Linked List

2015 2학기

kkman@sangji.ac.kr

SANGJI University

1. 리스트 개념

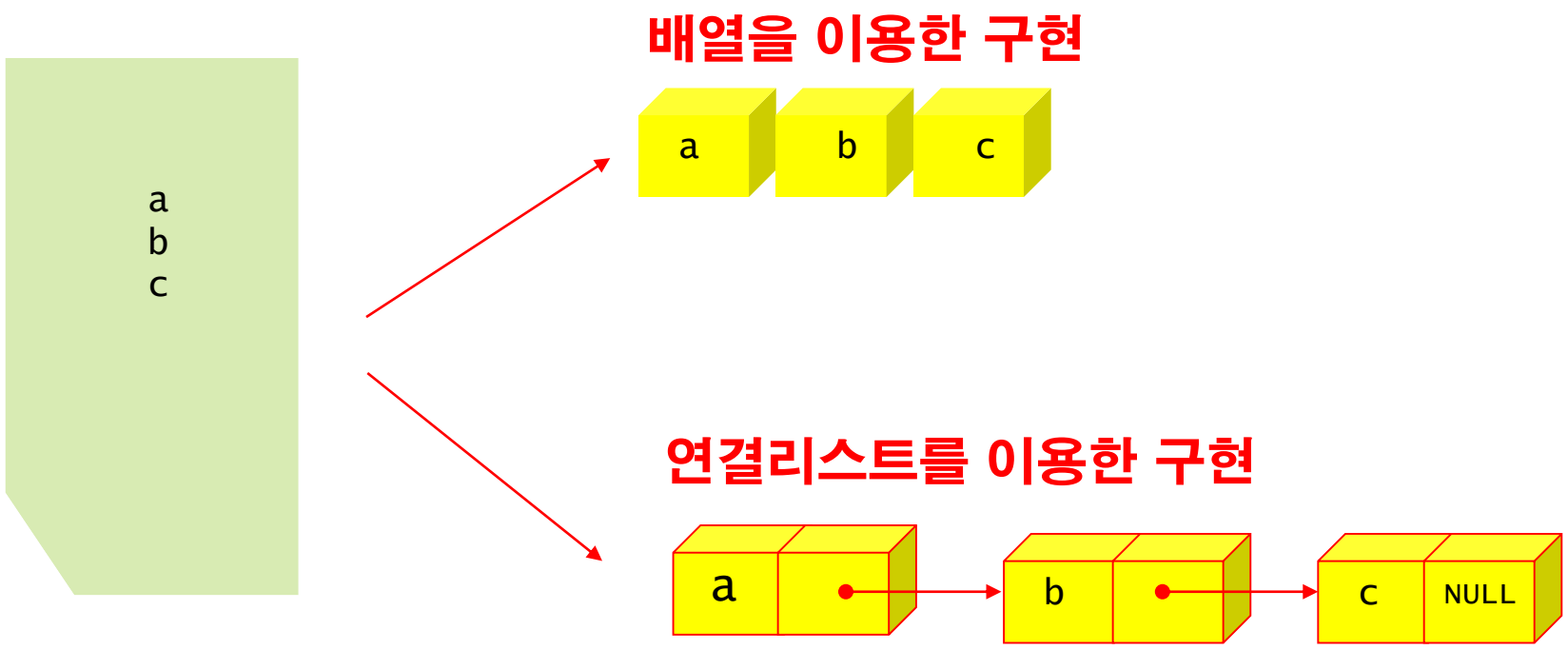
- **리스트(list)**
  - 고정된 길이의 자료(원소)들을 **순차적**으로 나열해 놓은 집합을 가르키는 자료구조의 **추상적**인 개념
  - 순서를 가진 항목들의 모임
    - 집합(set) : 항목간의 순서의 개념이 없음
  - 리스트의 예
    - 요일: (일요일, 월요일, …, 토요일)
    - 한글 자음의 모임: (ㄱ,ㄴ,…,ㅎ)
    - 핸드폰의 문자 메시지 리스트

- 리스트 연산, 검색+삽입+삭제
  - 새로운 항목을 리스트의 처음, 중간, 끝에 추가.
  - 기존의 항목을 리스트의 임의의 위치에서 삭제.
  - 모든 항목을 삭제.
  - 기존 항목을 대치(replace).
  - 리스트가 특정 항목을 가지고 있는지를 검색(search).
  - 리스트의 특정위치의 항목을 반환.
  - 리스트안의 항목의 개수를 센다(count).
  - 리스트가 비었는지(empty), 꽉 찼는지(full)를 체크.
  - 리스트안의 모든 항목을 표시.

리스트 구현 방법

- **배열(array)을 이용하는 방법**
  - 구현이 간단
  - 삽입, 삭제 동작에 따른 원소 이동.
  - 항목의 개수 제한, 정적 기억 공간 할당.
    - 메모리 낭비, 오버플로우

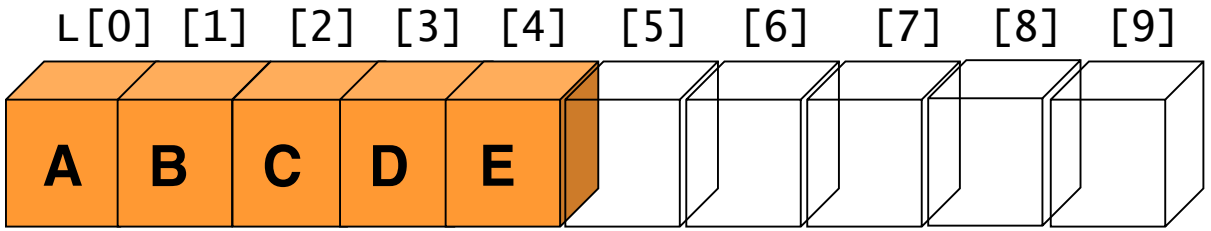
- **연결 리스트(linked list)를 이용하는 방법**
  - 구현이 복잡
  - 삽입, 삭제가 효율적
  - 크기가 제한되지 않음


a
b
c
배열을 이용한 구현
a
b
c
연결리스트를 이용한 구현
a
b
c
NULL

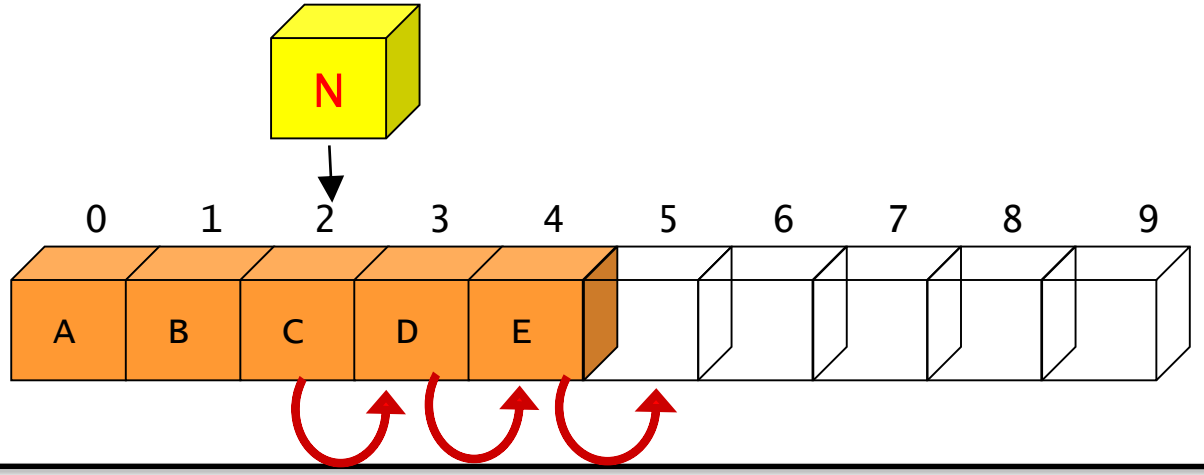
2. 배열로 구현한 리스트

- **1차원 배열**에 항목들을 **순서대로** 저장

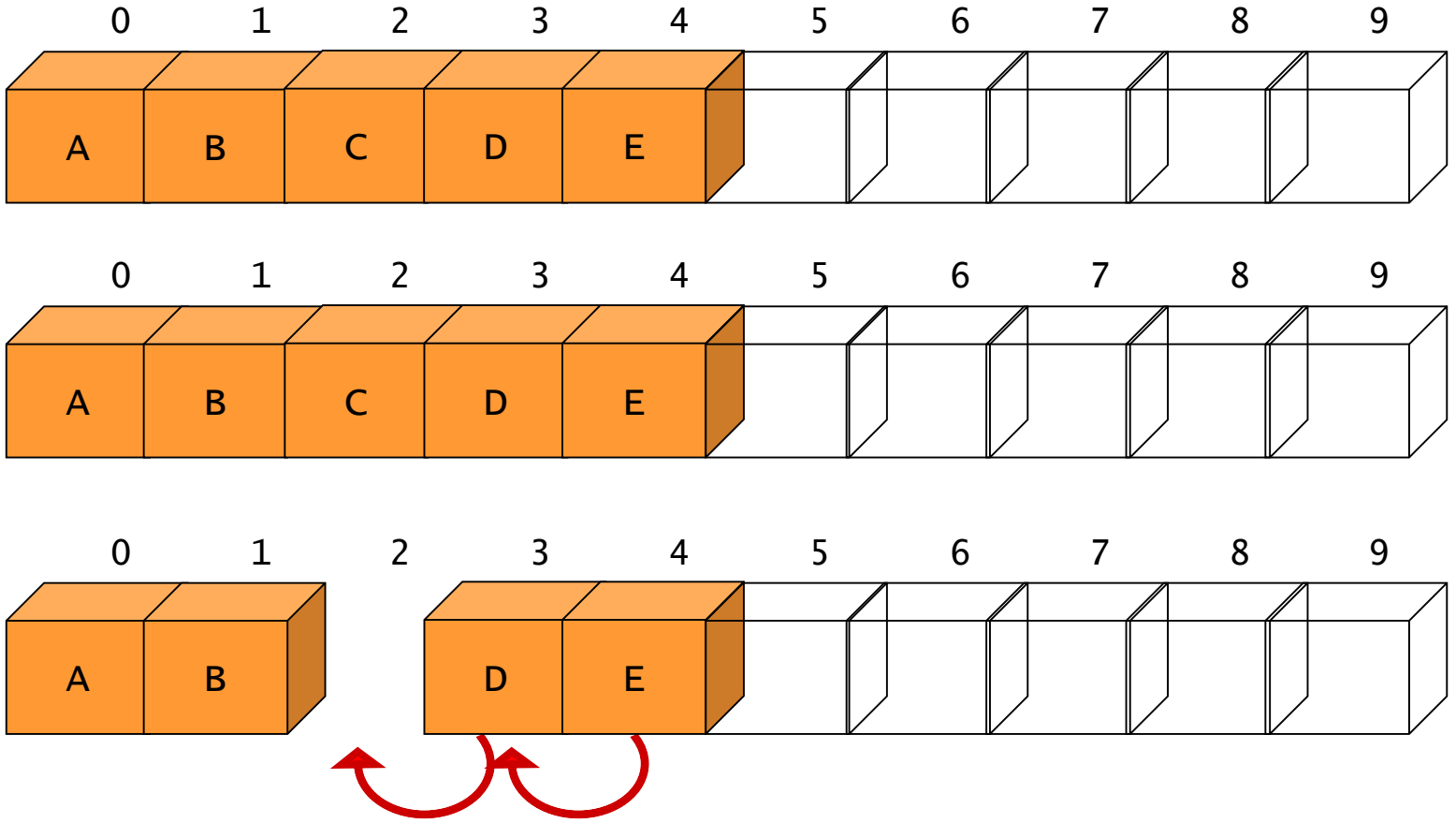
  L=(A, B, C, D, E)

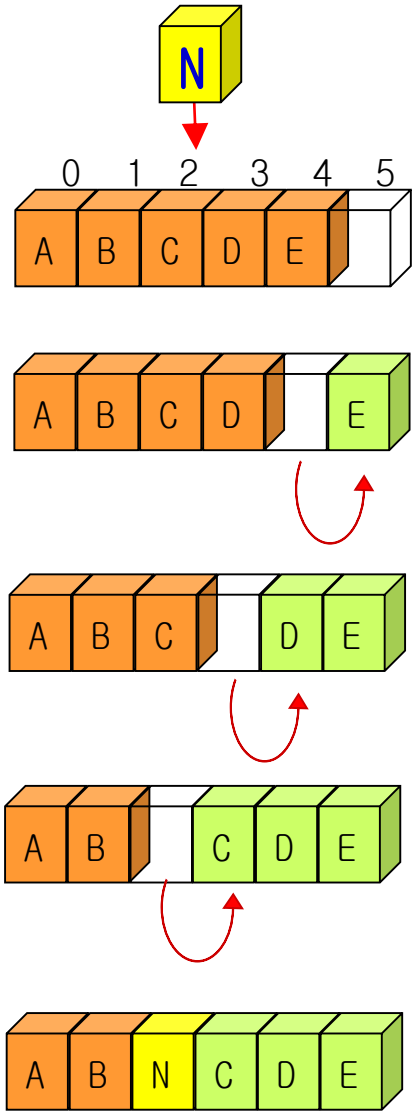


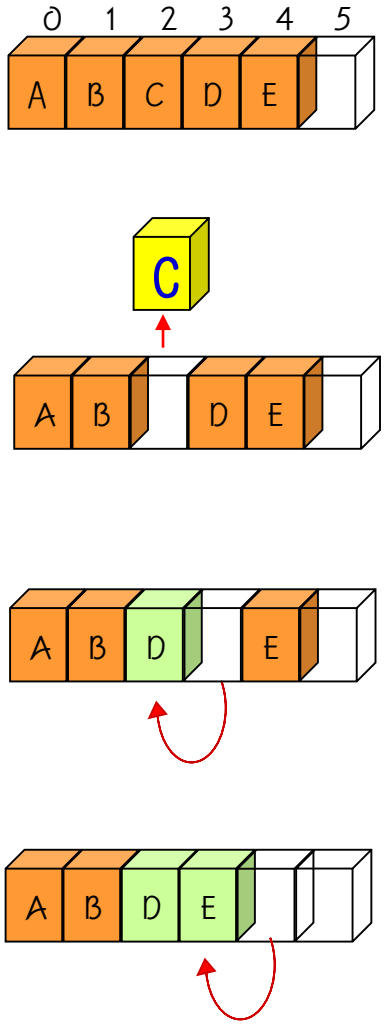
- **삽입**연산 : 삽입 위치 다음의 항목들을 **이동**하여야 함.



