

## Linked List

- Ordered List 의 문제점: 삽입, 삭제시 많은 양의 자료이동 필요

예) (A, C, D) 에서 “insert B between A and B  
or “remove “C” from the list

- ⇒ sequential representation 에서 임의의 삽입과 삭제는 time-consuming.
- ⇒ Another difficulty is “waste of storage”
- ⇒ Solution: Linked List Representation

### . Linked List

- 순차적 표현 : 기억장소에서도 인접한 위치
  - 연결표현(LL) : 기억장소의 어느곳에 위치해도 무관함.  
단, List 의 원소들은 다음 원소 찾는 정보필요  
(주소, 위치번호)
  - Node(원소) : consists of two fields
    - data
    - pointer to next node: the pointer is called LINK
  - Singly Linked Lists:
    - . Ordered (순서)sequence of nodes
    - . Nodes do not reside in sequential locations
- Linked List 의 종류
- 1. Single Linked List (SLL)
  - 2. Circular Linked List (CLL)
  - 3. Double Linked List (DLL)
  - 4. Generalized Linked List (GLL)

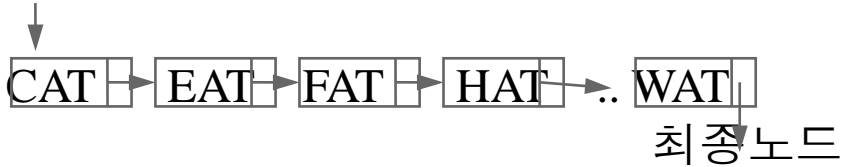
# 1. Single Linked List (SLL)

- Exercise (배열 이용한 Linked List) : 개념적 표현

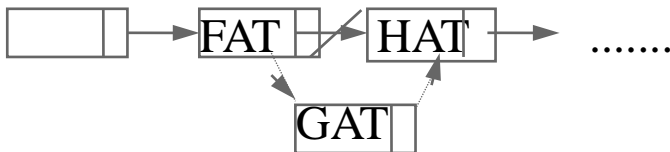
주소 Data Link

1	HAT	15
2		
3	CAT	4
4	EAT	8
5		
6	WAT	0
7		
8	FAT	1

시작노드



## ■ Insert “GAT”



1	HAT	15
	.....	
5	GAT	1
	.....	
9	FAT	1

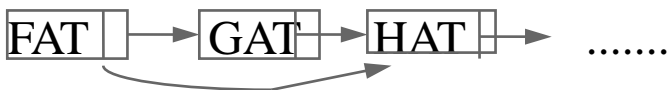
- 1) Get Currently unused Node, ex) Addr 5
- 2) Set data field 

5	GAT
---	-----
- 3) Set link field 

5	GAT	1
---	-----	---
- 4) FAT 의 link 

9	FAT	1
---	-----	---

## ■ Delete “GAT”



1	HAT	15
	.....	
5	GAT	1
	.....	
9	FAT	5

1	HAT	15
9	FAT	1

## ■ Node 정의

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

```
Class List{  
private  
    Node *head;  
public  
    List() {head = 0;}  
    void insert(int);  
    void isempty(int);  
    ....  
}
```

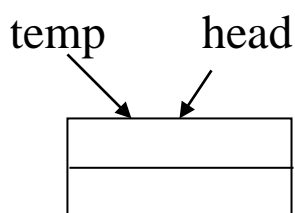
### ● head 선언 및 초기화

```
struct Node *head = NULL;
```

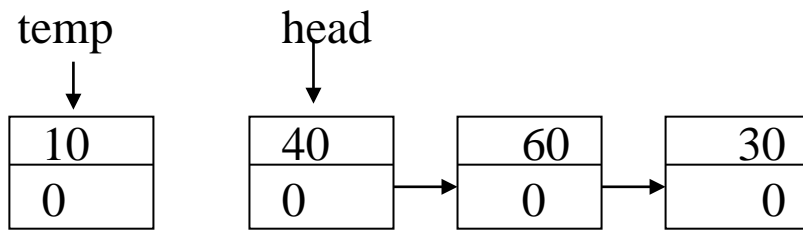
## ■ node 삽입

### 1) 맨앞에 삽입하기 (list 가 empty 일 경우)

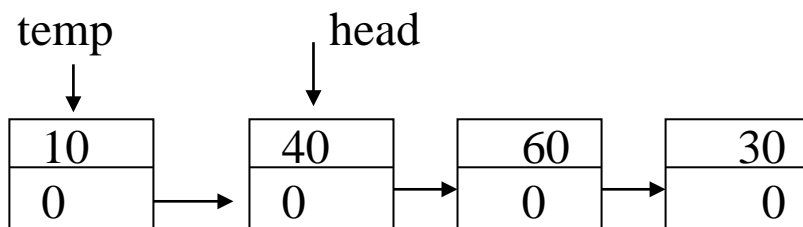
```
Node *temp = new Node; // 새노드 생성  
temp->data = num;  
temp->next = 0;  
head = temp; /* linked list 가 empty 인 경우에는  
head 가 temp 가리키는 node를 가리키게 한다.
```



