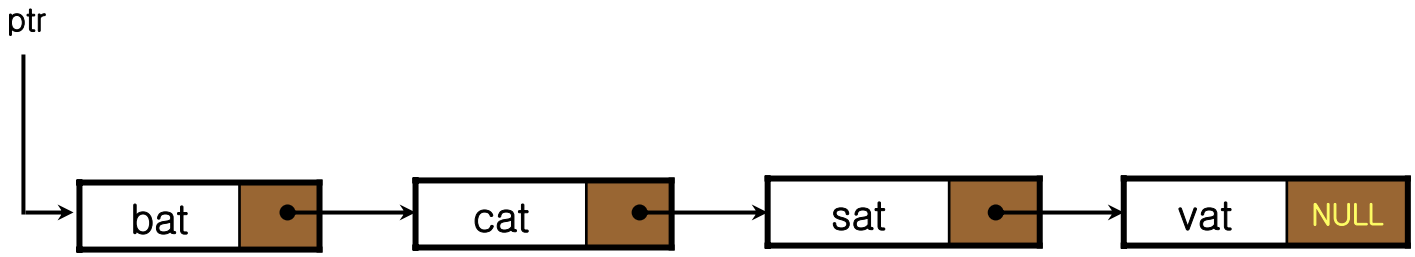
```
item1.link=&item2;
item2.link=&item3;
```



## 2. 단순 연결리스트(Singly Linked Lists)

- 연결리스트의 가장 간단한 형태로 그냥 "연결리스트"라고 부른다. 노드들을 연결한 형태이며 노드는 데이터부분과 링크 부분으로 구성된다.
- 데이터 부분은 한 개 혹은 여러 개의 속성을 저장할 수 있다. 링크 부분은 다음 노드의 주소값을 가리킨다.
- 동적 기억 할당과 해제가 가능하다.

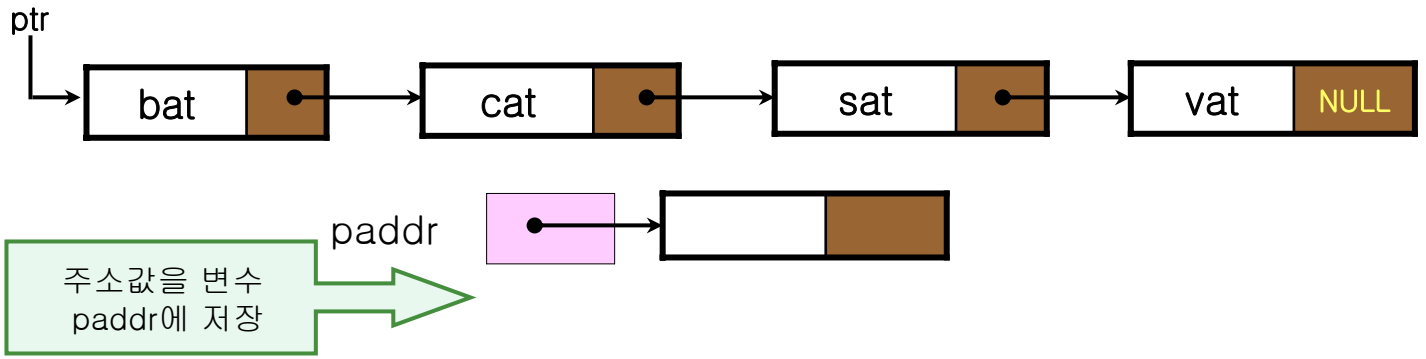
단순연결리스트



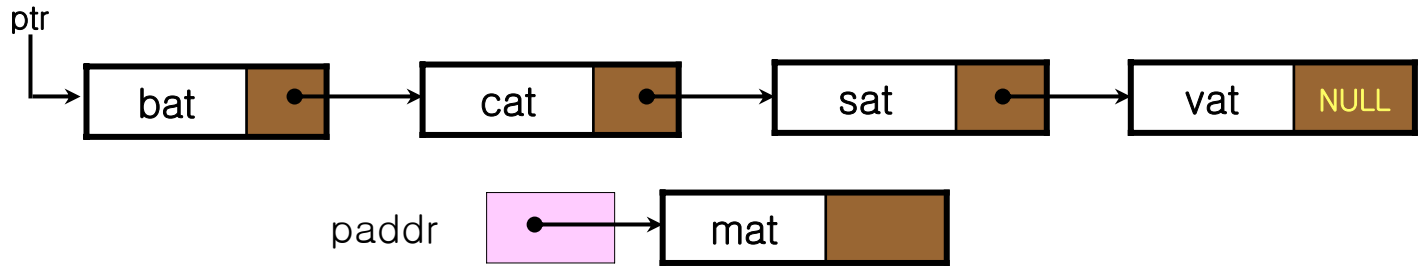
## 단순연결리스트 – 삽입 알고리즘

**"cat"과 "sat" 사이에 "mat"을 삽입한다면?**

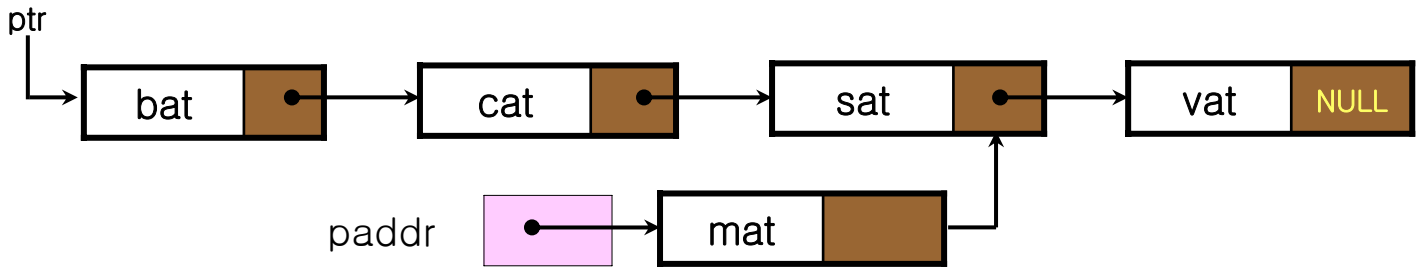
1) 기억장소로부터 노드를 얻는다.(주소값 : paddr)



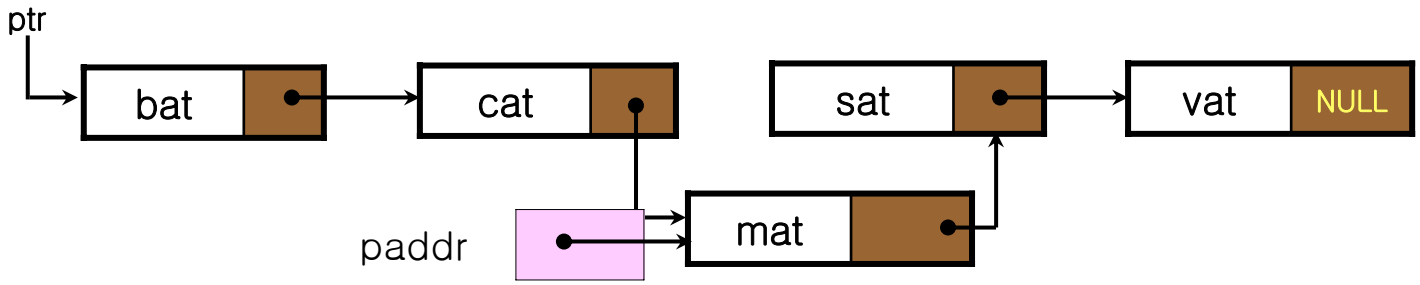

2) 노드에 "mat" 데이터를 저장한다.