- **삭제 연산 : 삭제위치 다음의 항목들을 이동하여야 함**



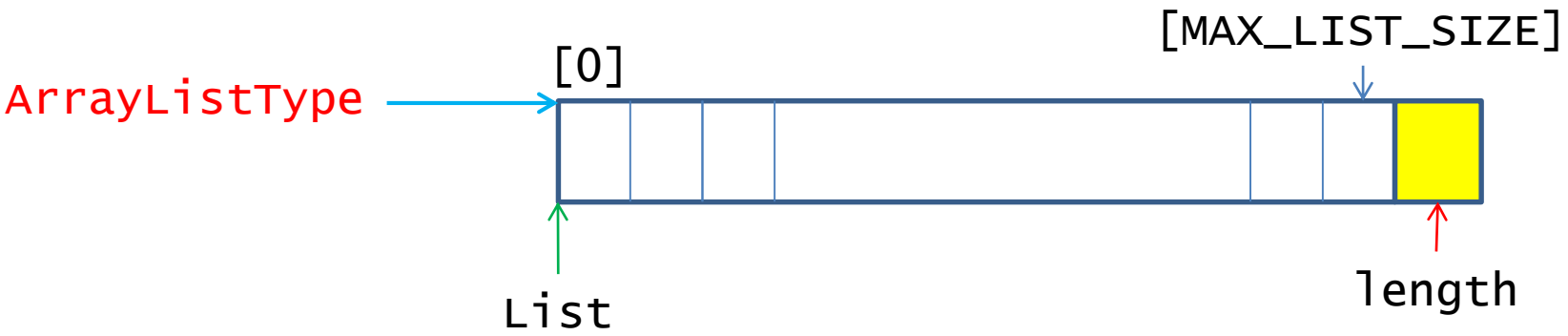

N
0 1 2 3 4 5
A B C D E
A B C D E
A B C D E
A B C D E
A B N C D E


0 1 2 3 4 5
A B C D E
C
A B D E
A B D E
A B D E

배열을 이용한 리스트 구현

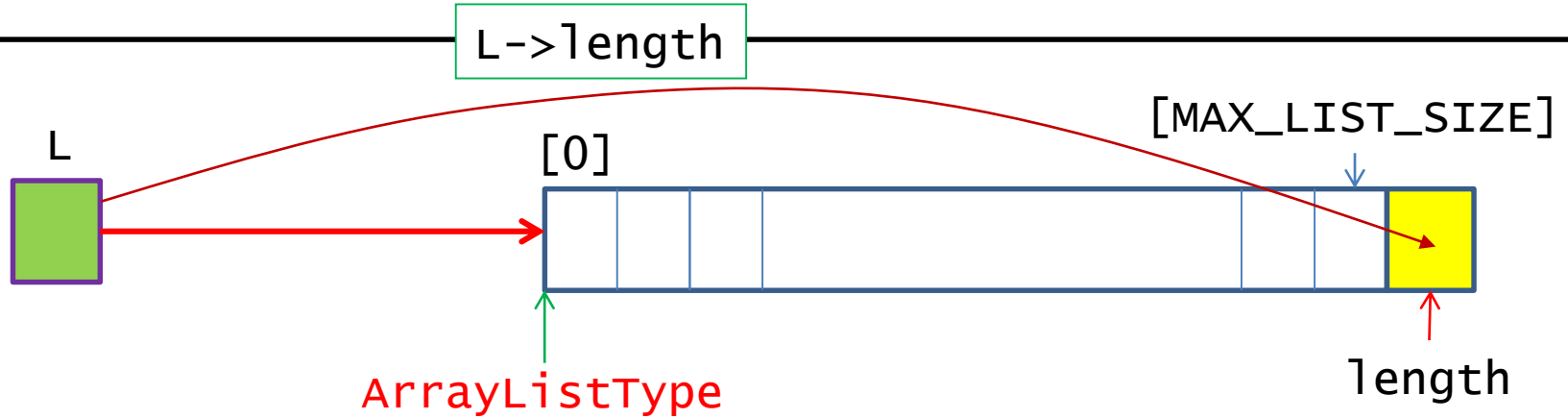
- list 1차원 배열에 항목들을 차례대로 저장
- length에 항목의 개수 저장

```
typedef struct {
        int list[MAX_LIST_SIZE];      // 배열 정의
        int length;                   //배열에 저장된 항목들의 개수
} ArrayListType;
```



- **is_empty(), is_full() 연산의 구현**

```
// 리스트가 비어 있으면 1을 반환, 그렇지 않으면 0을 반환
int is_empty(ArrayListType *L)
{
        return L->length == 0;
}
// 리스트가 가득 차 있으면 1을 반환, 그렇지 않으면 1을 반환
int is_full(ArrayListType *L)
{
        return L->length == MAX_LIST_SIZE;
}
```



- **삽입 연산 : add() 함수**
  - 배열이 **포화상태**인지를 검사, **삽입위치가 적합한 범위**에 있는지를 검사.
  - 삽입 위치 다음에 있는 자료들을 **한칸씩 뒤로 이동**.

```
// position: 삽입하고자 하는 위치, // item: 삽입하고자 하는 자료

void add(ArrayListType *L, int position, element item) {
  if( !is_full(L) && (position >= 0) &&
        (position <= L->length) ){
    int i;
    for(i=(L->length-1); i>=position;i--)
       L->list[i+1] = L->list[i];
    L->list[position] = item;
    L->length++;
  }
}
```

- **삭제 연산 : delete() 함수**
  - 삭제 위치 검사.
  - 삭제위치부터 맨끝까지의 자료를 한칸씩 앞으로 이동.

```
// position: 삭제하고자 하는 위치
// 반환값: 삭제되는 자료
element delete(ArrayListType *L, int position)
{
        int i;
        element item;

        if( position < 0 || position >= L->length )
                error("위치 오류");
        item = L->list[position];
        for(i=position; i<(L->length-1);i++)
                L->list[i] = L->list[i+1];
        L->length--;
        return item;
}
```

3. 연결 리스트를 이용한 리스트 구현

- 연결 리스트(linked list) 란 ?
  - 일정한 순서를 가지는 자료 요소들을 표현하는 자료 구조의 한 방법
  - 자료 요소들을 통합하여 관리함으로써 정보의 축적과 탐색을 효율적으로 실현하기 위해 사용되는 리스트 구조

- 연결 리스트 표현
  - 노드(NODE) 구성
    - 데이터(data): 리스트의 원소, 즉 데이타 값을 저장하는 곳
    - 링크(link): 다른 노드의 주소 값을 저장하는 장소(포인터)

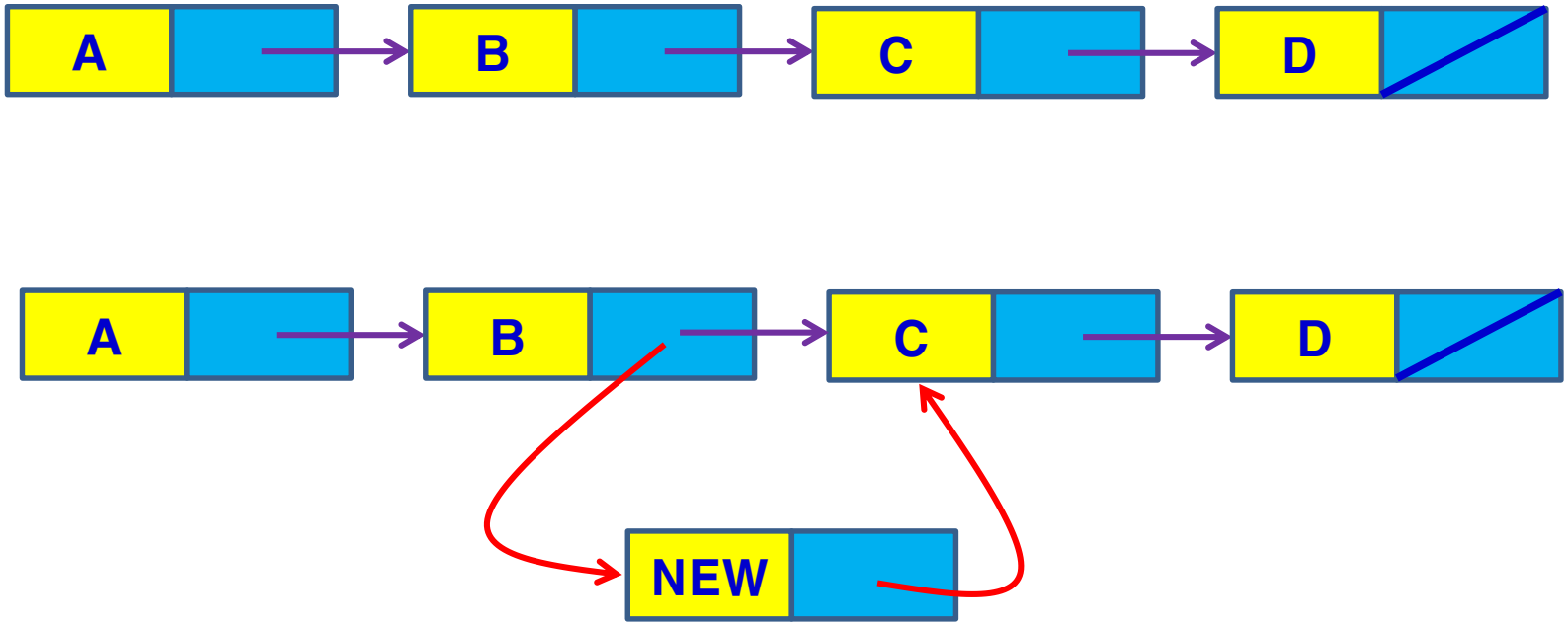
노드(node)

연결 리스트 개념

- 연결 리스트의 특징
  - 리스트의 항목들을 노드(node)라고 하는 곳에 분산하여 저장
  - 다음 항목을 가리키는 주소도 같이 저장
  - 노드 (node) : <데이터, 링크> 쌍
    - 데이타 필드 – 리스트의 원소, 즉 데이타 값을 저장하는 곳
    - 링크 필드 – 다른 노드의 주소값을 저장하는 장소(포인터)
  - 메모리안에서의 노드의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음

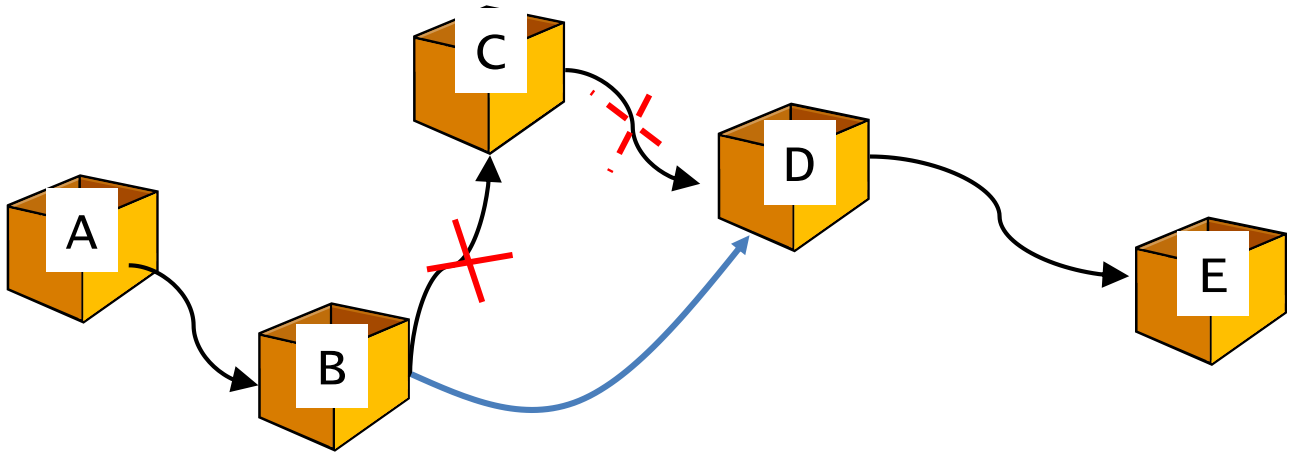
- **연결 리스트 장점**
  - 삽입, 삭제가 보다 용이.
  - 연속된 메모리 공간이 필요 없음.
  - 크기 제한이 없음.