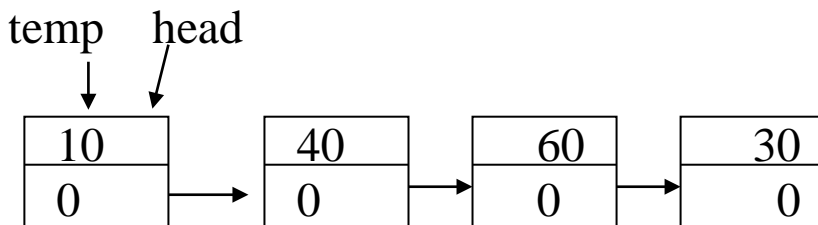
2) **head node** 가 0이 아닌경우 (즉, 여러 개의 노드가 있을 경우, 맨 앞에 삽입하기 )



- 우선 temp 와 head 연결 `temp->next = head;`



- head 가 list 의 맨앞을 가리키게 한다 `head = temp;`



3) **head node** 뒤에 **node** 삽입할 경우 (insert middle)

```
if (head->next == 0)
    head->next = temp; /* head node 하나밖에 없을 경우
else
{
    temp->next = head->next;
    head->next = temp;
}
```

#### 4) 맨 뒤에 node를 만들 경우 (insert last)

- 우선 삽입할 노드를 만들고 temp가 가리키게 한다.

```
Node *temp = new Node;  
temp->data = num;  
temp->next = 0;
```

- head 가 0인 경우 (list 가 empty 일 경우, head 가 temp 를 가리키게 한다)

```
head = temp;
```

- head가 0이 아닌경우 (list에 node 가 여러개 있을 경우)

```
p = head;  
while (p->next != 0)  
    p = p->next;  
p->next = temp;
```

#### ● 출력 하기

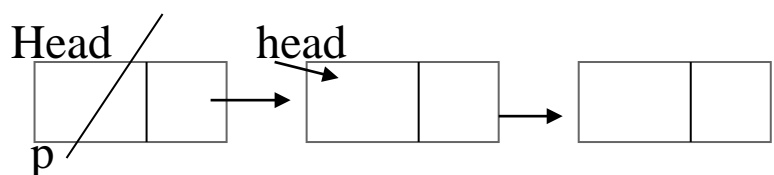
```
p = head;
```

```
While (p != 0) {  
    cout << p->data;  
    p = p-> next;  
}  
cout << endl;
```

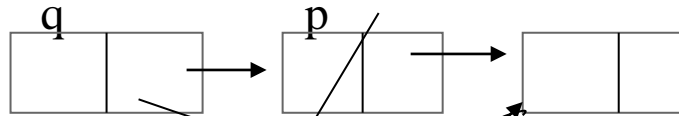
#### ■ node 삭제

##### 1) Delete from Front

```
If (head->data == num){  
    p = head;  
    head = head->next;  
    delete p;  
}
```



## 2) Delete from Middle

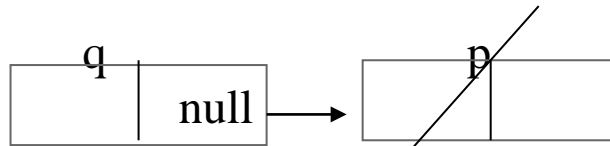


```

p = head; q= head;
while (p != NULL && p->data != num) {
    q=p;    p= p->next;
}
if (p != NULL) {
    q->next = p->next;
    delete p;
}
else
    cout << num << " is not in the list\n";

```

## 3) Delete from End



```

p = head;
q=head;

while ( !p->next= null) {
    q=p;    p = p->next;
}
q->next = null;
delete p;

```

### • Hint

- 우측으로 이동: current = current->next;
- 현재 노드를 Head로: current = Head;
- Traverse
 

```

p = head;
while (p != NULL) {
    /* cout<< p->data; */
    p = p->next;
}

```

## 1.1 Singly Linked List 알고리즘

### (1) Singly Linked List ADT

```
class Node {
    private:
        int data;
        Node *next;
    friend class List;
};

class List {
    private:
        Node *head;
    public:
        List () { head = 0;}
        void insertNode(int);
        void deleteNode(int);

        bool isEmpty();
        void traverseList();
};
```