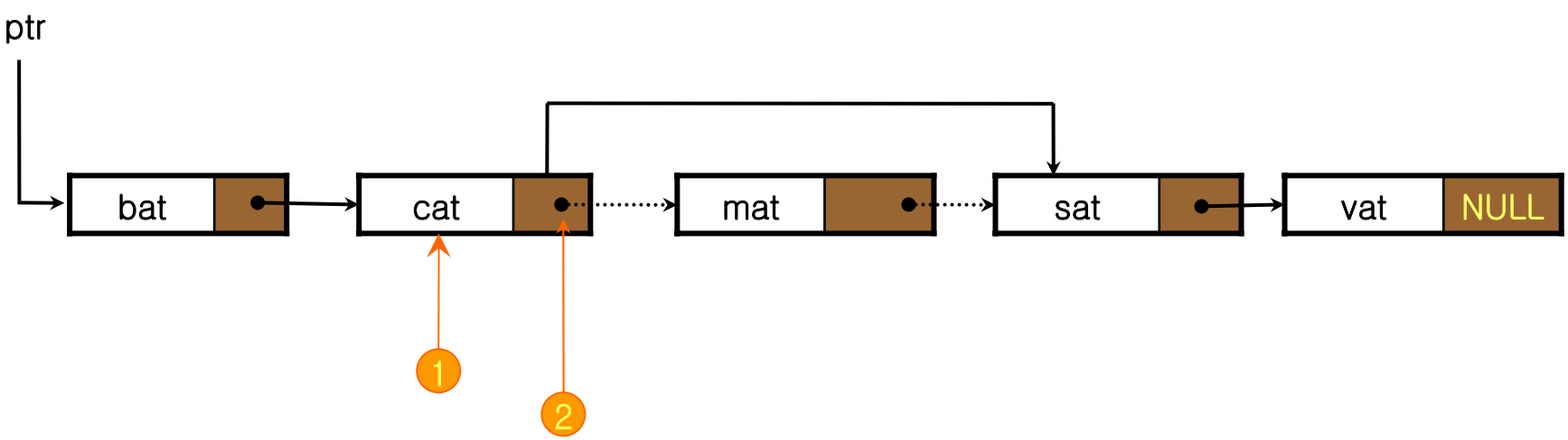
3) "*paddr*" 노드의 링크값에 "cat" 노드의 링크 값을 저장한다.




4) "cat" 노드의 링크 값에 paddr을 저장한다.



## 단순연결리스트 – 삭제 알고리즘




1) 삭제할 “mat” 노드의 앞 노드 “cat”을 찾는다.

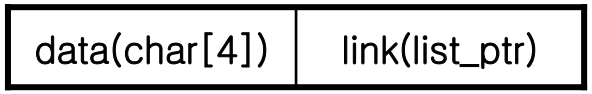
2) “cat” 노드의 링크를 “mat” 노드의 링크가 가리키는 데이터로 바꾼다.

## 단순연결리스트의 예

### 예1) 단순 연결리스트 노드의 선언과 생성

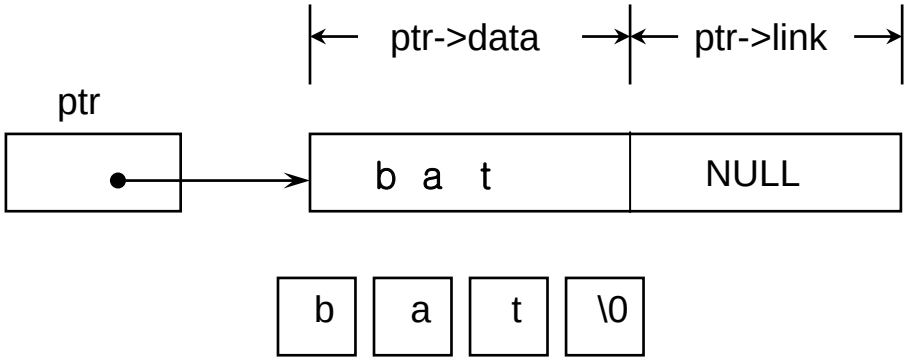
```
struct node {
        char data[4];
        struct node * link;
};
typedef struct node list_node;
typedef list_node * list_ptr;
```

list_node 타입의 구조



예2) 빈 리스트에 노드를 한 개 생성하여 “bat”를 저장

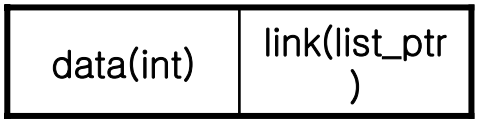
```
list_ptr ptr = NULL;
ptr=(list_ptr)malloc(sizeof(list_node));
strcpy(ptr->data,”bat”);
ptr->link=NULL;
```



## 예 3) 연결리스트의 생성(정수형 데이터 저장)

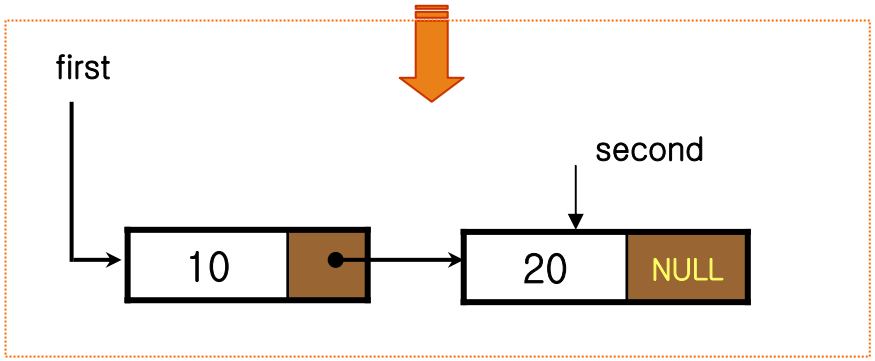
```
struct node {
        int data;
        struct node * link;
};
typedef struct node list_node;
typedef list_node * list_ptr;
```

list_node 타입의 구조

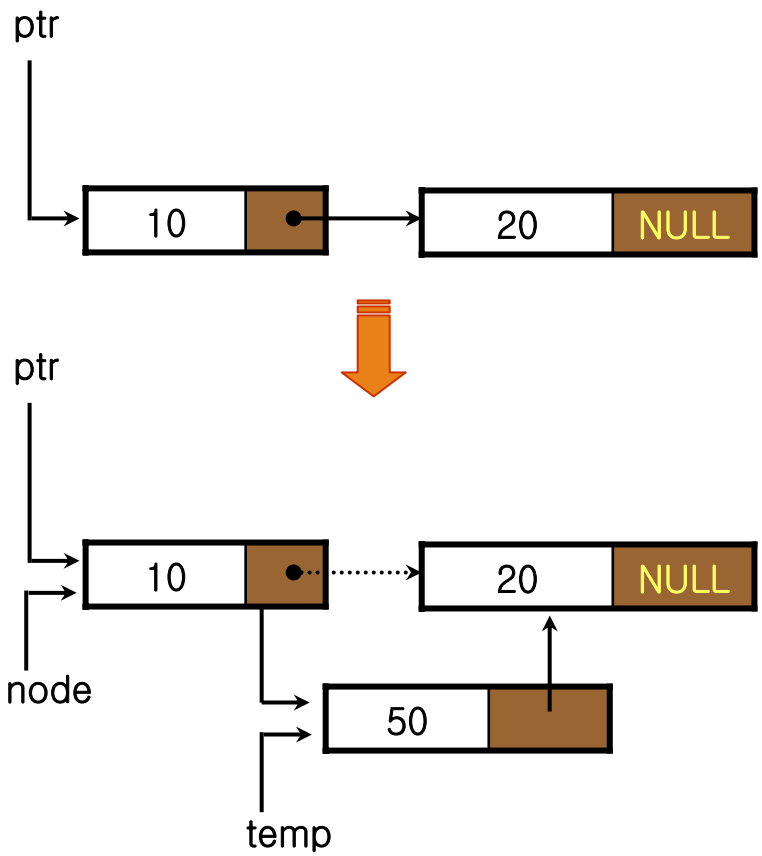


## 예 4) 연결리스트의 생성(정수형 데이터 저장)

```
list_ptr create() {
    list_ptr first, second;
    first =  (list_ptr)malloc(sizeof(list_node));
    second = (list_ptr)malloc(sizeof(list_node));
    second->link=NULL;
    second->data=20;
    first->data=10;
    first->link=second;
    return first;
}
```