삽입 연산

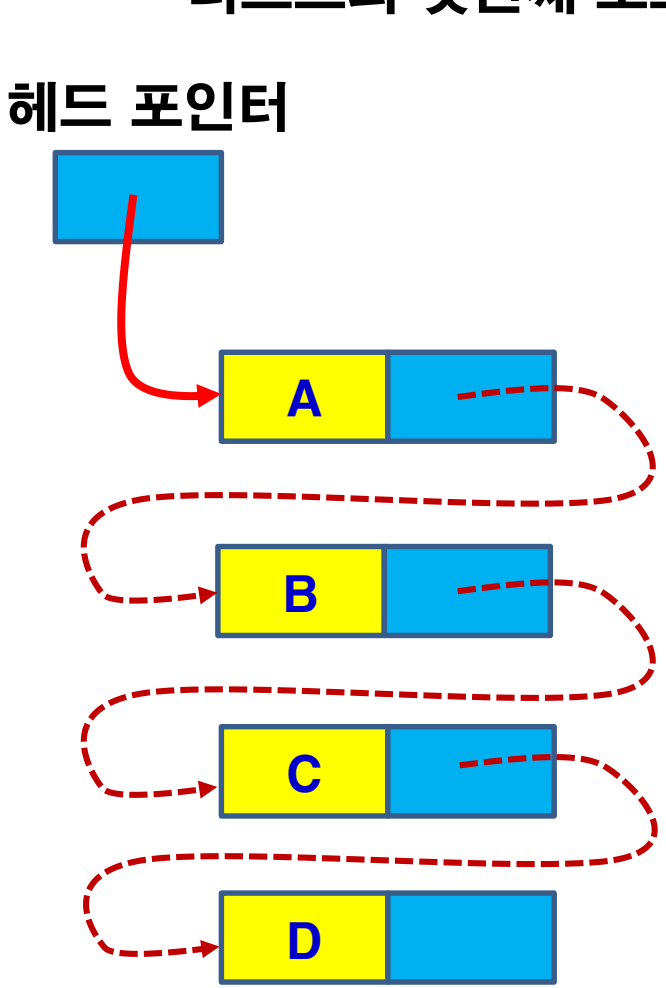


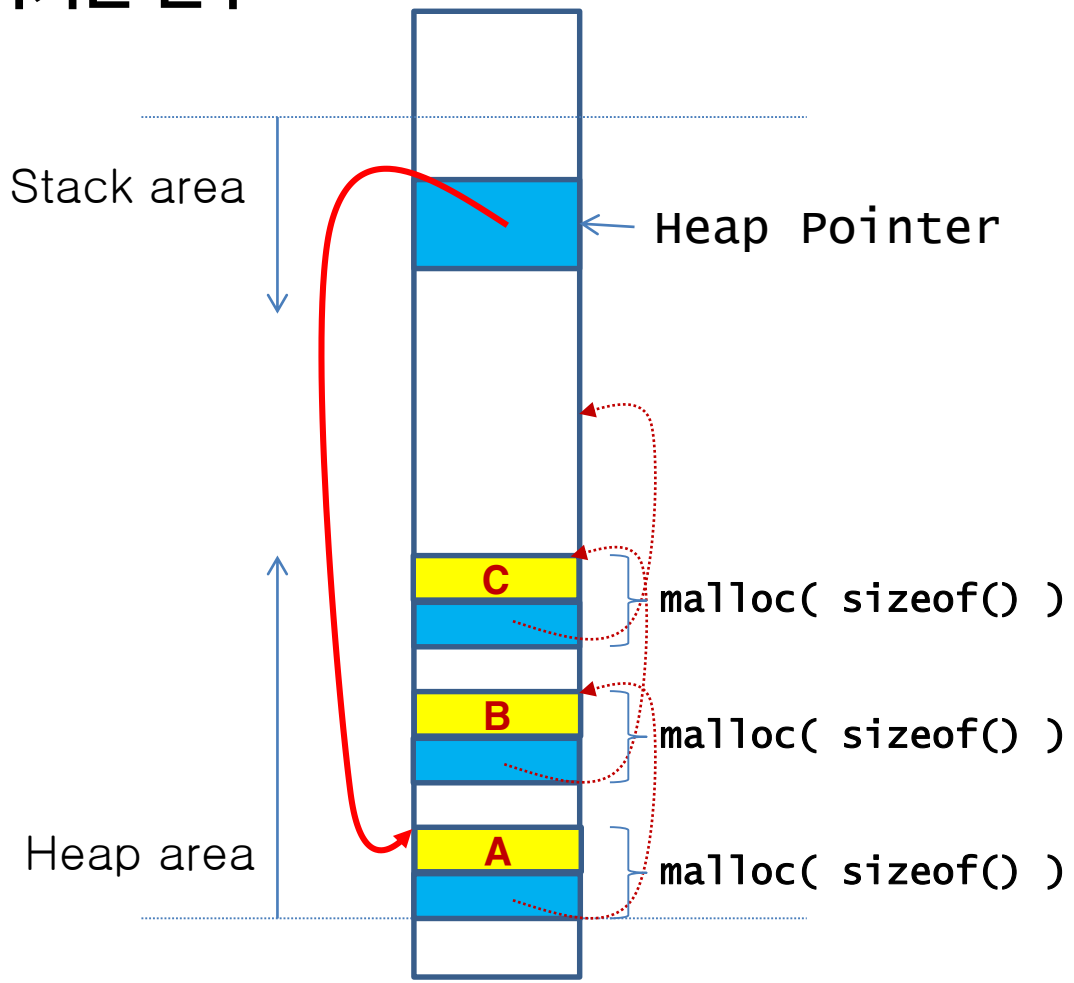
- **연결 리스트 단점**
  - 구현이 어렵다.
  - 오류가 발생하기 쉽다

삭제 연산

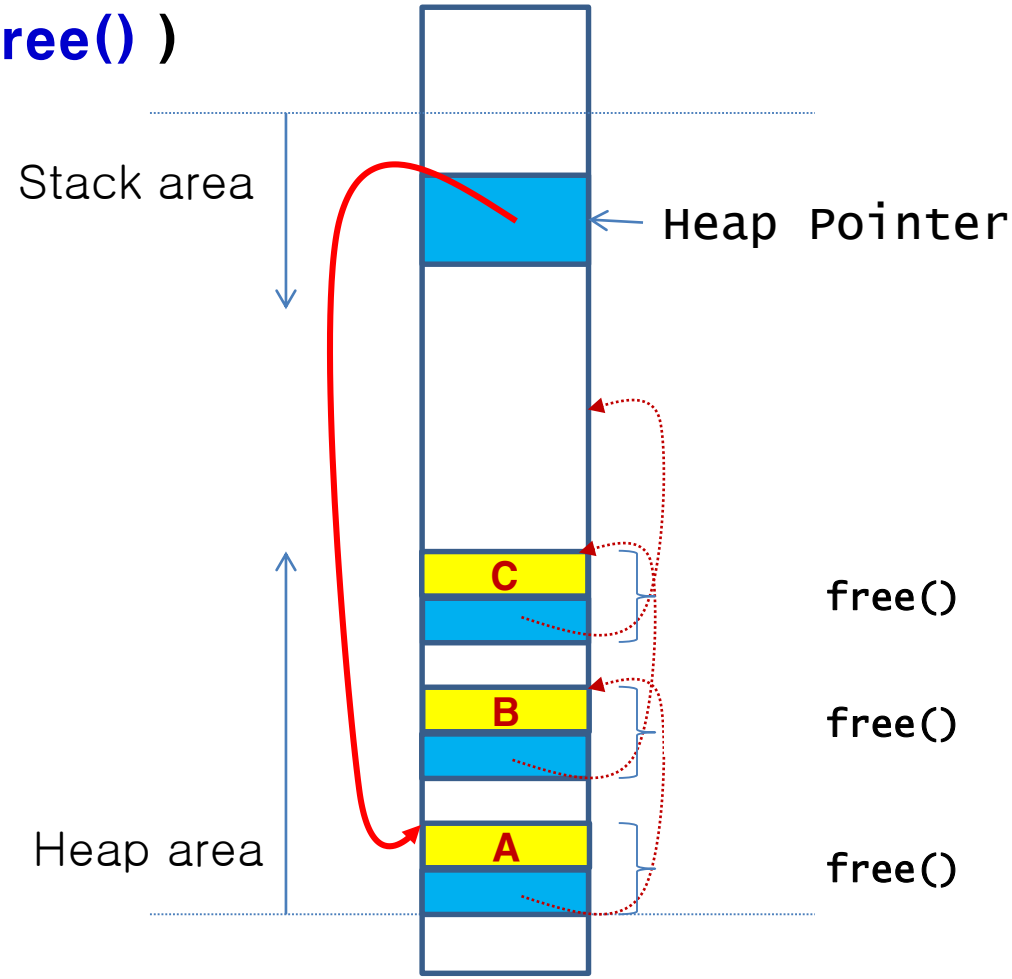


- **헤드 포인터(head pointer)**
  - 리스트의 첫번째 노드를 가리키는 변수



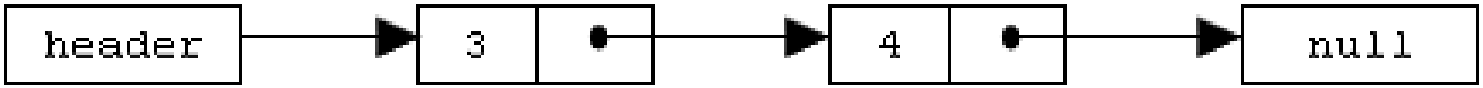


- **노드 생성과 반환**
  - 필요할 때마다 동적 메모리(heap) 할당 요구(malloc()).
  - 사용을 마친 노드 반환(free())

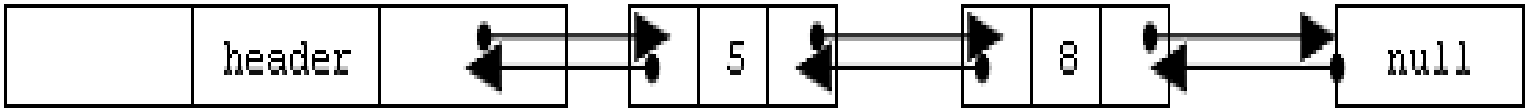


연결 리스트 종류

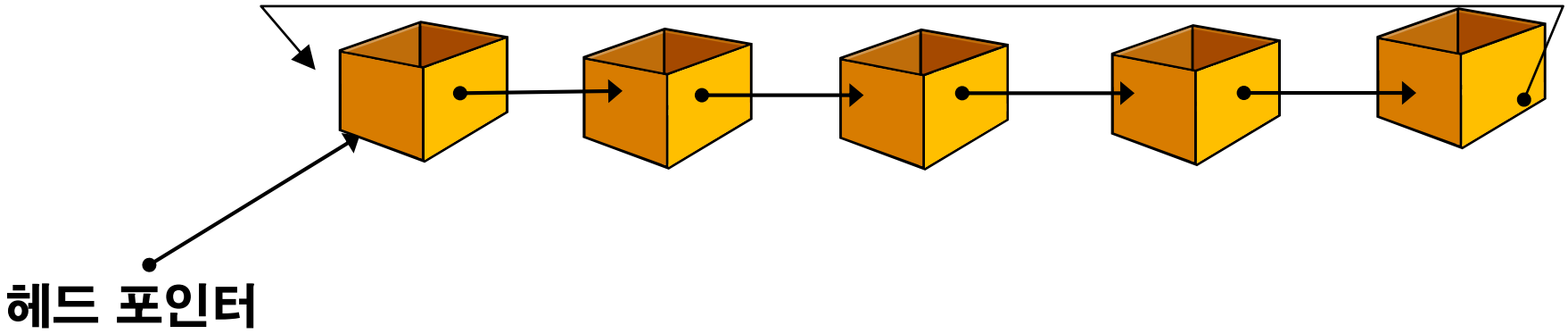
- **단순 연결 리스트(Simple Linked-List)**
  - 연결 리스트의 가장 단순한 형태
  - 단방향성
  - 마지막 노드의 링크
    - null 값을 가리킬 경우는 리스트의 끝을 나타냄.