### (2) List ADT 의 operations (member functions)

변수 및 함수선언부	설명
bool isEmpty()	연결리스트가 빈 것을 검사하는 함수
void insertNode(int num)	연결 리스트에 노드를 삽입하는 함수
void deleteNode(int num)	연결 리스트에 노드를 삭제하는 함수
void traverseList()	연결 리스트의 노드들 내용을 출력 함수
void searchList()	연결 리스트에서 데이터를 찾는 함수
~List()	연결 리스트의 각 노드들을 시스템에 반환하는 함수

### (3) isEmpty 함수 설명

기능: 현재 연결 리스트가 비어있는지의 여부를 검사  
반환값: head 가 NULL 이면 1 을 그렇지 않으면 0 을 반환

```
bool List::isEmpty()
{
    if (head == 0)    return TRUE;
    else    return FALSE;
}
```

### (4) insert 함수 (데이터값 크기에 따라 입력될 경우-오름차순)

```
void List::insertNode(int data)
{
    Node *temp = new Node(data);
    Node *p, *q;

    if (head == 0)    head = temp;
    else if (temp->data < head->data) {
        temp->next = head;
        head = temp;
    }
    else {
        p = head;
        while ((p != 0) && (p->data < temp->data)) {
            q = p;
            p = p->next;
        }
        if (p != 0) {
            temp->next = p;    q->next = temp;
        }
        else
            q->next = temp;
    }
}
```

- 첫 노드(head)가 만들어지는 경우 (head == NULL)
- head 노드 앞에 노드가 삽입될 경우 (temp->data < head->data)
- 연결리스트의 가운데에 노드가 삽입되는 경우



(5) delete 함수

- head 의 삭제, 또는 리스트 가운데 노드 삭제

```
void List::deleteNode(int num)
{
    Node *p, *q;

    if (head->data == num) {    /*head 삭제
        p = head;
        head = head->next;
        delete p;
    }
    else {
        p = head;
        while (p != 0 && p->data != num) {
            q = p;
            p = p->next;
        }
        if (p != 0) {
            q->next = p->next;
            delete p;
        }
        else
            cout << num << " is not in the list\n";
    }
}
```

(6) traverse 함수

```
void List::traverseList()
{
    Node *p;

    if (!isEmpty()) {
        p = head;
        while (p) {
            cout << p->data;
            p = p->next;
        }
        cout << endl;
    }
    else
        cout << "List is empty!\n";
}
```

(7) ~List() 함수 : ~List()는 소멸자

```
List::~~List()
{
    Node *p;

    while (head != 0) {
        p = head;
        head = head->next;
        delete p;
    }
}
```

## (8) search 함수

```
void List::searchList(int num)
{
    Node *p;

    if (head != 0) {
        p = head;
        while (p != 0 && p->data != num)
            p = p->next;

        if (p)
            cout << p->data << " is found." << endl;
        else
            cout << num << " is not in the list." << endl;
    }
    else
        cout << "List is empty\n";
}
```

## \* Single List 예제

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

class List {
private:
    Node *head;
public:
    List () { head = 0;}
    void insert(int);
    void append(int);

    bool isEmpty();
    void display();
};

void List::insert(int data)
{ Node *temp = new Node;

    temp->data = data;
    temp->next = 0;

    if (head != 0) {
        temp->next = head;
        head = temp;
    }
    else    head = temp;
}

void List::append(int data)
{
    Node *temp = new Node;

    temp->data = data;
    temp->next = 0;
```

```
    if (head == 0)
        head = temp;
    else {
        Node *ptr = head;
        while (ptr->next != 0)
            ptr = ptr->next;
        ptr->next = temp;
    }
}

bool List::isEmpty()
{
    if (head == 0)    return true;
    else              return false;
}

void List::display()
{ Node *ptr;

    ptr = head;
    while (ptr) {
        cout << ptr->data;
        ptr = ptr->next;
    }
    cout << endl;
}

void main()
{ List l1;

    l1.insert(40);
    l1.insert(30);
    l1.append(50);
    l1.append(80);

    l1.display();
}
```

**/\* output: 30 40 50 80**

## 1.2 Linked Stacks and Queues

### 1) implementation of a STACK

- Class 선언