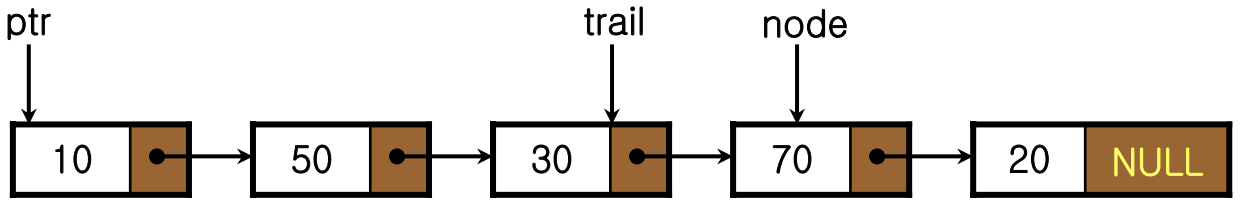
## 예 5) 노드 삽입



```
/* node 변수가 가리키는 곳 다음에 데이터 50을 저장하는
노드를 삽입하는 프로그램이다. */
void insert(list_ptr *ptr,list_ptr node) {
     list_ptr temp;
     temp=(list_ptr)malloc(sizeof(list_node));
     if(!temp ) {
          fprintf(stderr,"The momory is full\n");
          exit(1);
     }
     temp->data=50;
     if(*ptr) {
          temp->link = node->link;
          node->link = temp;
     }
     else {
          temp->link = NULL; *ptr = temp;
     }
}
```

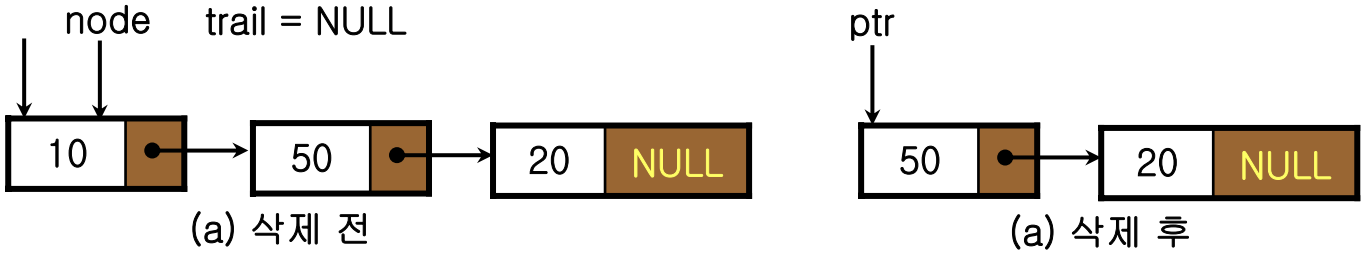
## 예 6) 노드삭제

- *ptr*: 리스트 첫 노드를 가리키는 포인터 변수
- *node*: 삭제될 노드를 가리키는 변수
- *trail*: 삭제될 노드의 바로 앞 노드를 가리키는 변수

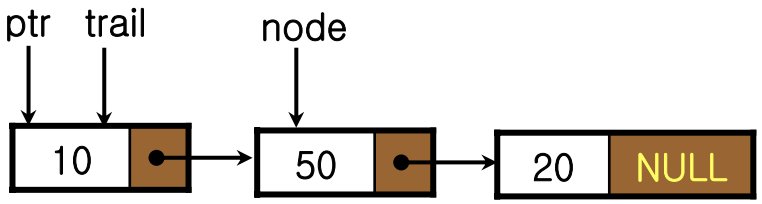



삭제될 노드가 첫 노드일 경우(삭제 후 ptr 값이 바뀌는 경우)와 그렇지 않은 경우로 나눈다.

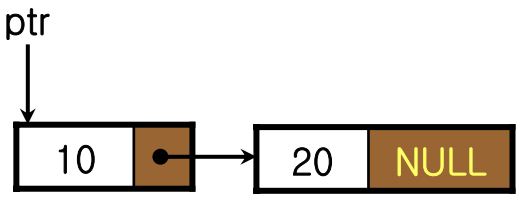
1)삭제될 노드가 리스트의 첫 노드일 때 - 리스트 시작인 ptr 값이 바뀐다



2) 그 외의 경우 리스트 시작인 ptr 값이 안바뀐다



(a) 삭제 전



(a) 삭제 후

```
void delete(list_ptr *ptr, list_ptr trail, list_ptr node)
{
    if(trail) /* 노드가 여러 개 있는 경우 */
        trail->link = node->link;
    else     /* 노드가 한 개 밖에 없는 경우 */
        *ptr = (*ptr)->link;
    free(node);
}
```

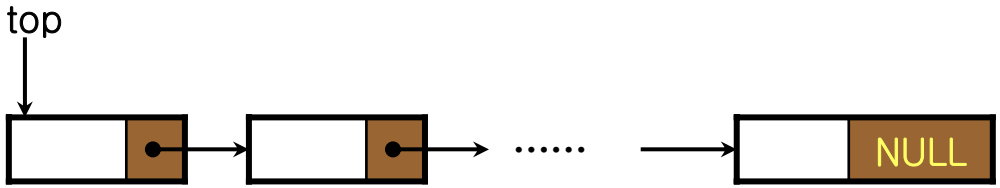
## 예 7) 리스트의 출력 알고리즘

```
while not end of the list do
    print out data field;
    move to the next node;
end;
```

```
void print_list(list_ptr ptr) {
    printf("The list contains: ");
    for( ; ptr; ptr = ptr->link)
        printf("%4d", ptr->data);
    printf("\n");
}
```

# 3. 연결리스트를 이용한 스택과 큐의 구현

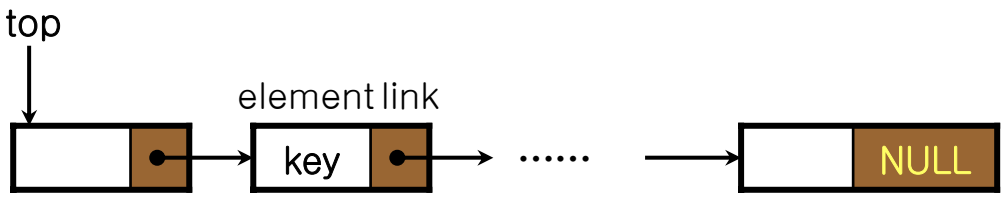
## 3.1.연결리스트를 이용한 스택의 구현(Dynamically Linked Stacks)




연결리스트 스택

```
#define MAX_STACKS 10 /* n=MAX_STACKS=10 */
typedef struct {
    int key;
    /* other fields */
} element;
typedef struct stack *stack_ptr;
typedef struct stack {
    element item;
    stack_ptr link;
};
stack_ptr top;
```

연결리스트를 이용한 스택 구조의 선언




스택의 초기 상태

top = NULL

스택의 상태

- 비어있는 스택 :

  top = NULL

- 꽉 찬 상태

  malloc( ) 함수 호출 시 기억 장소 공간이 부족할 때

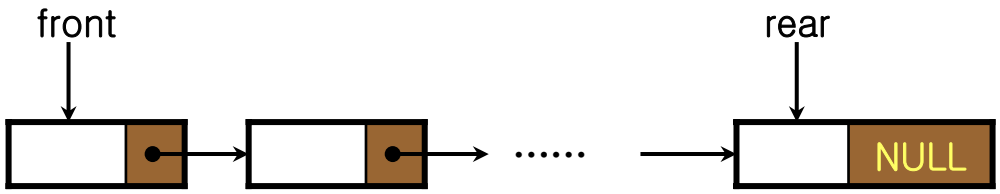
스택의 push() 함수

```
void push(stack_ptr *top, element item) {
    stack_ptr temp =
        (stack_ptr)malloc(sizeof (stack));
    if( !temp) {
        fprintf(stderr,”The memory is
        full\n”);
        exit(1);
    }
    temp->item=item;
    temp->link=*top;
    *top = temp;
}
```