- **이중 연결 리스트(doubly linked list)**
  - **양 방향으로 노드들을 탐색할 수 있음**
  - **각 노드들은 두개의 링크를 갖음**
    - **이전노드**
    - **다음 노드**



- **원형 연결 리스트(circular linked list)**
  - 마지막 노드의 링크가 첫번째 노드를 가리키는 리스트
  - 한 노드에서 다른 모든 노드로의 접근이 가능



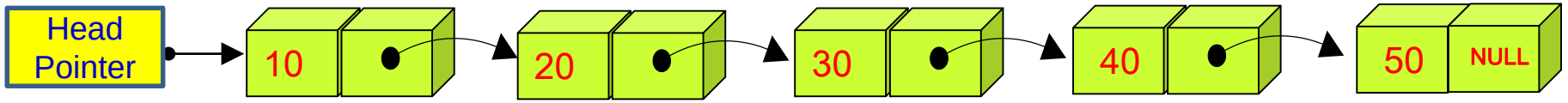
배열과 연결 리스트 구현 차이

- **연결 리스트 장점**
  - 삽입/삭제 연산이 위치에 관계없이 빠르게 수행
  - 무한개의 자료 저장
    - 배열 : 고정된 크기

- **연결 리스트 단점**
  - 순차 접근, 불연속 단위로 저장
    - 노드 접근에 긴 지연 시간(delay time)
  - 참조의 지역성(locality of reference)

4. 단순 연결 리스트

- 단순 연결 리스트(Simple Linked-List)
  - 연결 리스트의 가장 단순한 형태
  - 각 노드들은 오직 하나의 링크를 갖기 때문에 -> 단방향성.
  - 마지막 노드의 링크
    - null 값을 가리킬 경우는 리스트의 끝을 나타냄.
  - 헤더
    - null 값을 가리킬 경우는 빈 리스트를 나타냄
    - 값을 갖지 않는다.



단순 연결 리스트 구현(in C)

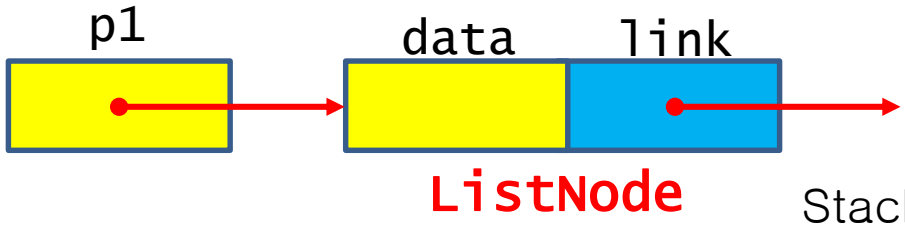
- 노드 생성
  - 데이터 필드 : 구조체 정의
  - 링크 필드 : 포인터

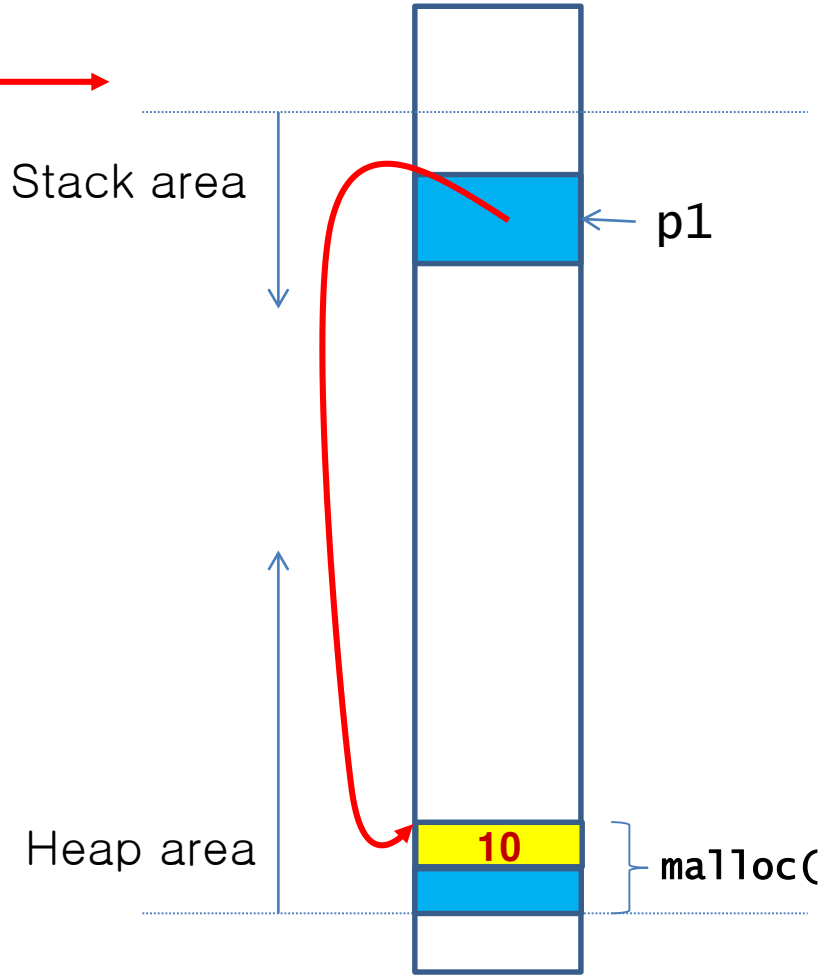
```
struct ListNode {
        int data;
        struct ListNode *link;
} ListNode;
```

  - 노드의 생성 : 동적 메모리 생성 라이브러리 malloc 함수 이용

```
ListNode *p1;
p1 = (ListNode *)malloc(sizeof(ListNode));
```

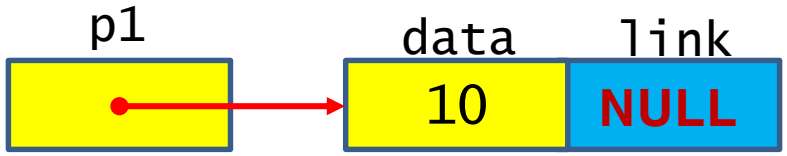
```
ListNode *p1;
p1 = (ListNode *)malloc(sizeof(ListNode));
```





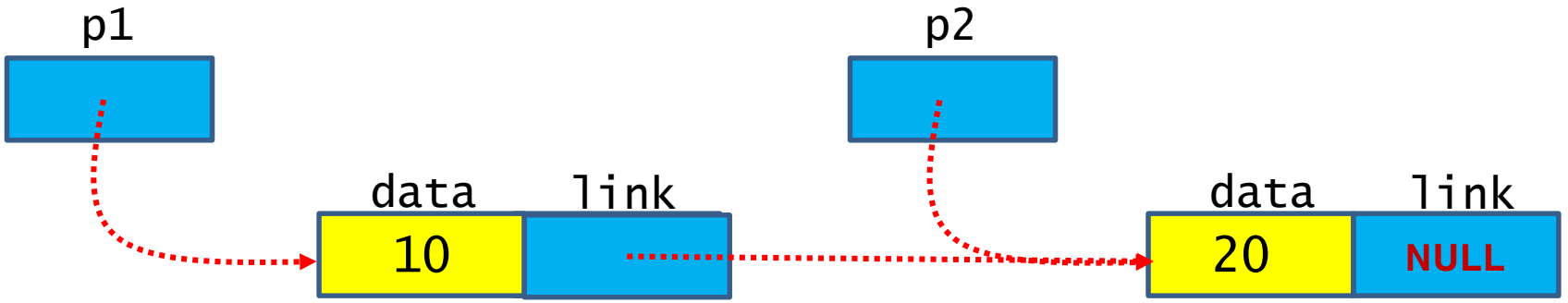
□ 데이터 필드와 링크 필드 설정

```
p1->data = 10;
p1->link = NULL;
```

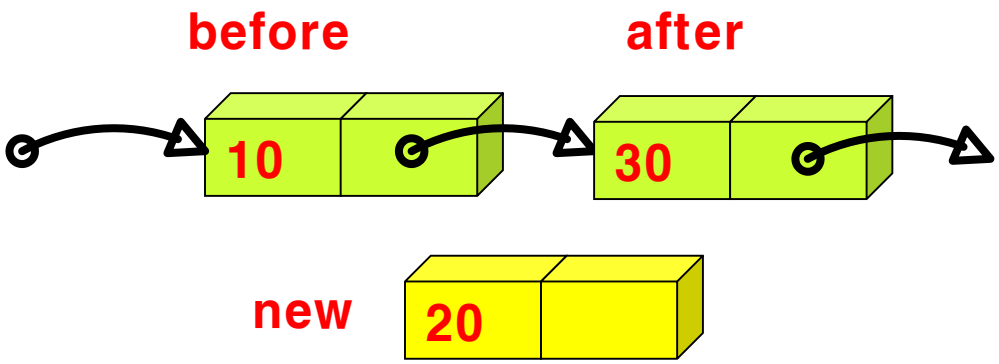


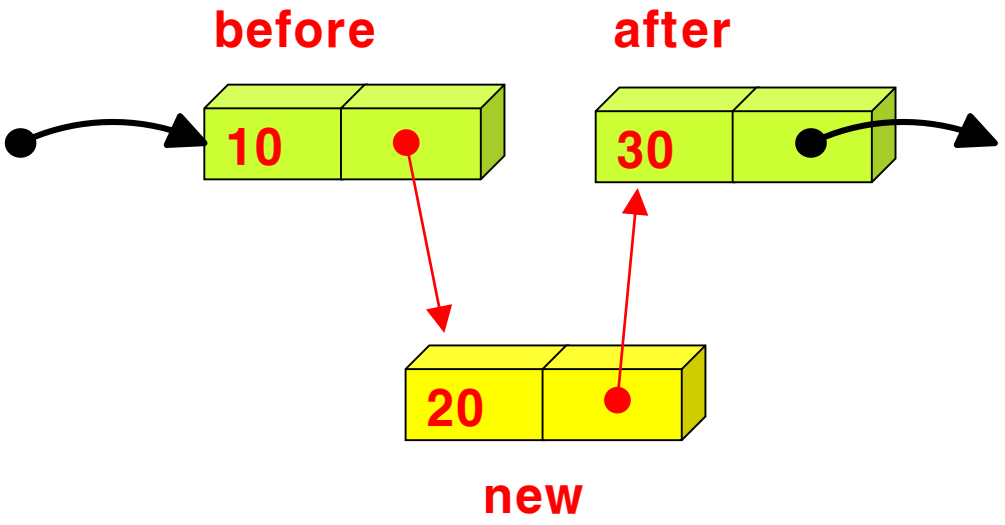
- **두번째 노드 생성과 첫번째 노드 연결.**

```
ListNode *p2;
p2 = (ListNode *)malloc(sizeof(ListNode));

p2->data = 20;
p2->link = NULL;
p1->link = p2;
```



단순 연결 리스트의 삽입 연산



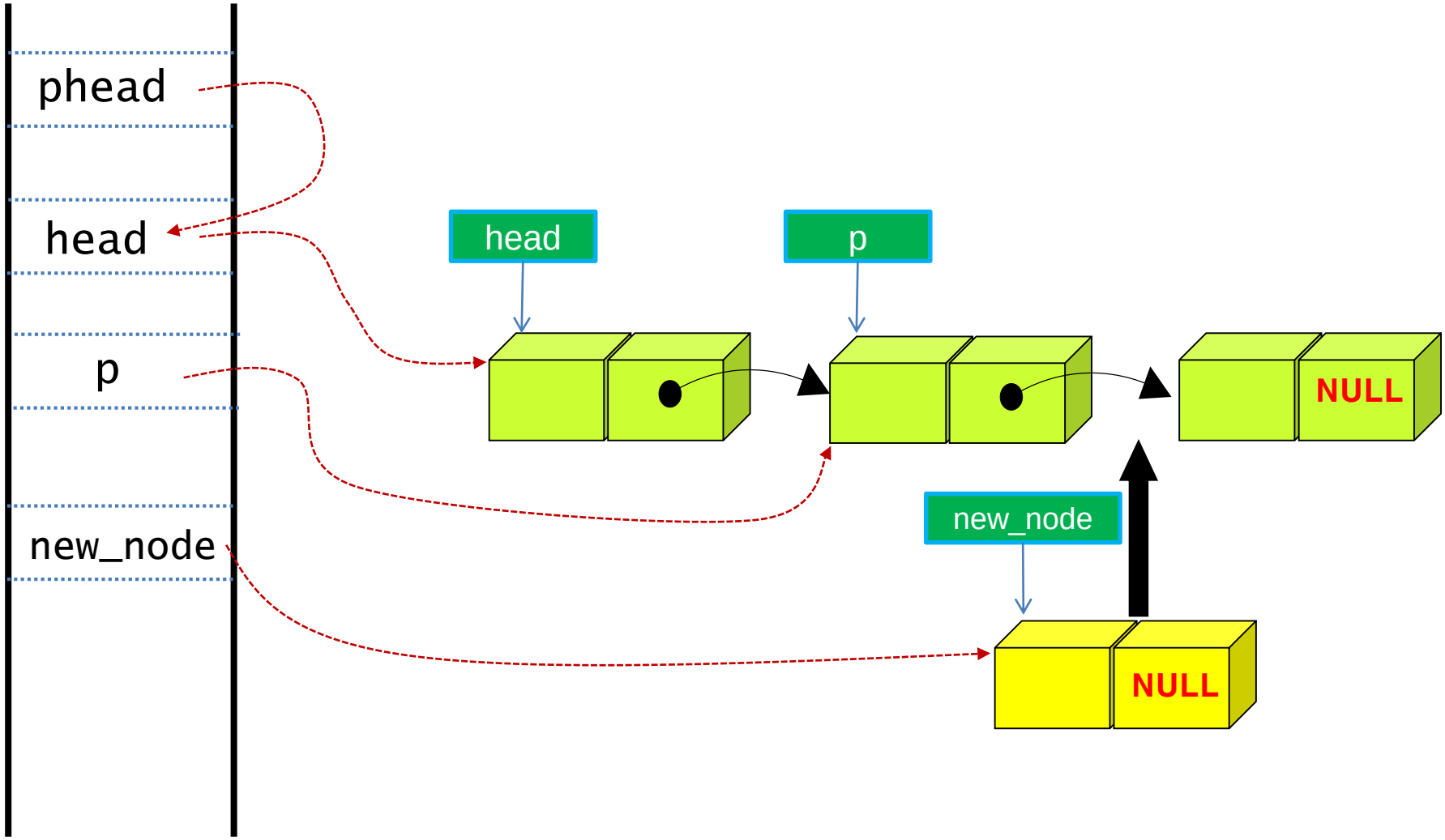


- **삽입 함수의 프로토타입**

```
void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)
```