```
class Node {  
    private:  
        int data;  
        Node *next;  
        Node(int value)  
            { data = value; next = 0; }  
        friend class linkedStack;  
};  
  
class linkedStack {  
    private:  
        Node *head;  
  
    public:  
        linkedStack () { head = 0; }  
        ~linkedStack() {};  
        void createStack();  
        void push(int);  
        int pop();  
        int isEmpty();  
        void displayStack();  
        void searchStack(int);  
};
```

- Stack Create 함수

```
void linkedStack::createStack()  
{ head = 0; }
```

- \* PUSH 함수

```
void linkedStack::push(int data)  
{  
    Node *temp = new Node(data);  
  
    if (head == 0)  
        head = temp;  
    else {  
        temp->next = head;  
        head = temp;  
    }  
}
```

- \* POP 함수

```
int linkedStack::pop()  
{  
    Node *p;    int num;  
  
    num = head->data;  
    p = head;  
    head = head->next;  
    delete p;  
    return num;    }
```

- STACK-EMPTY 함수

```
int linkedStack::isEmpty()  
{  
    if (head == 0)    return 1;  
    else    return 0;  
}
```

## \* DisplayStack 함수

```
void linkedStack::displayStack()
{
    Node *p;

    if (!isEmpty()) {
        p = head;
        while (p) {
            cout << p->data;
            p = p->next;
        }
        cout << endl;
    }
    else
        cout << "Stack is empty!\n";
}
```

## 2) Linked List Implementation of Queue

- Class 선언

```
class Node {  
    private:  
        int data;  
        Node *next;  
        Node(int value) {data = value; next = 0;}  
    friend class linkedQueue;  
};  
  
class linkedQueue {  
    private:  
        Node *front;  
        Node *rear;  
  
    public:  
        linkedQueue () {front = 0; rear = 0;}  
        ~linkedQueue() { };  
        void createQueue();  
        void enqueue(int);  
        int  dequeue();  
        int  isEmpty();  
  
        void displayQueue();  
        void searchQueue(int);  
};
```

\* Queue-empty 함수

```
int linkedQueue::isEmpty()
{
    if (front == 0)    return 1;
    else                return 0;
}
```

● create 함수

```
void linkedQueue::createQueue()
{
    front = 0;    rear = 0;
}
```

● Enqueue 함수

```
void linkedQueue::enqueue(int data)
{
    Node *temp = new Node(data);

    if (front == 0) { /* 큐가 empty 인 경우
        front = temp;
        rear = temp;
    }
    else {
        rear->next = temp;
        rear = temp;
    }
}
```



- dequeue 함수

```
int linkedQueue::dequeue()
{
    Node *p;    int  num;

    num = front->data;
    p = front;

    if (front == rear) { front = 0;    rear = 0; }
    else                front = front->next;

    delete p;
    return num;
}
```

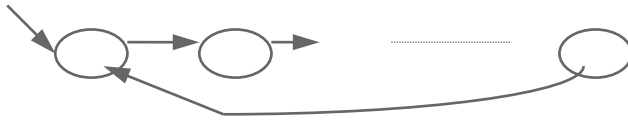
- Display-Queue 함수

```
void linkedQueue::displayQueue()
{
    Node *p;

    if (!isEmpty()) {
        p = front;
        while (p) {
            cout << p->data;
            p = p->next;
        }
        cout << endl;
    }
    else
        cout << "Queue is empty!\n";
}
```

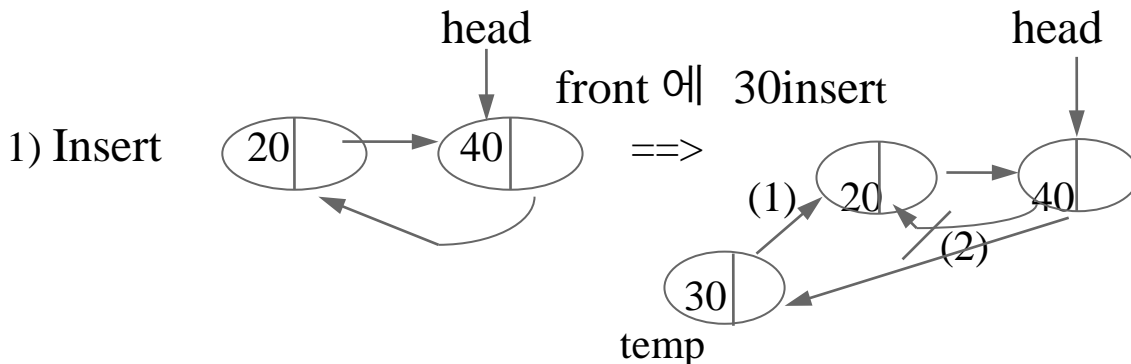
## 2. Circularly Linked List (CLL)

원형 연결 리스트



(Head Node 필요함, . No NIL . I/O 시 Buffer 에 이용)

### - 원형 연결 리스트 연산