스택의 pop( ) 함수

```
element pop(stack_ptr *top) {
    stack_ptr temp = *top;
    element item;
    if( !temp ) {
        fprintf(stderr,"The stack is
        empty\n");
        exit(1);
    }
    item=temp->item;
    *top=temp->link;
    free(temp);
    return item;
}
```

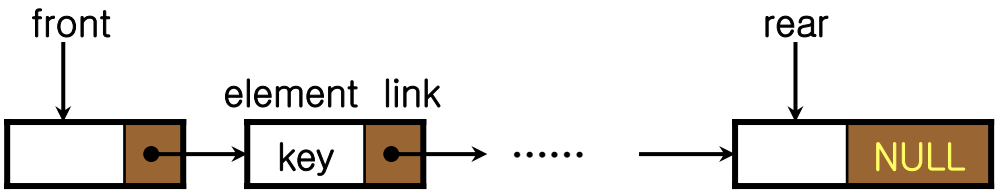
## 3.2 .연결리스트를 이용한 큐의 구현 (Dynamically Linked Queues)




연결리스트 큐

- 연결리스트를 이용한 1개의 큐 구현

```
#define MAX_QUEUES 10 /* m=MAX_QUEUES=10 */
typedef struct queue *queue_ptr;
typedef struct queue {
    element item;
    queue_ptr link;
};
queue_ptr front, rear;
```




큐의 초기 상태

front = rear = NULL

큐의 상태

- 비어있는 큐 :

  front = NULL

- 꽉 찬 상태 :

  malloc( ) 함수 호출 시 기억장소 확보가 안될 때

연결리스트 큐에 삽입 함수
– insert( )

```
void insert(queue_ptr *front,queue_ptr *rear,
element x) {
    queue_ptr temp =
        (queue_ptr)malloc(sizeof(queue));
    if( !temp ) {
        fprintf(stderr,"The memory is full\n");
        exit(1);
    }
    temp->item=x;
    temp->link=NULL;
    if(*front) (*rear)->link=temp;
    else *front = temp;
    *rear = temp;
}
```

연결리스트 큐에 삭제 함수
– delete( )

```
element delete(queue_ptr *front) {
    queue_ptr temp=*front;
    element x;
    if(!(*front)) {
        fprintf(stderr,"The queue is
        empty\n");
        exit(1);
    }
    x=temp->item;
    *front=temp->link;
    free(temp);
    return item;
}
```

# 4. 연결리스트를 이용한 다항식 문제 해결

연결리스트를 이용한 응용이 많이 있지만 대표적인 예로 다항식의 계산을 보도록 한다.

다항식은 $\mathbf{A(x) = 7x^3 + 4x^2 + \cdots + 2}$ 같은 식처럼 여러 개의 항(term)으로 구성된 식이다.

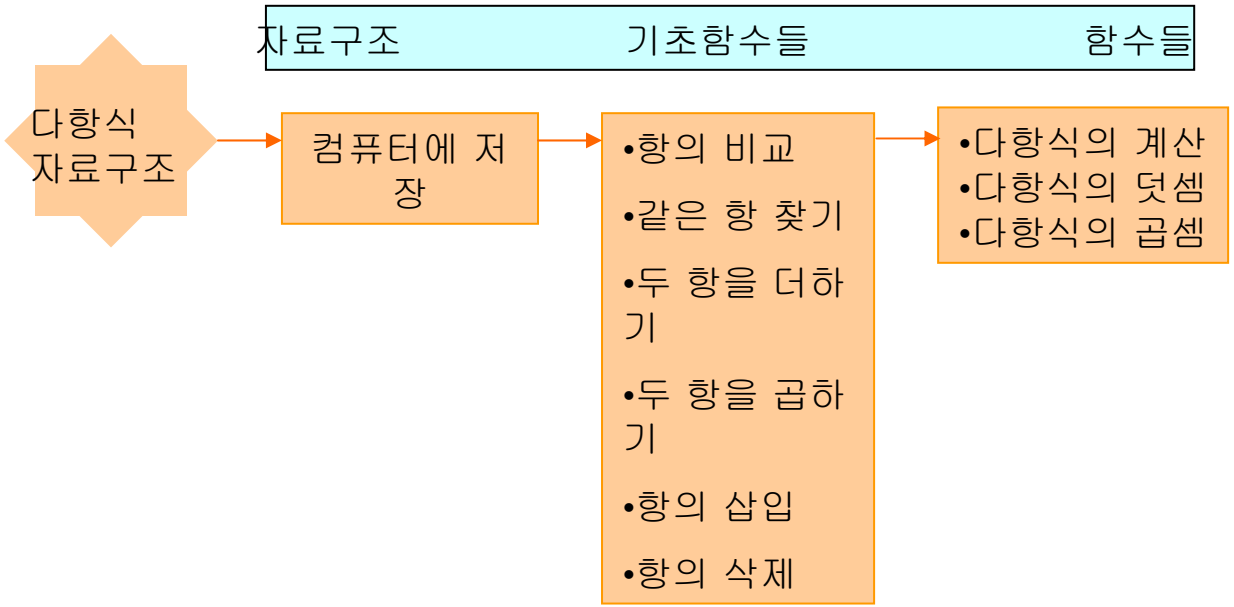
다항식에 관한 연산을 하기위해서 항의 검색, 삽입, 삭제가 필요하고 이러한 연산을 하기위한 자료구조로 연결리스트를 사용하면 편리하다.

- 두 개의 다항식의 덧셈을 통하여 연결리스트를 조작하는 프로그래밍 기법
- 연결리스트의 노드를 획득하고 해지하는 함수를 통하여 기억장소를 관리

## 4.1. 다항식(Polynomials)과 연결리스트

### 다항식이란?

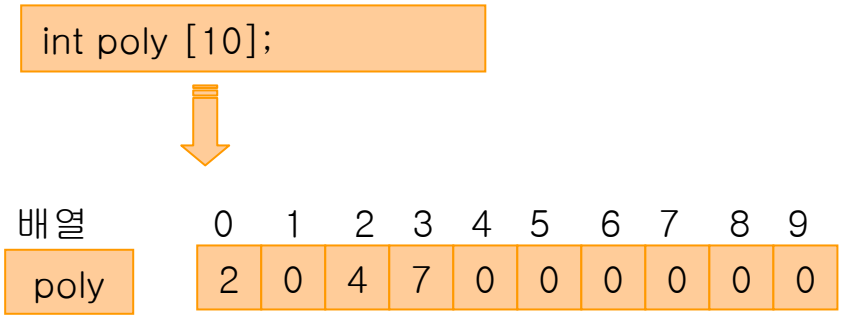
다항식은 $A(x) = 7x^3 + 4x^2 + \cdots + 2$ 같은 식처럼 여러 개의 항(term)으로 구성된 식이다. 다항식의 계산, 두개의 다항식의 덧셈, 곱셈 등에 관한 여러 가지 문제를 해결하려면 다항식을 컴퓨터 프로그램에 표현하여 저장하여야 한다. 저장된 다항식은 여러 가지 연산 과정을 거쳐서 계산을 하거나, 다항식끼리 덧셈, 다항식끼리 곱셈 등을 하게 된다. 이러한 문제를 효율적으로 해결하려면 다항식의 항들을 저장하는 방법을 잘 선택하여야 한다.



## 다항식을 배열에 표현하면?

예를 들어 다음 다항식 $7x^3 + 4x^2 + 2$ 은 배열에 다음 { (7,3), (4,2), (2,0) }을 저장하여야 한다.

예를 들어 다음과 같이 선언하고 계수(coefficient) 만을 배열에 저장하면 된다.



## 다항식을 연결 리스트에 표현하면?

다음 다항식을 배열로 표현하면 어떻게 될까?