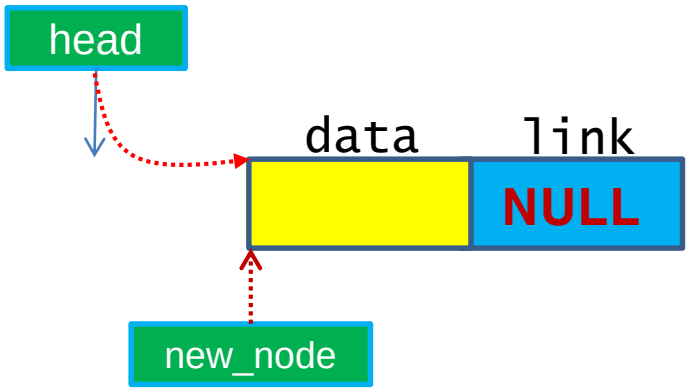
  - **phead** : 헤드 포인터에 대한 포인터
  - **p** : 삽입될 위치의 선행노드를 가리키는 포인터, **이 노드 다음에 삽입**.
  - **new_node** : 새로운 노드를 가리키는 포인터

- **삽입의 3가지 경우**
  - **head가 NULL인 경우: 공백 리스트에 삽입**
  - **p가 NULL인 경우: 리스트의 맨처음에 삽입**
  - **일반적인 경우: 리스트의 중간에 삽입**


phead
head
p
new_node
head
p
NULL
new_node
NULL

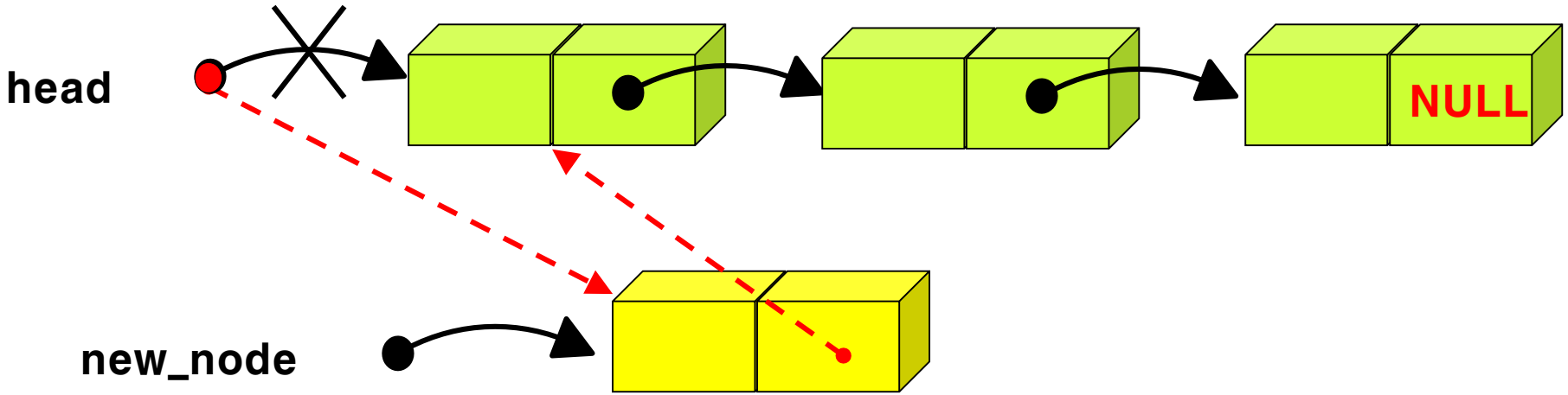
- **Head 포인터가 NULL인 경우 ?**
  - **삽입 노드가 첫 번째 노드.**
  - **head의 값만 변경.**

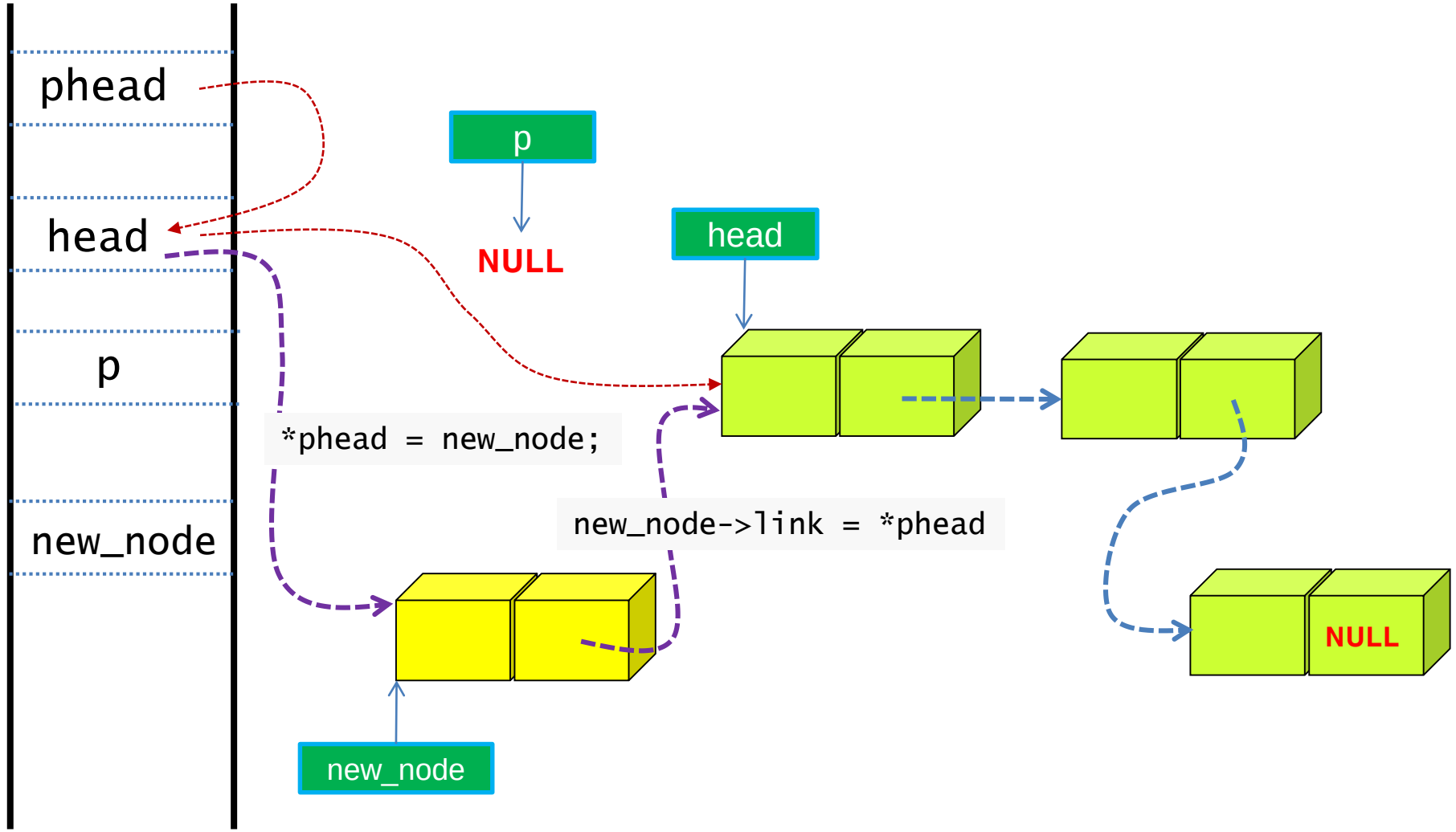
```
if( *phead == NULL ){    // 공백리스트인 경우
        new_node->link = NULL;
        *phead = new_node;
}
```



- **p가 NULL인 경우 ??**
  - 새로운 노드를 **리스트의 맨앞**에 삽입

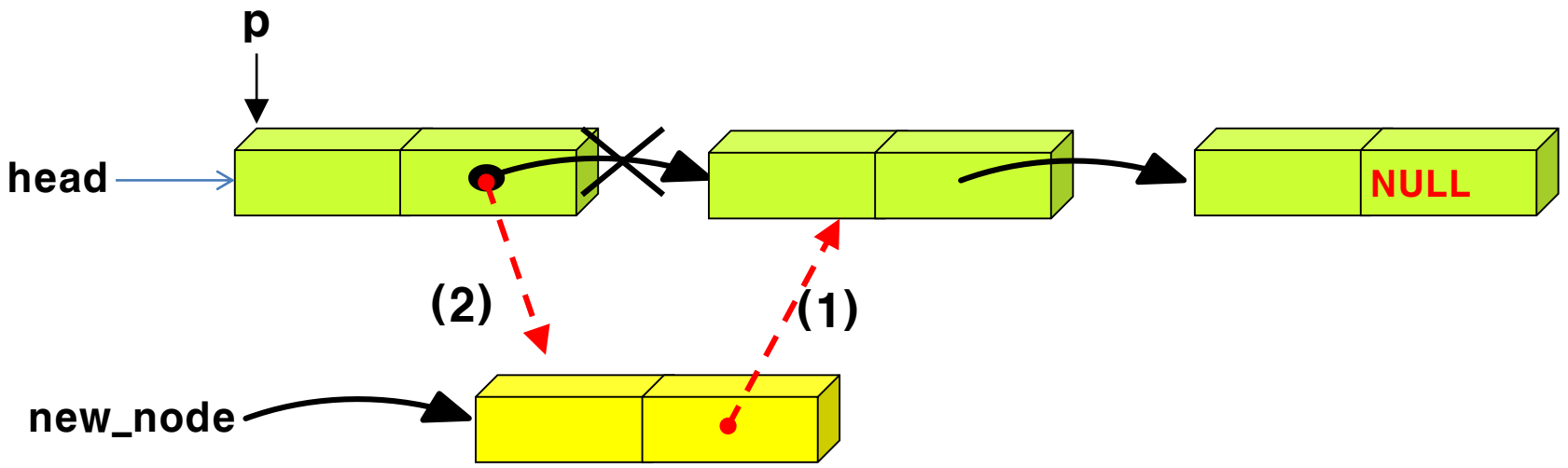
```
if( p == NULL )  { // p가 NULL이면 첫번째 노드로 삽입
        new_node->link = *phead;
        *phead = new_node;
}
```




phead
head
p
new_node
p
NULL
head
*phead = new_node;
new_node->link = *phead
NULL
new_node