```
void insert_front(list_pointer *head, list_pointer p)
```

```
/* head 다음에 노드를 삽입시 */
```

```
{
```

```
    if (IS_EMPTY(head)) {
```

```
        head = temp;           // 리스트가 공백일 경우,
```

```
        temp->link = temp;    } // head->link=head
```

```
    else {                     /* 리스트가 공백이 아닌 경우
```

```
        temp->link = head->link; (1)
```

```
        head->link = temp;      (2)
```

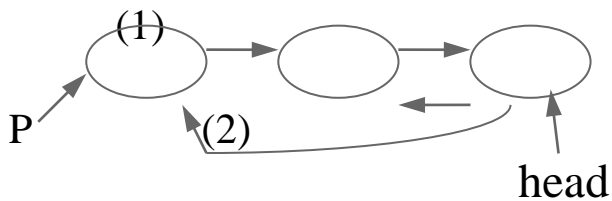
```
/*Head 변경시, 다음줄 추가 */
```

```
    //head = temp;
```

```
}
```

```
}
```

## 2) Delete



```
if (head == NULL)
    {list_empty()}
else{
    //head다음 노드 삭제시
    p = head->link;    (1)
    head->link = p->link  (2)
    delete p;  //delete p
}
```

## 3) length

---

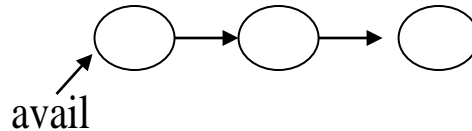
```
int length(list_pointer head)
{
    /* 원형 리스트의 길이를 계산한다. */
    list_pointer temp;
    int count = 0;

    if (head) { // 공백이 아닐경우
        temp = head;
        do {
            count++;
            temp = temp->link;
        } while (temp != head);
    }
    return count;
}
```

## ● CLL 의 응용

### Available space list(가용공간 리스트)의 관리

- . 사용하지 않는 공백리스트가 SLL로 구성되어 있다고 가정.
- . 필요시, 새노드 생성하지 않고 가용리스트에서 가져옴.

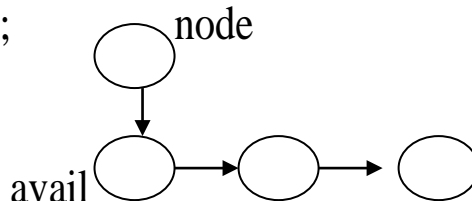


```
poly_pointer get_node( )    /* 사용할 노드를 제공 */
{
    poly_pointer node;

    if (avail) {
        node = avail;
        avail = avail->link;
    }
    else {          /* 사용할 노드가 없을때 새노드 생성 */
        new node;
    }
    return node;
}
```

---

```
void return_node(poly_pointer node) /* 가용 리스트에 노드를 반환 */
{
    node->link = avail;
    avail = node;
    /* node = 0 ; */
}
```



## ● 원형 리스트의 전체 제거

(가용리스트에 원형 리스트 한꺼번에 반환시 사용)

```
void cerase(poly_pointer ptr)
```

```
{          /* 원형 리스트 ptr을 제거 */
```

```
    poly_pointer temp;
```

```
    if (ptr) {          //공백리스트가 아닐경우
```

```
        temp = ptr->link;    (1)
```

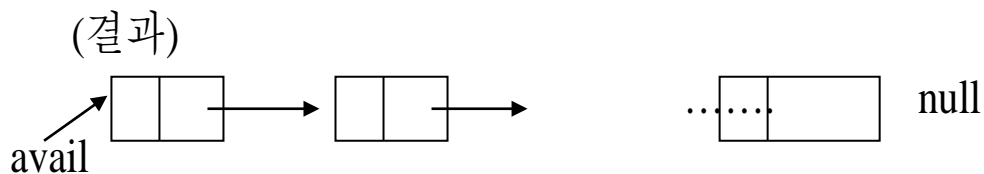
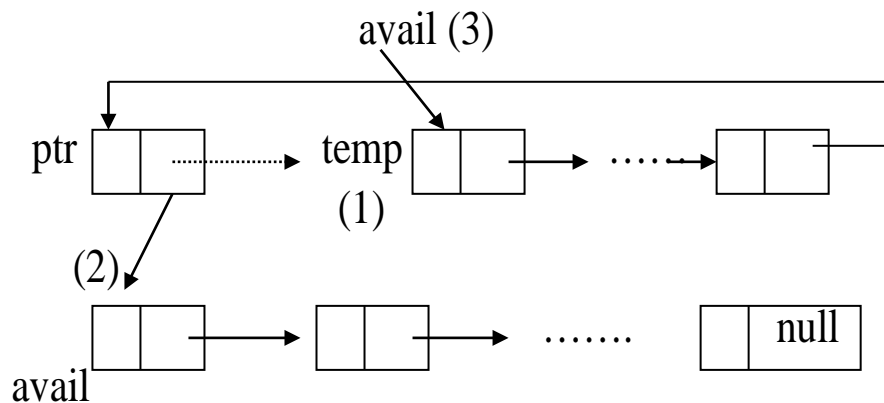
```
        ptr->link = avail;    (2)
```