$7x^{1000} + 4x^{23} + \cdots + 2$

- 방법 1 : 배열에 단순 저장
- 방법 2 : 연결리스트로 다항식의 항 { (7,1000), (4,23), (2,0) }을 저장

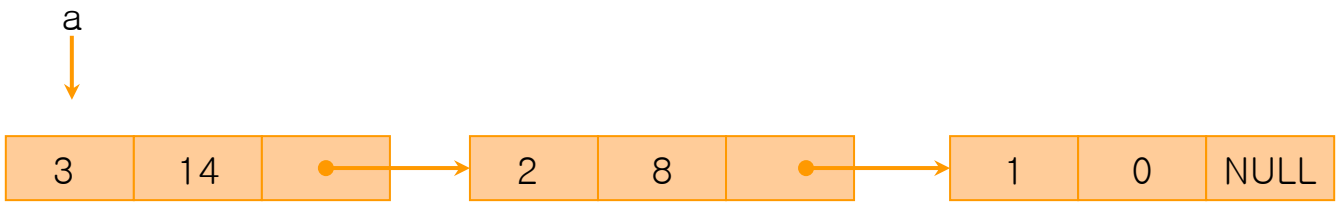
다항식을 연결리스트로 표현하면?

$A(x) = a_{m-1}x^{m-1} + \cdots + a_0x^0$

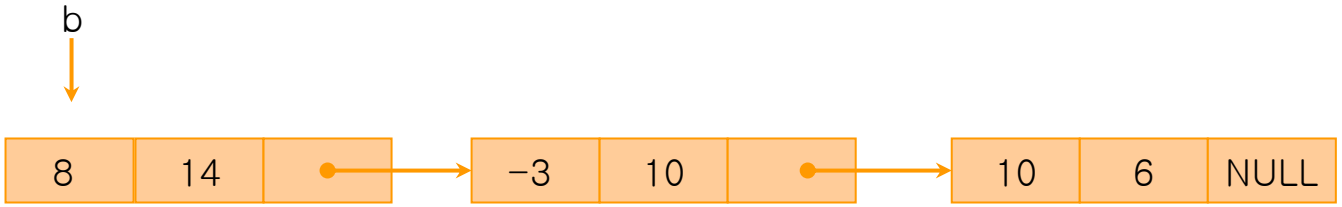
```
typedef struct poly_node *poly_ptr;
typedef struct poly_node {
     int coef;
     int expon;
     poly_ptr link;
};
poly_ptr a,b,d;
```

예 )

$a = 3x^{14} + 2x^{8} + 1$




$b = 8x^{14} - 3x^{10} + 10x^{6}$



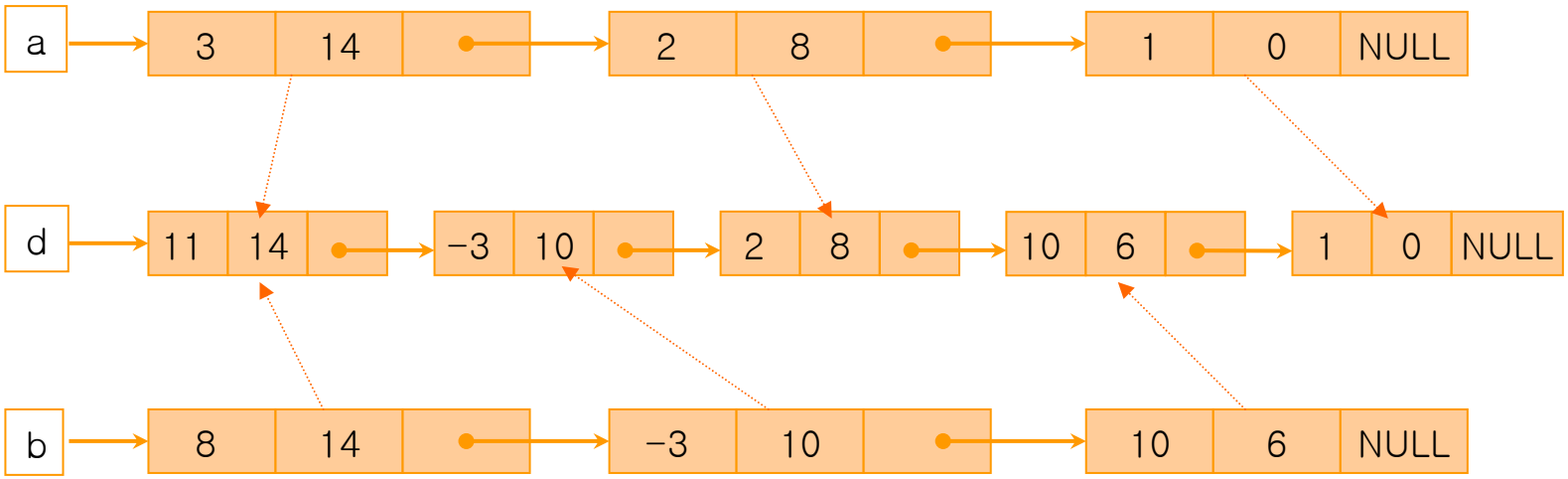
## 4.2. 연결리스트를 이용한 다항식의 덧셈

- 연결리스트로 표현된 두 개의 다항식을 더하는 프로그램을 작성하여 보자.

$$a = 3x^{14} + 2x^{8} + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

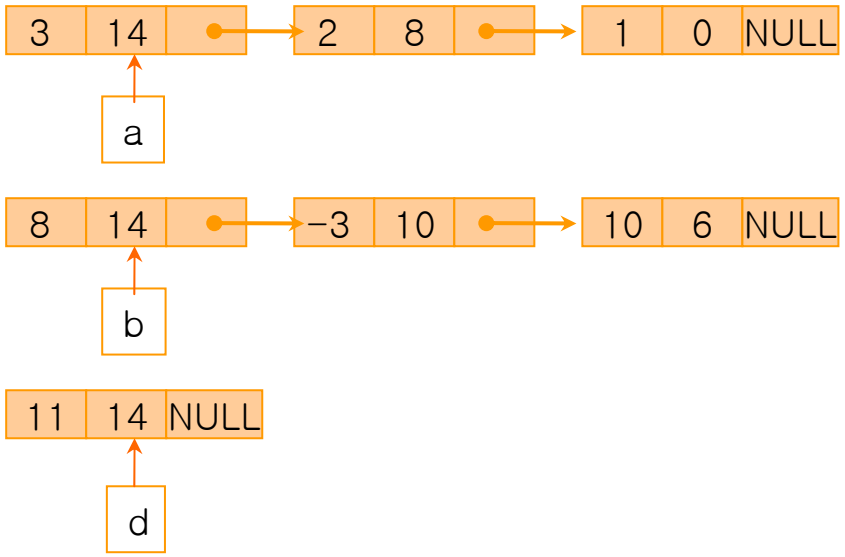
$$d = 11x^{14} - 3x^{10} + 2x^{8} + 10x^{6} + 1$$



연결리스트로 표현된 두 개의 다항식 a, b를 더하는 경우를 단계별로 보자.

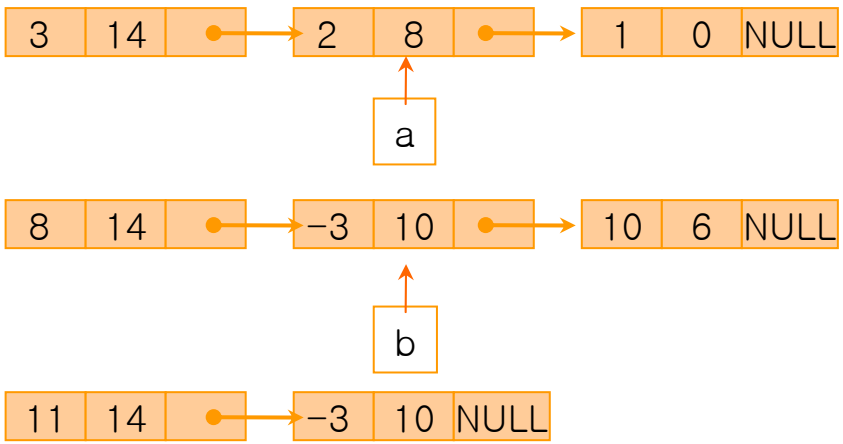
(a) a->expon == b->expon (지수가 같은 경우)

두 다항식 a,b의 최고차 항 두개를 비교하여 같으면 두 항의 계수를 더하여 새로운 다항식 d에 항을 만든다. 덧셈 후 포인터를 이동한다.