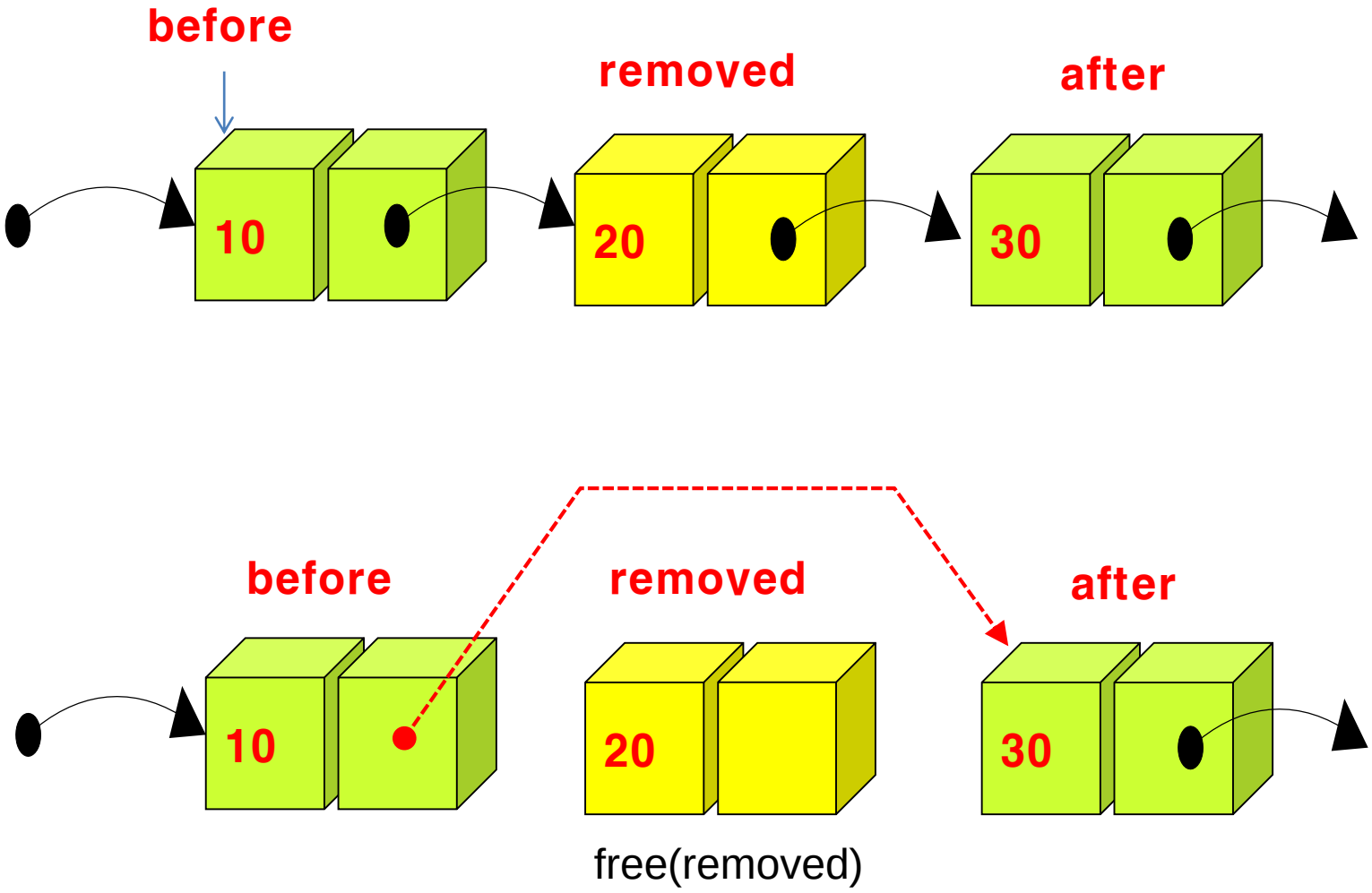
- **head와 p가 NULL이 아닌 경우 ??**
  - **(1) new_node의 link에 p->link값을 복사.**
  - **(2) p->link가 new_node를 가리키도록 함.**

```
else {                    // p 다음에 삽입
      new_node->link = p->link; // (1)
      p->link = new_node;       // (2)
}
```



```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드
// new_node : 삽입될 노드

void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)
{
    if( *phead == NULL ){       // 공백리스트인 경우
        new_node->link = NULL;
        *phead = new_node;
    }

    else if( p == NULL ){ // p가 NULL이면 첫번째 노드로 삽입
        new_node->link = *phead;
        *phead = new_node;
    }

    else {                          // p 다음에 삽입
        new_node->link = p->link;
        p->link = new_node;
    }
}
```

단순 연결 리스트의 삭제


before
removed
after
10
20
30
before
removed
after
10
20
30
free(removed)

- **삭제 함수의 프로토타입**

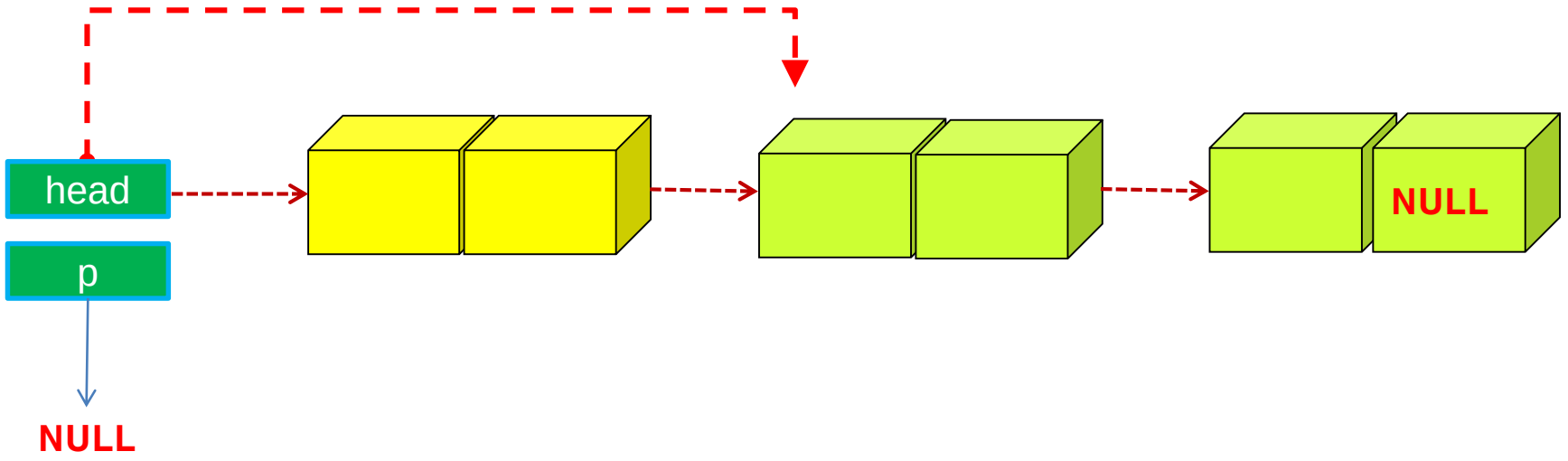
```
void remove_node(ListNode **phead, ListNode *p, ListNode *removed)
```

  - **phead**: 헤드 포인터의 포인터
  - **p**: 삭제될 노드의 선행 노드를 가리키는 포인터
  - **removed**: 삭제될 노드를 가리키는 포인터

- **삭제의 2가지 경우**
  - **p가 NULL인 경우**: 맨 앞의 노드를 삭제
  - **p가 NULL이 아닌 경우**: 중간 노드를 삭제

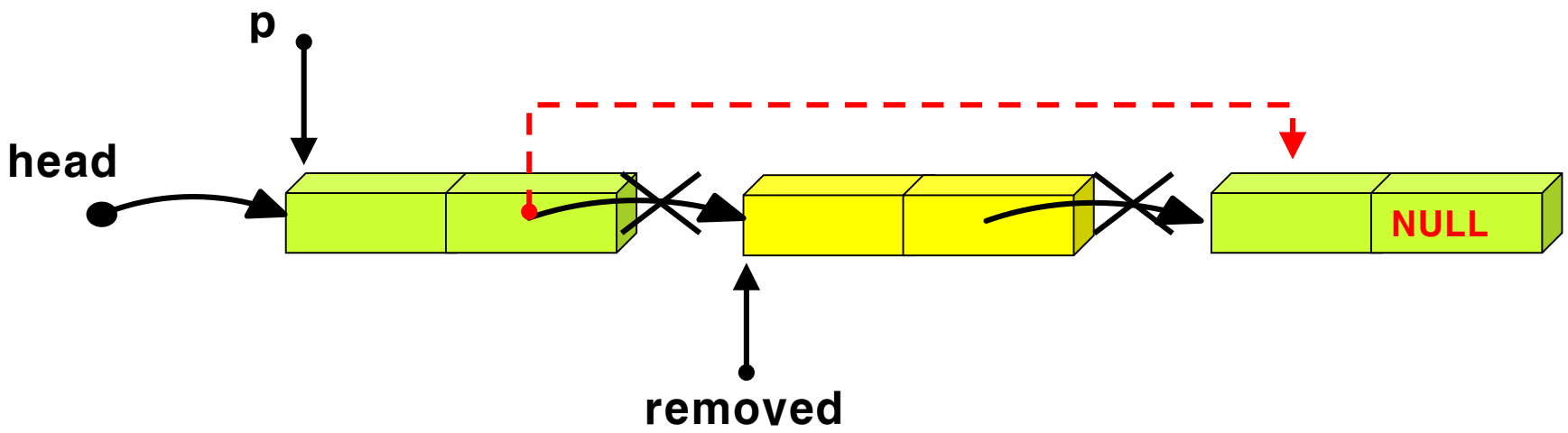
- **p가 NULL인 경우 ??**
  - 연결 리스트의 첫 번째 노드를 삭제.
  - 헤드포인터 변경.

```
if( p == NULL )
        *phead = (*phead)->link;
```



- **p가 NULL이 아닌 경우 ??**
  - removed 앞의 노드인 p의 링크가 removed 다음 노드를 가리키도록 변경

```
else
        p->link = removed->link;
        free(removed);
```



```
// phead : 헤드 포인터에 대한 포인터
// p: 삭제될 노드의 선행 노드
// removed: 삭제될 노드

void remove_node
   (ListNode **phead, ListNode *p, ListNode *removed)
{
   if( p == NULL )
        *phead = (*phead)->link;
   else
        p->link = removed->link;
   free(removed);
}
```

단순 연결 리스트의 방문 연산

- 방문 연산
  - 리스트 상의 노드를 순차적으로 방문
  - 반복과 순환 기법을 모두 사용가능

- 반복버전

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

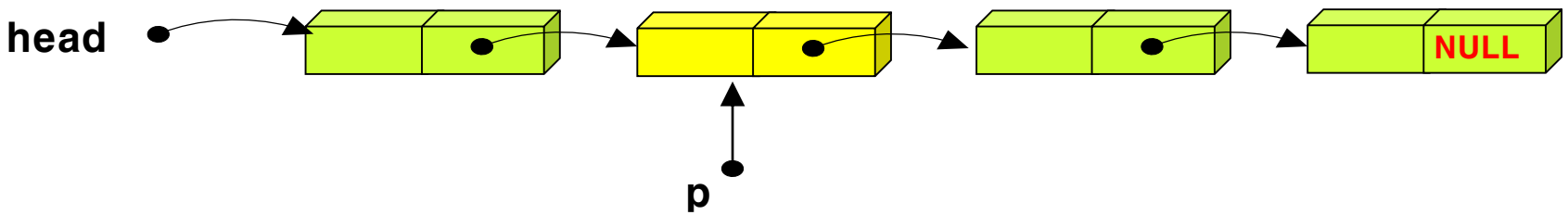
- **순환 버전**

```
void display_recur(ListNode *head)
{
      ListNode *p=head;
      if( p != NULL ){
            printf("%d->", p->data);
            display_recur(p->link);
      }
}
```

단순 연결 리스트의 탐색 연산

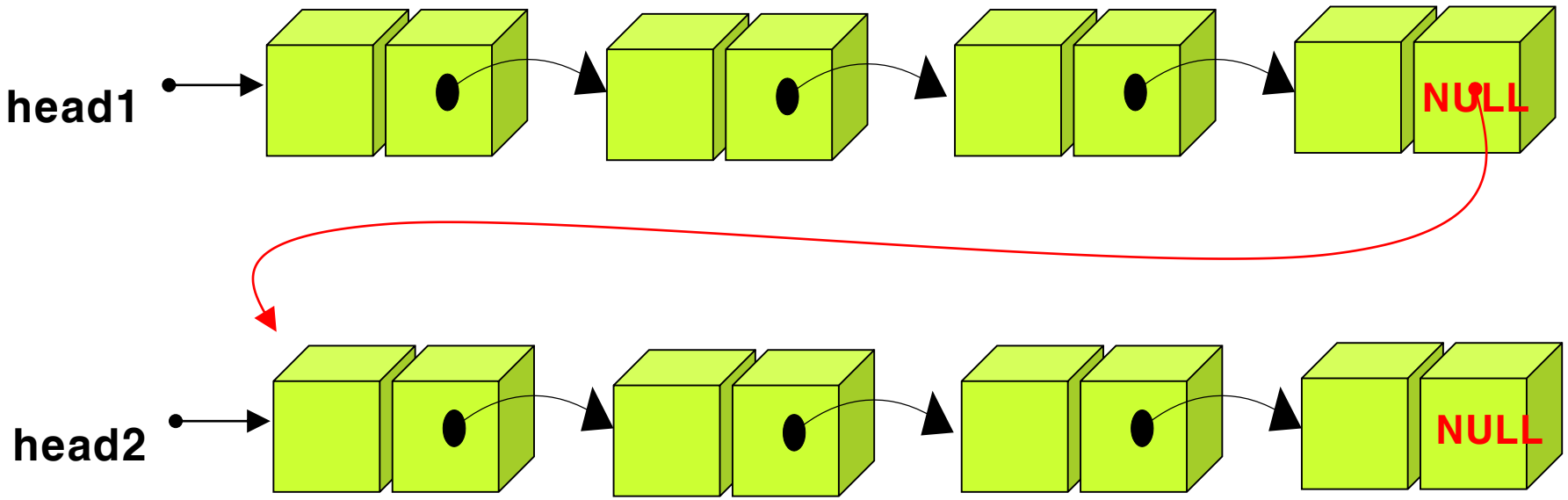
- **탐색 연산**
  - 특정한 데이터값을 갖는 노드를 찾는 연산

```
ListNode *search( ListNode *head, int x ) {
        ListNode *p;
        p = head;
        while( p != NULL ){
                if( p->data == x ) return p;  // 성공
                p = p->link;
        }
        return p;  // 탐색 실패일 경우 NULL 반환
}
```