```
        avail = temp;        (3)
```

```
        ptr = NULL;
```

```
    }
```

```
}
```



- 추가적인 리스트 연산

---

```
list_pointer invert(list_pointer head)
```

```
{    /* head가 가리키고 있는 리스트를 역순으로 만든다. */
    list_pointer middle, trail;
    middle = NULL;
    while (head) {
        trail = middle;
        middle = head;
        head = head->link;
        middle->link = trail;
    }
    return middle;
}
```

---

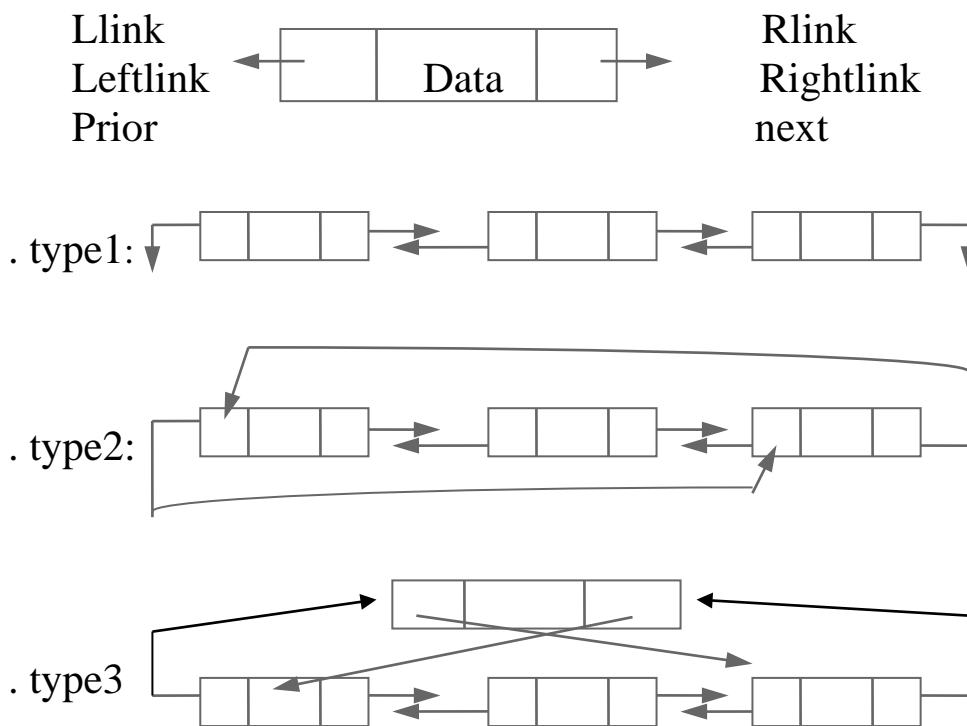
```
list_pointer concatenate (list_pointer ptr1,  list_pointer ptr2)
```

```
{ /* 리스트 ptr1 뒤에 리스트 ptr2가 접합된 새 리스트를 생성한다.
   ptr1이 가리키는 리스트는 영구히 바뀐다.*/
```

```
    list_pointer temp;
    if (IS_EMPTY(ptr1)) return ptr2;
    else {
        if (!IS_EMPTY(ptr2)) {
            for (temp = ptr1; temp->link; temp = temp->link)
                ;
            temp->link = ptr2;
        }
        return ptr1;
    }
}
```

### 3. Doubly Linked List

- SLL 의 단점: 특정노드의 이전노드를 찾기 위해서는, 처음부터 전체 list 검색해야 한다 =>  $O(n)$  time  
⇒ DLL 은 이 문제를 2개의 link 로 해결
- DLL 의 정의 (DLL may or may not be circular )



- 노드선언(Declaration)

Typedef int Type;

```
class Node{
    private:
        Type val;
        Node *next;
        Node *prev;
        Node (Type data)
            { val = data;    next = 0;    prev = 0;}
    friend class List;
};
```