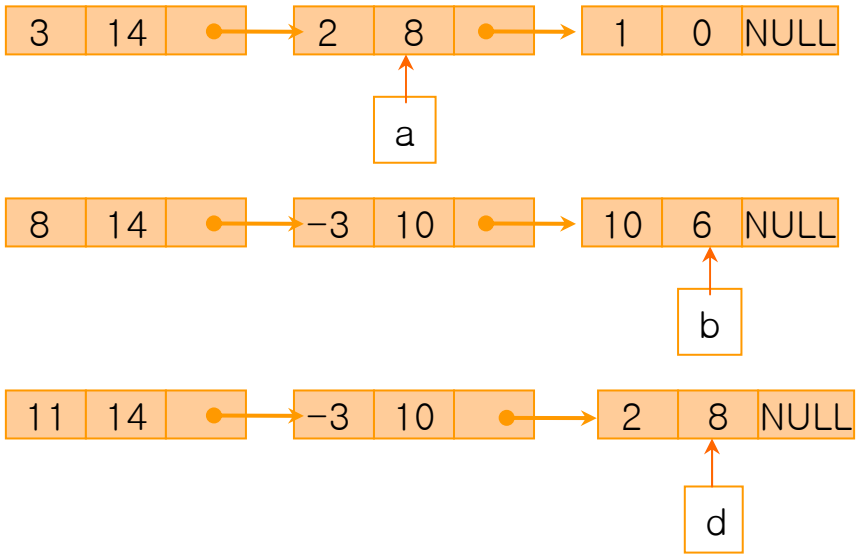
(b) a->expon < b->expon (b의 지수가 더 큰 경우)

두 다항식 a,b의 최고차 항 두개를 비교하여 같으면 항의 계수가 큰 항을 새로운 다항식 d에 더한다. 예의 경우 b가 가르키는 지수 값이 크므로 b의 항을 d에 더한다.



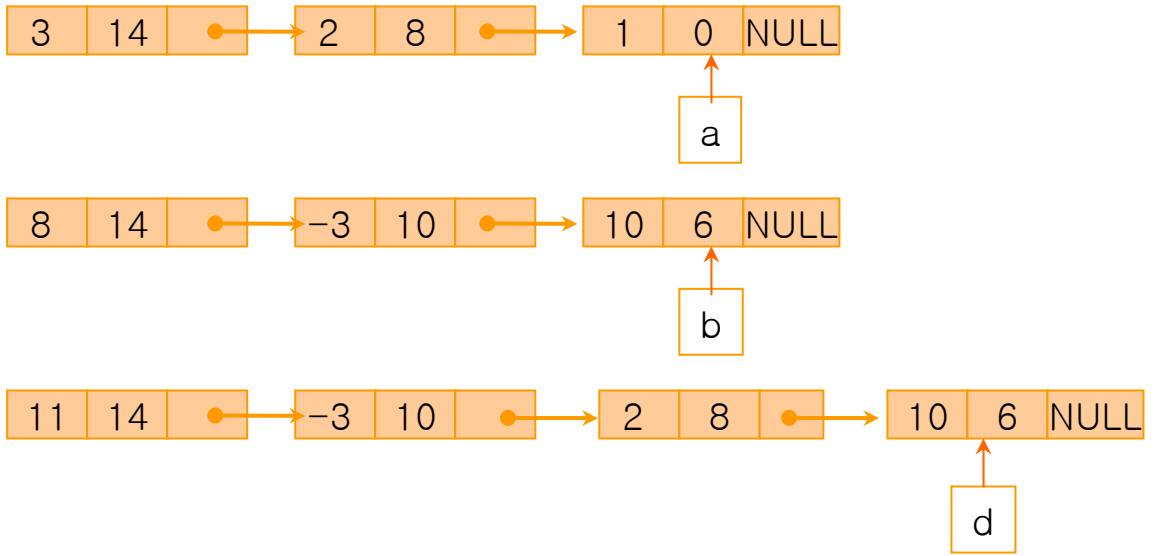
(c) a->expon > b->expon (a의 지수가 더 큰 경우)

두 다항식 a, b의 최고차 항 두개를 비교하여 같으면 항의 계수가 큰 항을 새로운 다항식 d에 더한다(예의 경우 a의 가르키는 지수 값이 크므로 a의 항을 d에 더한다).



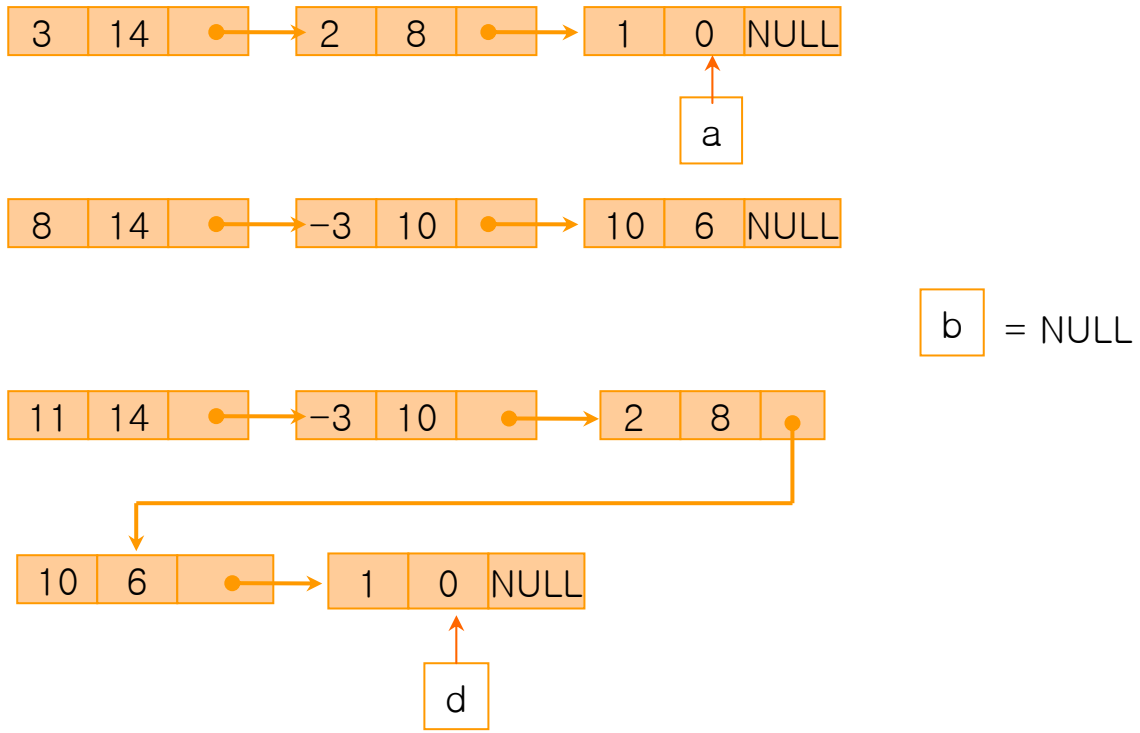
(d) a->expon < b->expon

두 다항식 a,b의 최고차 항 두개를 비교하여 같으면 항의 계수가 큰 항을 새로운 다항식 d에 더한다(예의 경우 b가 가르키는 지수 값이 크므로 b의 항을 d에 더한다).



(e) b == NULL;

두 다항식 a,b중 b의 항이 NULL이 되면 a의 남은 항을 d에 복사하여 연결한다(예의 경우 a가 가르키는 지수 값이 크므로 b의 항을 d에 더한다).