단순 연결 리스트의 합병 연산

- **합병 연산**
  - 2개의 리스트를 합하는 연산



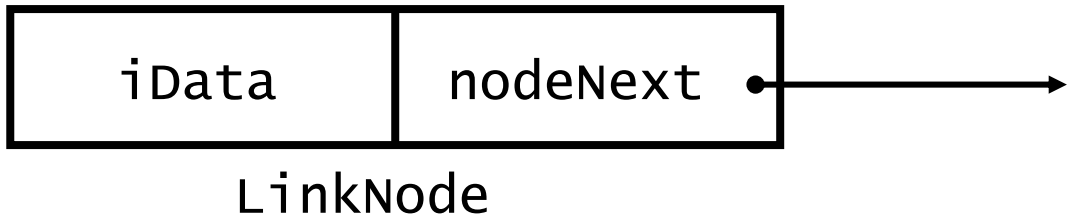
```
ListNode *concat(ListNode *head1, ListNode *head2)
{
  ListNode *p;

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

단순 연결 리스트 구현(in Java)

- 단순 연결 리스트(Simple Linked-List)
  - 연결 리스트의 가장 단순한 형태
  - 각 노드들은 오직 하나의 링크를 갖기 때문에 하나의 노드만을 가리킨다. -> 단방향성
  - 마지막 노드의 링크
    - null 값을 가리킬 경우는 리스트의 끝을 나타냄.
  - 헤더
    - null 값을 가리킬 경우는 빈 리스트를 나타냄
    - 값을 갖지 않는다.

노드 구조

```
class LinkNode {
        private int iData;                  // 자료(키)
        private LinkNode nodeNext;          //다음 노드포인터

        // 생성자
        public LinkNode( int iDataInput )
        {
                iData = iDataInput;
        }
}
```

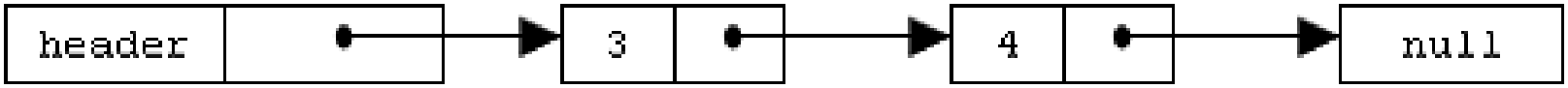


LinkNode 주요 메소드

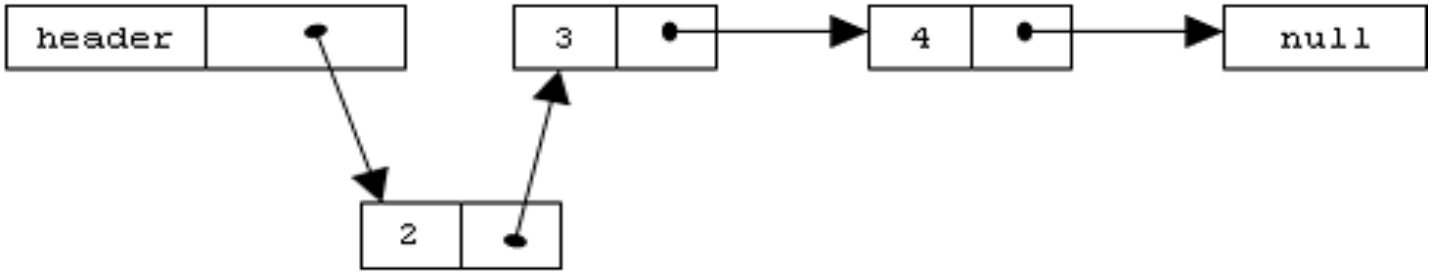
```
class LinkNode {
    public void displayNode()  {      // 자료를 출력 메소드
        System.out.print( iData );
    }
     public int getData()  {          // 자료에 접근 메소드
        return iData;
    }
    public void setData( int inputData ) {   //자료 변경 메소드
        iData = inputData;
    }
    public LinkNode getNodeNext() {   // 다음 노드에 접근하는 메소드
        return nodeNext;
    }

    // 다음 노드 참조자를 변경하는 메소드
    public void setNodeNext( LinkNode inputNode ) {
        nodeNext = inputNode;
    }
}
```

insertFirst() 메소드

- **insertFirst()**
  - **새로운 노드를 리스트의 가장 첫 번째 자리에 삽입하는 메소드**



a) 2를 삽입하기 전의 리스트

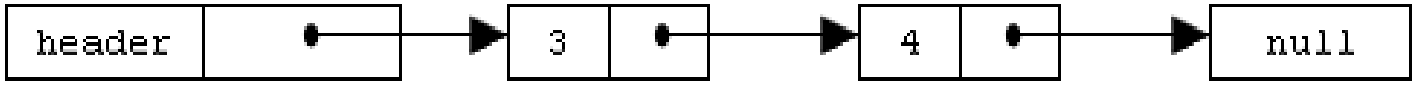


b) 2를 삽입한 후의 리스트

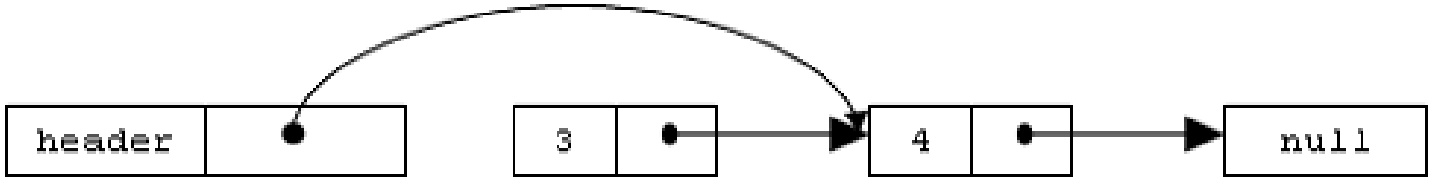
```
public void insertFirst( int iKey )
{

    // 새로운 노드 생성
    LinkNode nodeNew = new LinkNode(iKey);

    // 새로운 노드는 기존의 첫 번째 노드를 참조
    nodeNew.setNodeNext( HeaderNode.getNodeFirst() ) ;

    // 헤더는 새로운 노드를 참조
    HeaderNode.setNodeFirst( nodeNew );

}
```

deleteFirst() 메소드

- **deleteFirst()**
  - **리스트의 첫 번째 노드를 삭제하는 역할을 수행**



a) 3을 삭제하기 전의 리스트



b) 3을 삭제한 후의 리스트

```
public LinkNode deleteFirst()  {

      // 빈 리스트인지 확인
      if (HeaderNode.isEmpty() == false) {

         LinkNode tempNode = HeaderNode.getNodeFirst();

         // 헤더는 두 번째 노드를 참조
         HeaderNode.setNodeFirst
             ( HeaderNode.getNodeFirst().getNodeNext() );

               return tempNode;  // 삭제된 노드를 반환
      }
      else return null;              // 빈 리스트라면 null값 반환
}
```

findNode() 메소드

- findNode()
  - 리스트 내에서 특정 자료를 갖고 있는 노드를 찾는 메소드

```
public LinkNode findNode( int iKey ) {

    // 리스트의 첫 번째 노드부터 탐색
    LinkNode current = HeaderNode.getNodeFirst();

    // 키 값을 탐색
    while (current.getData() != iKey) {
        current = current.getNodeNext();    // 다음 노드 검색.
    }
    return current;          // 키 값 갖고 있는 노드를 반환
}
```

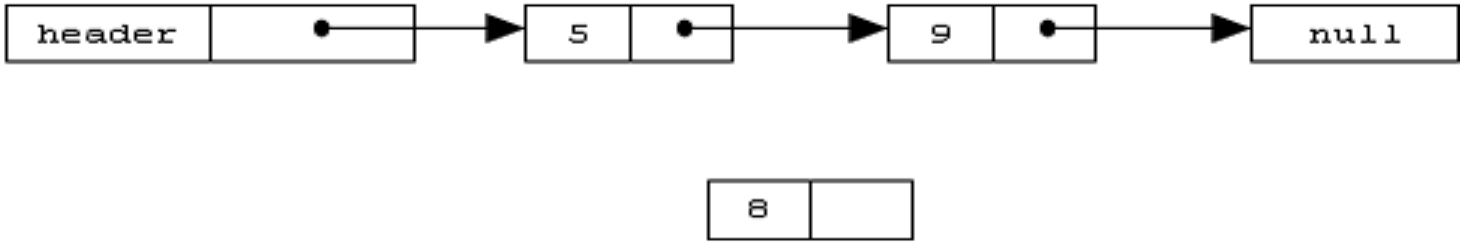
displayList() 메소드

- **displayList() 메소드**
  - 리스트로 연결되어 있는 모든 자료들 즉, 각 노드에 저장되어 있는 자료들을 출력하는 메소드

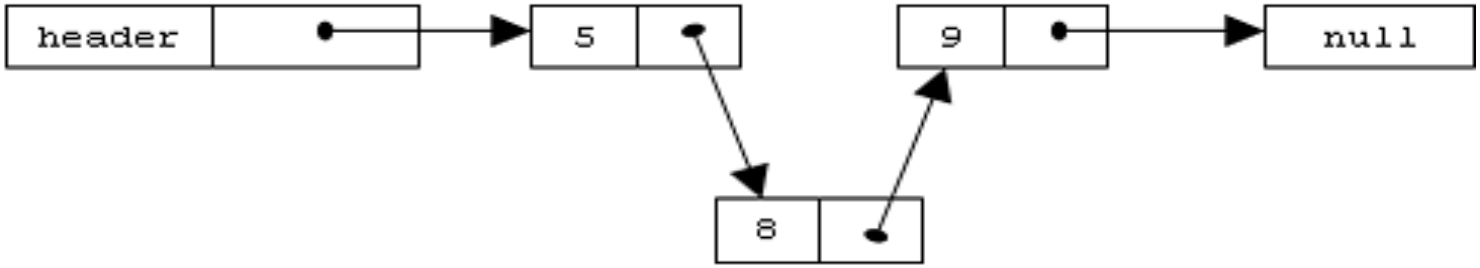
```
public void displayList() {

    // 리스트의 첫 번째 노드부터 출력
    LinkNode current = HeaderNode.getNodeFirst();

    while (current != null)  {
        current.displayNode();    // 노드 자료 출력
        System.out.println("");
        current = current.getNodeNext();   // 다음 노드 탐색
    }
}
```

insertNode() 메소드

- **insertNode()**
  - 리스트에 새로운 노드를 특정 위치에 삽입하는 메소드



a) 8을 삽입하기 전의 리스트

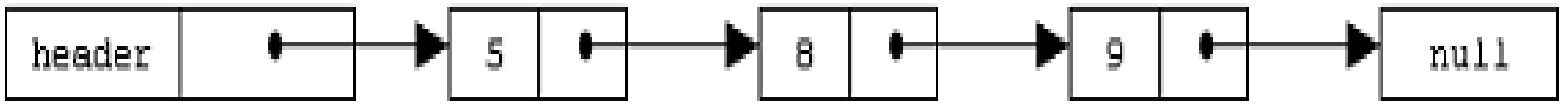


b) 8을 삽입한 후의 리스트

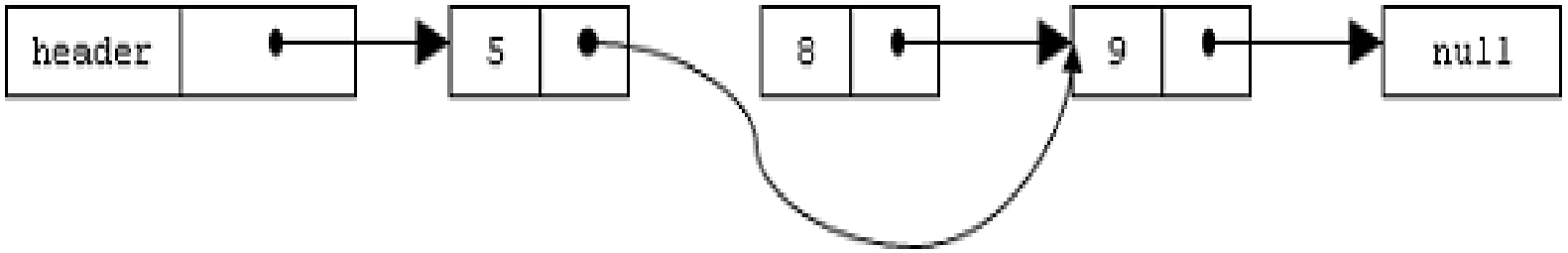
```
public void insertNode( int iKey ) {
        // 새로운 노드 생성
        LinkNode nodeNew = new LinkNode( iKey );

        // 첫 번째 노드부터 검색
        LinkNode current = HeaderNode.getNodeFirst();
        LinkNode previous = null;

        // 빈 리스트가 아니고, 마지막 노드도 아니며, 새로운 키 값이 더 클 경우,
        while (current != null && iKey > current.getData()) {
                previous = current;
                current = current.getNodeNext();
        }

        if (previous == null)    // 빈 리스트, 새로운 노드를 첫 번째 자리로 삽입
                HeaderNode.setNodeFirst( nodeNew );
       else                              // 이전 노드 참조자가 새로운 노드를 가리키도록 변경
                previous.setNodeNext( nodeNew );