```

class List {
    private:
        Node *head;
    public:
        List();
        ~List();
        void insertList(int, char[]);    //Node insert
        void deleteList(int);            //Node delete
        void forwardList();               // print from Head node
        void backwardList();             // Print from last node
        void searchList(int);            // search data
        int isEmpty();
};

```

- 이중 연결 리스트 특성

**ptr = ptr->prev->next = ptr->next->prev**

- 함수

### 1) isempty 함수

```

int List::isEmpty()
{
    return (head == 0);
}

```

### 2) 노드생성

```

List::List()
{
    head = 0;    current = 0;
}

```



### 3) 소멸자 함수

```
List::~~List() {  
    Node *p;  
  
    while (head != 0) {  
        p = head;    head = head->next;  
        delete p;  
    } }
```

### 4) insert first 함수

```
void List::insertFirst (Type data) {  
    Node *temp = new Node(data);  
  
    if (head == 0)    head = temp;  
    else {  
        temp->next = head;  
        head->prev = temp;  
        head = temp;  
    } }
```

### 5) insert last 함수

```
void List::insertLast(Type data) {  
    Node *temp = new Node(data);    Node *p;  
  
    if (head == 0)    head = temp;    // if empty list  
    else {  
        p = head;  
        while (p->next != 0)    p = p->next;    // move to the last  
        p->next = temp;  
        temp->prev = p;  
    } }
```

## 6) insert 함수(통합)

```
void List::insertList(Type data) // 숫자의 경우(오름차순)
{
    Node *temp = new Node(data, name);    Node *p, *q;

    if (head == 0)        // 첫노드일때
        head = temp;

    else if (temp->data < head->data) { //head node 앞에 삽입
        temp->next = head;
        head->prev = temp;
        head = temp;
    }

    else {                // 가운데 삽입
        p = head;
        q = head;
        while ((p != 0) && (p->data < temp->data)) { //이동
            q = p;
            p = p->next;
        }

        if (p != 0) { // 중간에 삽입
            temp->next = p;
            temp->prev = q;
            q->next = temp;
            p->prev = temp;
        }

        else { // temp 가 큰경우
            q->next = temp;
            temp->prev = q;
        }
    }
}
```

## 7) Delete 함수

```
void List::deleteList(int key)
{
    Node *p, *q;

    if (head == 0) {out<< "List is empty" << endl;}
    else {
        if (head->data == key) { // 삭제될 노드가 head 일 경우
            if (head->next != 0) { // head 가 only node 아닐 경우
                p = head; head = head->next;
                head->prev = 0; delete p;
            }
            else
                head=0; // head only node
        }
        else { // 가운데 노드가 삭제될 경우
            q = head; p = head;
            while (p != 0 && p->data != key) { //이동
                q = p; p = p->next;
            }
            if (p->next != 0 && p->data==key) {
                q->next = p->next;
                p->next->prev = q; delete p; }
            else if (p->next == 0 && p->data ==key) {
                q->next=0; delete p;
            }
            else
                cout << key << " is not in the list\n";
        }
    }
}
```

#### 8) forward 함수

```
void List::forwardList()
{
    if (!isEmpty()) {
        Node *p = head;
        cout << "----- Forward List -----\\n";
        while (p!= 0) {
            cout << p->data << p->name << endl;
            p = p->next;
        }
    }
    else
        cout << "List is empty!\\n";
}
```

#### 9) backward 함수

```
void List::backwardList()
{
    if (!isEmpty()) {
        Node *p = head;
        while (p->next != 0)
            p = p->next;
        cout << "----- Backward List -----\\n";
        while (p!= 0) {
            cout << p->data << p->name << endl;
            p = p->prev;
        }
    }
    else
        cout << "List is empty!\\n";
}
```

#### 10) search 함수

```
void List::searchList(int key)
{
    if (!isEmpty()) {
        Node *p = head;
        while (p != 0 && p->data != key)    p = p->next;

        if (p != 0)
            cout << p->data << " is in the list\n";
        else
            cout << key << " is not int the list\n";
    }
    else
        cout << "List is empty!\n";
}
```

#### 11) Locate Nth 함수