## 두 다항식의 덧셈

- 앞의 예를 참고하여 두 개의 다항식을 더하는 프로그램 padd()를 작성하였다.

padd() 함수

```
/* 두 개의 다항식을 더하는 함수, 인자는 두 다항식의
포인터, 결과는 새로 생성된 다항식의 포인터이다 */
poly_ptr padd(poly_ptr a,poly_ptr b) {
        poly_ptr front,rear,temp;
        int sum;
        rear=(poly_ptr)malloc(sizeof(poly_node));
        if(IS_FULL(rear)) {
                fprintf(stderr,"The memory is full₩n");
                exit(1);
        }
        front = rear;
        while(a && b)
                switch(COMPARE(a->expon,b->expon)) {
                case -1: /* a->expon < b->expon */
                        attach(b->coef,b->expon,&rear);
                        b = b->link;
                        break;
(계속)
```

```
            case 0: /* a->expon = b->expon */
                    sum = a->coef + b->coef;
                    if(sum) attach(sum,a->expon,&rear);
                    a = a->link; b = b->link; break;
            case 1: /* a->expon > b->expon */
                    attach(a->coef,a->expon,&rear);
                    a = a->link;
            }
        for( ; a; a=a->link)
                attach(a->coef,a->expon,&rear);
        for( ; b; b=b->link)
                attach(b->coef,b->expon,&rear);
        rear->link = NULL;
        temp=front; front=front->link; free(temp);
        return front;
}
```

## 리스트의 끝에 노드를 첨가

padd() 함수에서 필요한 다항식의 끝에 항을 붙이는 함수 attach() 이다.

attach 함수

```
/* 항을 다항식에 연결하는 함수, 인자는 항의 계수, 지수,
   다항식의 포인터이고 출력인자는 다항식 포인터 *ptr 이다 */
void attach(float coef, int exp, poly_ptr *ptr) {
        poly_ptr temp;
        temp=(poly_ptr)malloc(sizeof(poly_node));
        if(IS_FULL(temp)) {
                fprintf(stderr,"The memory is full₩n");
                exit(1);
        }
        temp->coef = coef;
        temp->expon = exp;
        (*ptr)->link = temp;
        *ptr=temp;
}
```

두 개의 다항식을 더하는 프로그램의 수행 시간은 ?

m, n : 두 다항식의 항의 수

- 계수 덧셈

  O(min{m, n})

- 지수 비교

  O(m + n)

- 다항식 d의 노드 생성 시간

  O(m + n)

전체 시간 복잡도

O(m + n)

## 4.3. 연결리스트 노드의 관리

다항식의 예에서 연산을 하다보면 필요없는 항이 만들어진다.
이러한 항은 기억장소에 다시 반환되어야 한다.

**예** 아래 예에서 임시로 계산된 다항식 temp의 모든 항은 기억장소에 반환되어야 한다.

$$e(x) = a(x) * b(x) + d(x)$$

(프로그램)

```
poly_ptr a, b, d, temp, e;
        . . .
a = read_poly();
b = read_poly();
d = read_poly();
temp = pmult(a, b);
e = padd(temp, d);
print_poly(e);
```

다항식을 가리키는 포인터가 인자로 주어지면 다항식에 포함된 항을 하나씩 모두 기억 장소에 반환하는 프로그램은 다음과 같다.

다항식의 삭제

erase 함수

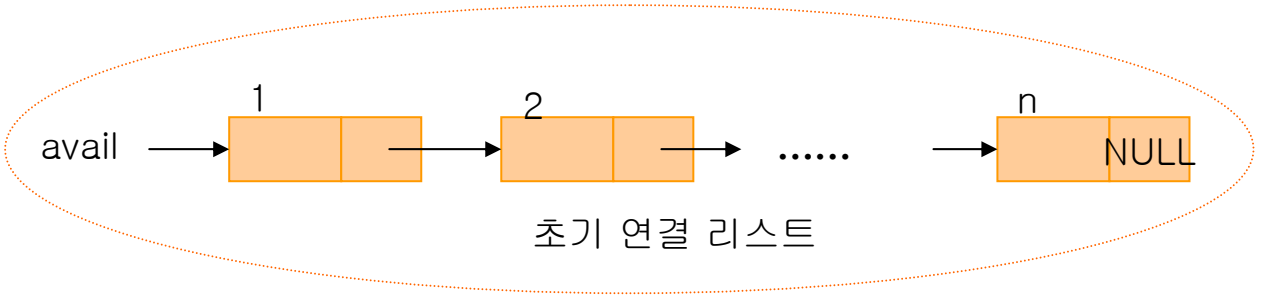
```
/* 다항식 항을 기억장소로 반환하는 함수 인자는
다항식의 포인터이다. */

void erase(poly_ptr *ptr) {
        poly_ptr temp;
        while (*ptr) {
                temp = *ptr;
                *ptr = (*ptr)->link;
                free(temp);
        }
}
```

## 노드(항)의 할당과 해지

다항식의 계산을 위해 자주 일어나는 노드의 생성과 해지를 위하여 노드를 저장하는 리스트(노드 창고)를 만들어 놓을 수도 있다.

- 처음에 모든 사용가능한 노드를 기억장소(연결리스트)에 모아 둔다.
- *avail* : 연결리스트의 처음을 가르키는 포인터




노드 저장소(Storage pool)

get_node 함수

```
/* 노드가 필요할 경우 노드저장소에서 노드 한 개를 를
반환해주는 함수 get_node() */
poly_ptr get_node(void) {
        poly_ptr node;
        if (avail) {
                node = avail; avail = avail->link;
        }
        else {
                node =
                        (poly_ptr)malloc(sizeof(poly_node));
                if(IS_FULL(node)) {
                        fprintf(stderr,"The memory is full₩n");
                        exit(1);
                }
        }
        return node;
}
```

ret_node 함수

```
/* 필요없는 노드를 반환 받아 노드저장소 avail에 저장하는
함수 ret_node() */
void ret_node(poly_ptr ptr) {
        ptr->link = avail;
        avail = ptr;
}
```

erase 함수

```
/* 필요없는 다항식 전체를 받아서 각 노드를 반복하여 avail
연결리스트에 반환하는 함수 */
void erase(poly_ptr *ptr) {
        poly_ptr temp;
        while (*ptr) {
                temp = *ptr;
                *ptr = (*ptr)->link;
                ret_node(temp);
        }
}
```

## 다항식의 노드를 쉽게 반환하려면?

다항식의 노드를 반환하는 함수 erase()는 O(n)의 시간이 소요된다.

좀 더 효율적인 방법으로 O(1)의 시간에 해결하려면?

- 다항식을 원형 연결리스트로 바꾼다.

  (마지막 노드의 링크가 리스트의 시작 노드를 가리킨다.)


- 효율적인 erase 알고리즘 -알고리즘 효율성 O(1)

cerase 함수

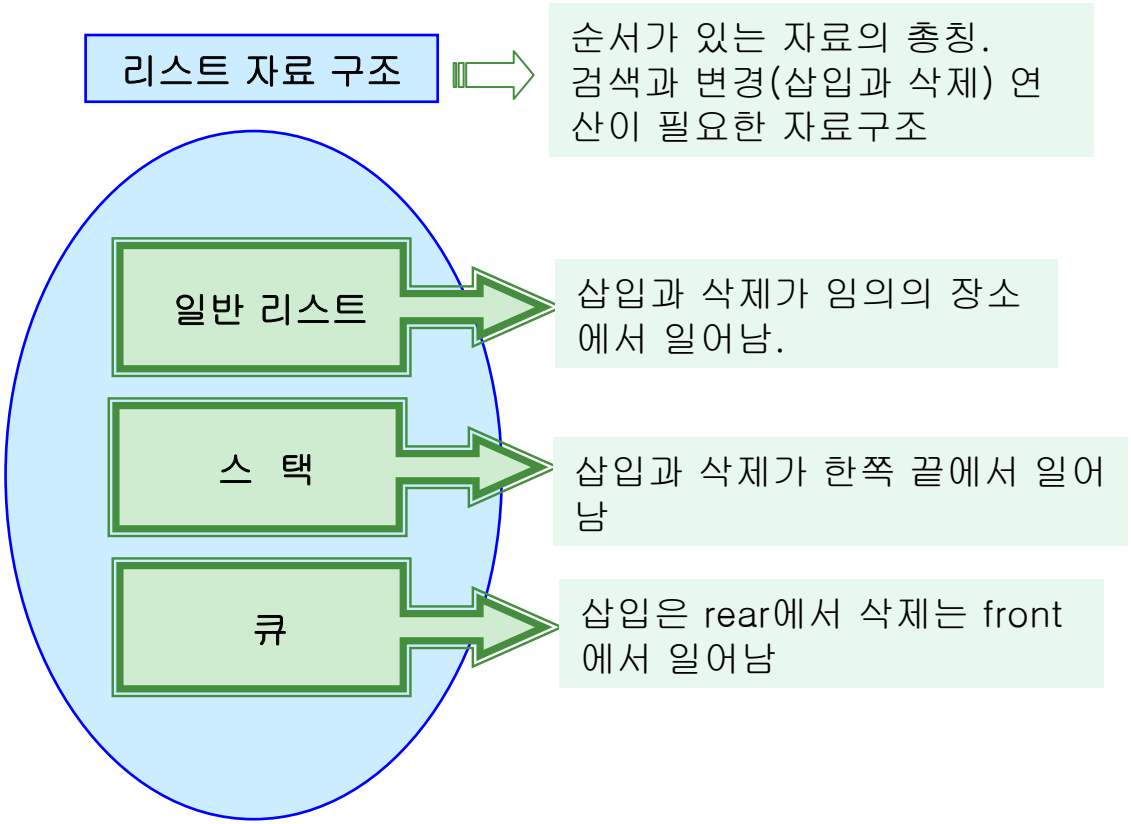
```
/* 필요없는 노드들을 한번에 원형
노드저장소에 반환하는 함수 */
void cerase(poly_ptr *ptr) {
        if(*ptr) {
                temp = (*ptr)->link;
                (*ptr)->link = avail;
                avail = temp;
                *ptr = NULL;
        }
}
```

# 5. 리스트와 연결리스트(List and Linked List)

- **리스트(list)**
  - 순서가 있는 데이터를 말한다.
  - 리스트 자료에 대한 연산은 검색, 변경(삽입, 삭제)이다.
  - 스택과 큐는 리스트의 특수한 형태이다.

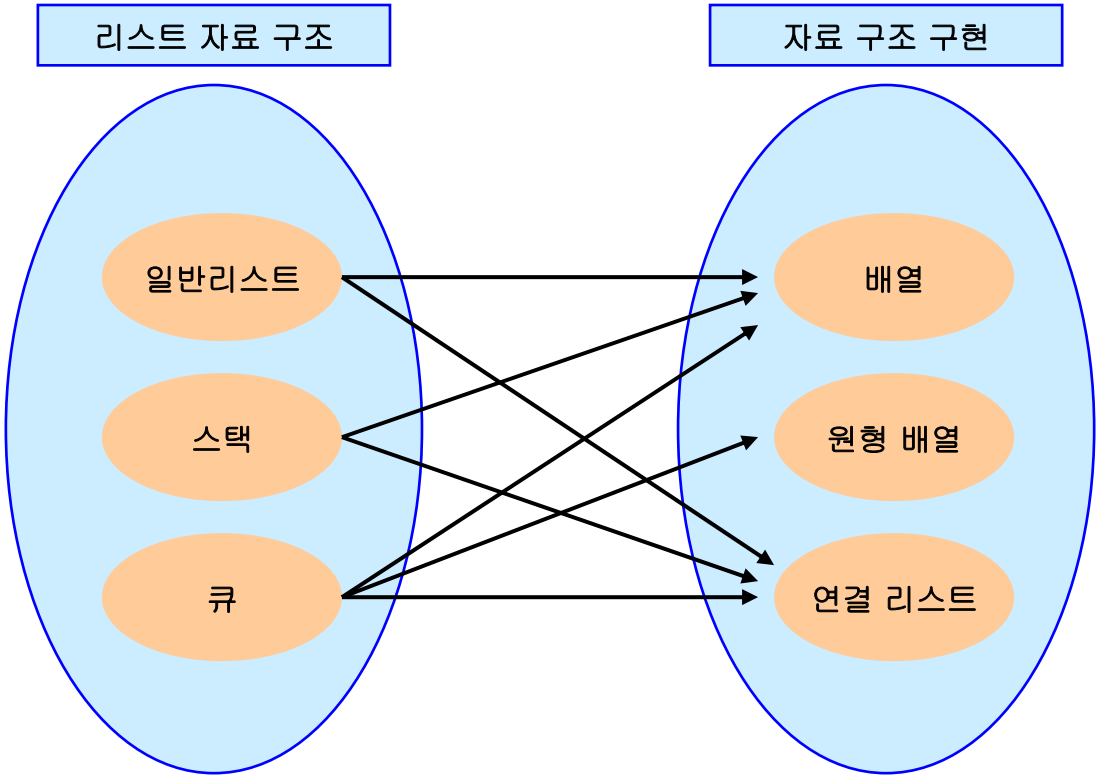
- **리스트 자료구조의 구현**
  - 배열은 리스트 자료구조를 구현하는 방법이다.
  - 연결리스트도 리스트를 구현하는 방법이다.

- **배열을 이용한 리스트 구현은 다음과 같은 장단점이 있다.**
  - 연속된 기억 장소
  - 데이터의 중간에 삽입, 삭제 시 데이터 이동이 필요하다.
  - 데이터의 크기가 수행 전(컴파일때) 결정된다.
  - 정적인 기억장소 할당(static storage allocation)


리스트 자료 구조
순서가 있는 자료의 총칭. 검색과 변경(삽입과 삭제) 연산이 필요한 자료구조
일반 리스트
삽입과 삭제가 임의의 장소에서 일어남.
스 택
삽입과 삭제가 한쪽 끝에서 일어남
큐
삽입은 rear에서 삭제는 front에서 일어남

## 리스트 자료구조를 구현하는 프로그램 자료구조의 종류

(설명) 왼쪽은 자료구조이고 오른쪽은 구현방법이다. 예를 들면 스택은 배열로 구현할 수도 있고 연결리스트로도 구현할 수 있다.



## 리스트를 구현하는 두 가지 방법의 비교

| | 배열 | 연결리스트 | 설명 |
|---|---|---|---|
| 구현방법 | 배열로 선언<br>예) int list[N]; | 연결리스트<br>예) list_ptr ptr; | |
| 기억장소 확보 시점 | 프로그램 수행 전(compile time)<br>(기억장소 N개로 시작) | 프로그램 수행 후 (run time)<br>(기억장소 1개로 시작) | 배열에서 기억장소가 수행 전에 할당되는 경우 기억 장소 크기가 고정되고, 사용여부에 관계없이 기억장소를 확보한다. |
| 기억장소 연속성 | 연속된 기억 장소 할당 | 기억장소 힙 (heap) 영역에서 임의로 할당 | 연결리스트에서는 기억장소가 여기저기 흩어져 있고 링크를 통하여 연결이 된다. |
| 삽입과 삭제 | 간단하다 | 복잡하다. | |

## Review

⦿ 포인터는 데이터의 효율성, 함수의 주소인자, 동적인 자료구조의 세가지 기능을 구현하기위한 방법이다. 포인터는 주소값을 저장하며 수행시간(동적, dynamic)에 리스트 데이터들을 주소값으로 연결하는 데 사용된다. 리스트를 배열을 이용하면 쉽게 표현할 수 있다.

⦿ 연결리스트는 포인터를 이용하여 리스트 데이터를 연결하는 방법이다.

⦿ 연결리스트는 스택과 큐를 구현하는 방법이 된다.

⦿ 다항식 문제는 배열을 이용하여 해결할 수 있다

⦿ 연결리스트의 생성과 반환을 위한 함수들에 대한 정확한 이해가 필요하다.
C 언어에서 주로 쓰는 함수는 malloc()과 free() 함수이다.