```
void List::locateCurrent(int Nth)  {
Node *p;    int pos = 1;

if (head == 0)    cout << "List is empty!" << endl;
else if (listLength()  >= Nth) {
    p = head;
    while (pos != Nth) { p = p->next;    pos++;    }
    current = p;
    cout << pos << " * ";    cout <<    current->val    << endl;
}
else
    cout << "No such a line" << endl;
}
```

- 기타 함수: listLength

- **4. Generalized List : 일반리스트**

- 선형 리스트  $A = (\alpha_1, \alpha_2, \dots, \alpha_i, \dots, \alpha_n)$ ,  $n \geq 0$ ,
- 선형리스트의 구성요소는 원자에 국한되므로,  $1 \leq i \leq n$  인  $i$  에 대하여,  $\alpha_i$  가  $\alpha_{i+1}$  보다 먼저 나옴
- 일반리스트는  $\alpha_i$  가 (원자, 리스트) 일수 있기 때문에, 다차원의 구조를 가질 수 있다.

[정의: 일반리스트 A 는 원자 또는 list 원소들의 유한순차  $\alpha_1, \dots, \alpha_n (n \geq 0)$  이다.

원소 ( $\alpha_i$  ( $1 \leq i \leq n$ ))가 리스트 일때 이를 A 의 sublist 라 한다]

ex)  $D = ()$  : NULL/empty list,  $n=0$

$$A=(a, (b,c)) \quad : \quad n=2, \alpha_1=a, \quad \alpha_2=(b,c),$$

$$\text{Head}(A)=a, \quad \text{Tail}(A)=(b,c)$$
$$B = (A, A, ()) : \quad n=3, \alpha_1=A, \alpha_2=A, \alpha_3=NULL, \\ \text{Head}(B)=A, \quad \text{Tail}(B)=(A,())$$

$C = (a, C)$  :  $n = 2$ ,  $C = (a, (a, (a, \dots)))$  무한리스트

Ex)

A=(4,6)      A: 


0	4	
---	---	--

 $\rightarrow$ 

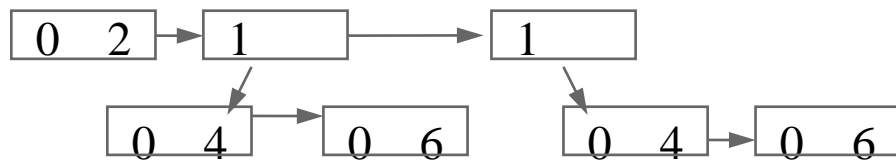
0	6	
---	---	--

 $\rightarrow$

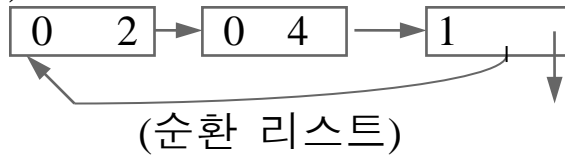
$B = ((4, 6), 8)$        $B :$

$C = (((4)), 6)$       C: 

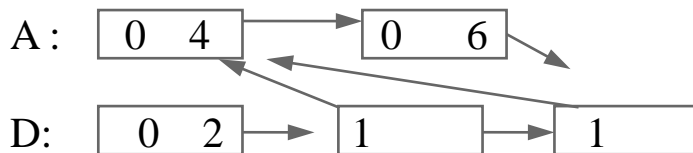
$$D=(2, A, A) \Rightarrow (2, (4,6), (4,6))$$



$$E = (2, 4, E)$$



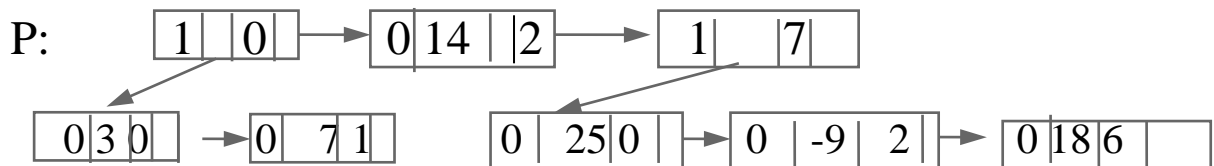
. Shared List



## ● 다항식의 G. List

Tag	Coef/dlink	exp	link
-----	------------	-----	------

$$\begin{aligned} \text{ex) } P(x,y) &= 3+7x+14y^2 + 25y^7 - 9x^2y^7 + 18x^6y^7 \\ &= (3+7x) + 14y^2 + (25 - 9x^2+ 18x^6)y^7 \end{aligned}$$



$$\text{ex) } P(x,y,z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^3y^4z + 6x^3y^4z + 2yz$$

⇒ 순차적으로 표현불가능,

⇒ G-list 에서는

$$((x^{10}+2x^8)y^3 + 3x^8y^2) z^2 + ((x^4+6x^3)y^4 + 2y)z$$

$$\Rightarrow Cz^2 + DZ$$

$$\Rightarrow C(x,y) = Ey^3 + Fy^2 \dots$$