        // 새로운 노드의 다음 노드 참조자를 현재 노드를 기리키도록 변경
                nodeNew.setNodeNext( current );
}
```

deleteNode() 메소드

- **deleteNode()**
  - 리스트에서 특정 노드를 검색한 후 삭제하는 메소드



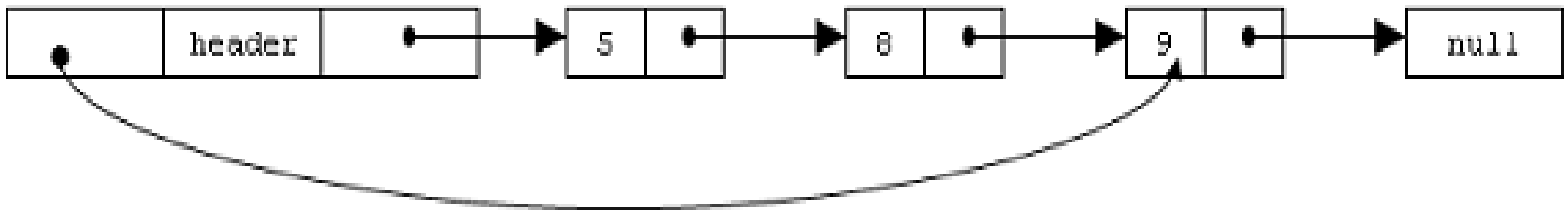
a) 8을 삭제하기 전의 리스트



b) 8을 삭제 한 후의 리스트

```
public LinkNode deleteNode( int iKey ) {

	// 첫 번째 노드부터 검색
	LinkNode current = HeaderNode.getNodeFirst();
	LinkNode previous = null;


	// 빈 리스트가 아니고, 마지막 노드도 아니며, 찾는 키 값이 아닐 경우,
	while (current != null && iKey != current.getData()) {
		previous = current;
		current = current.getNodeNext();
	}


	// 빈 리스트 & 마지막 노드가 아닌 경우, 포인터 변경하고 삭제된 노드 반환
	if (previous != null && current != null) {
		previous.setNodeNext( current.getNodeNext() );
	}

	else if (previous == null && current != null) {
		HeaderNode.setNodeFirst( current.getNodeNext() );
	}

	return current;
}
```

- **개념**
  - **단순 연결 리스트와 유사한 구조**
  - **헤더가 첫 번째 노드뿐만 아니라 마지막 노드를 가리키는 포인터를 함께 갖고 있음**



감시 노드

```
class LinkSentinelNode {
    private LinkNode nodeFirst;    // 첫 번째 노드를 참조
    private LinkNode nodeLast;     // 마지막 노드를 참조


    // 생성자
    public LinkSentinelNode() {
        nodeFirst = null;
        nodeLast = null;
    }

    // 첫 번째 노드를 참조하는 포인터 접근 메소드
    public LinkNode getNodeFirst()  {
        return nodeFirst;
    }
}
```

```
class LinkSentinelNode {
    // 첫 번째 노드를 참조하는 포인터 변경 메소드
    public void setNodeFirst( LinkNode inputNode ) {
        nodeFirst = inputNode;
    }
    // 마지막 노드를 참조하는 포인터 접근 메소드
    public LinkNode getNodeLast() {
        return nodeLast;
    }
    // 마지막 노드를 참조하는 포인터 변경 메소드
    public void setNodeLast( LinkNode inputNode ) {
        nodeLast = inputNode;
    }
    // 빈 리스트인지 확인하는 메소드
    public boolean isEmpty() {
         return (nodeFirst == null);
    }
}
```

insertNode()

```
public void insertNode( int iKey ) {
    // 새로운 노드 생성
    LinkNode nodeNew = new LinkNode( iKey );

    // 첫 번째 노드부터 검색
    LinkNode current = HeaderNode.getNodeFirst();
    LinkNode previous = null;

    // 빈 리스트 & 마지막 노드도 아니며, 새로운 키 값이 더 클 경우,
    while (current != null && iKey > current.getData()) {
        previous = current;
        current = current.getNodeNext();
    }

    // 빈 리스트일 경우
    if (HeaderNode.isEmpty() == true) {
        // 새로운 노드를 첫 번째 자리로 삽입
        HeaderNode.setNodeFirst( nodeNew );
        HeaderNode.setNodeLast( nodeNew );
    }
```

```
    // 새로운 노드를 첫 번째 자리로 삽입
    else if (HeaderNode.getNodeFirst() == current) {
        HeaderNode.setNodeFirst( nodeNew );
    }

    // 새로운 노드를 마지막 자리로 삽입
    else if (current == null) {
        previous.setNodeNext( nodeNew );
        HeaderNode.setNodeLast( nodeNew );
    }

    // 새로운 노드를 중간에 삽입
    else previous.setNodeNext( nodeNew );

    nodeNew.setNodeNext( current );
}
```

deleteNode()

```
public LinkNode deleteNode( int iKey ) {
    LinkNode current = HeaderNode.getNodeFirst();
    LinkNode previous = null;

    //빈 리스트 & 마지막 노드도 아니며, 찾는 키 값이 아닐 경우
    while(current != null && iKey != current.getData()) {
        previous = current;
        current = current.getNodeNext();
    }
    // 첫 번째 노드를 삭제할 경우
    if (HeaderNode.getNodeFirst() == current)  {
        if (HeaderNode.getNodeFirst() ==
                          HeaderNode.getNodeLast()) {
            HeaderNode.setNodeFirst( null );
            HeaderNode.setNodeLast( null );
        }

        else
         HeaderNode.setNodeFirst( current.getNodeNext() );
    }
```

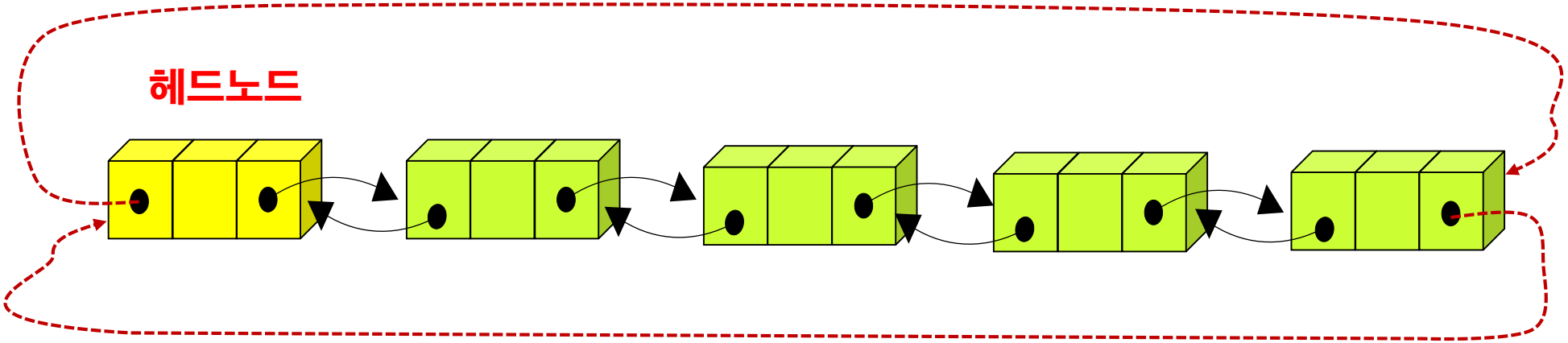
```
        else if (HeaderNode.getNodeLast() == current) {
            previous.setNodeNext( null );
            HeaderNode.setNodeLast( previous );
        }

        // 중간 노드를 삭제할 경우
        else {
           previous.setNodeNext( current.getNodeNext());
        }

        // 삭제된 노드를 반환
        return current;
    }
```

displayLastNode()

```
public void displayLastNode() {
        HeaderNode.getNodeLast().displayNode();
        System.out.println("");
    }
```

5. 이중 연결 리스트(doubly linked list)

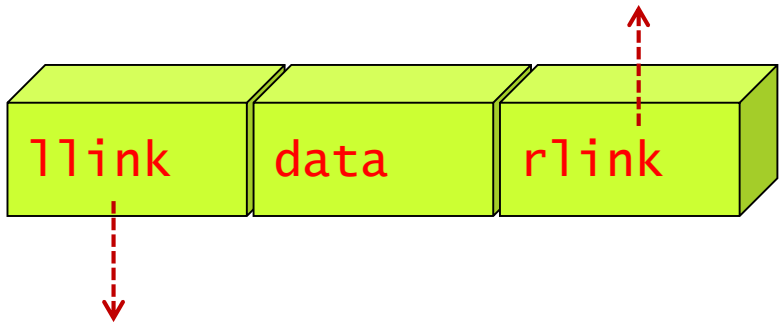
- 개념
  - 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
  - 링크가 양방향이므로 양방향으로 검색이 가능
  - 공간을 많이 차지하고 코드가 복잡
  - 실제 사용되는 이중연결 리스트의 형태
    - 헤드 노드+ 이중 연결 리스트+ 원형 연결 리스트



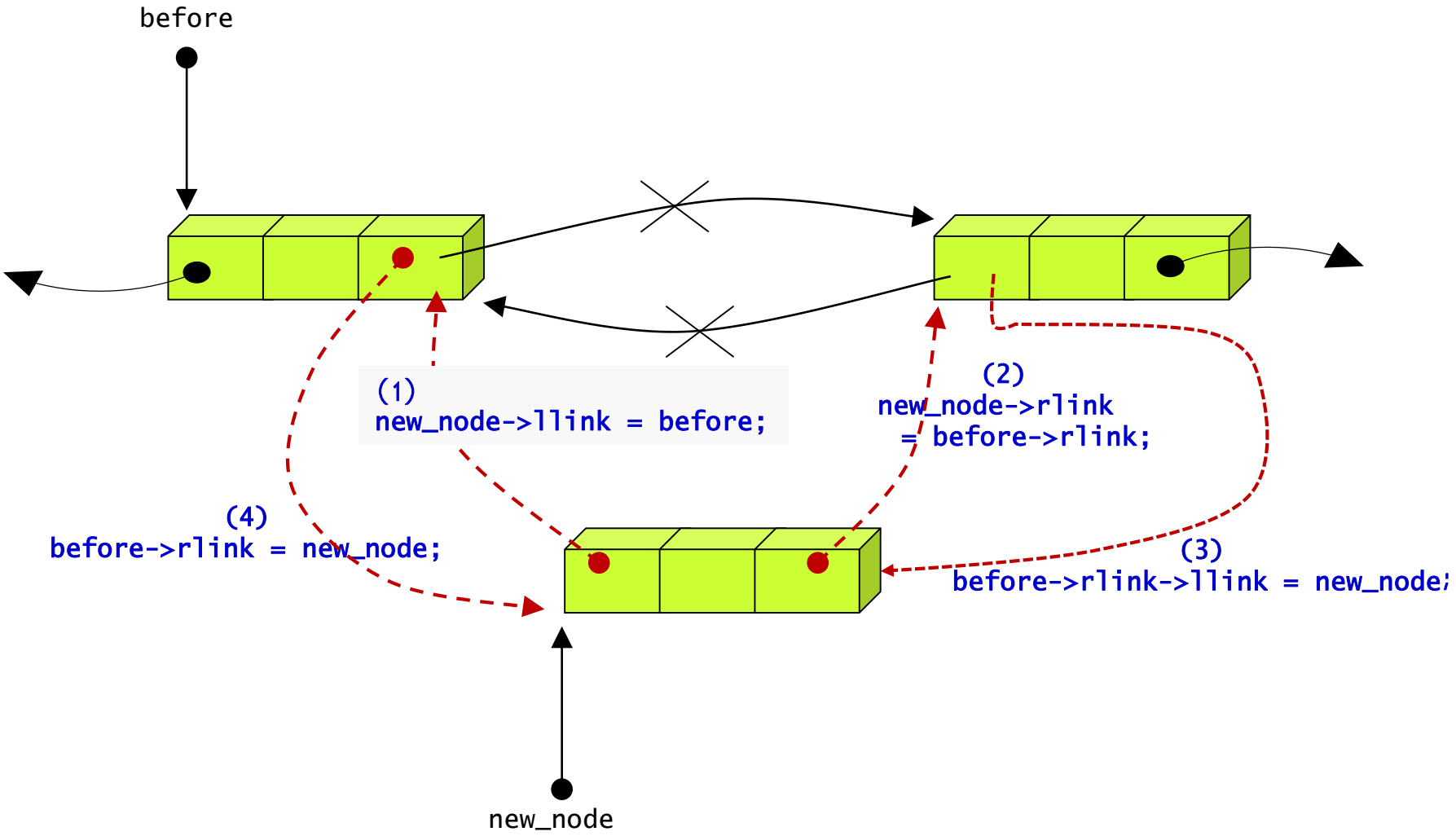
- **헤드 노드(head node)**
  - 데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드
  - 헤드 포인터와의 구별 필요
  - 공백 상태에서는 헤드 노드만 존재

- **이중 연결 리스트의 노드 구조**

```
typedef struct DlistNode {
   int data;
   struct DlistNode *llink;
   struct DlistNode *rlink;
} DlistNode;
```

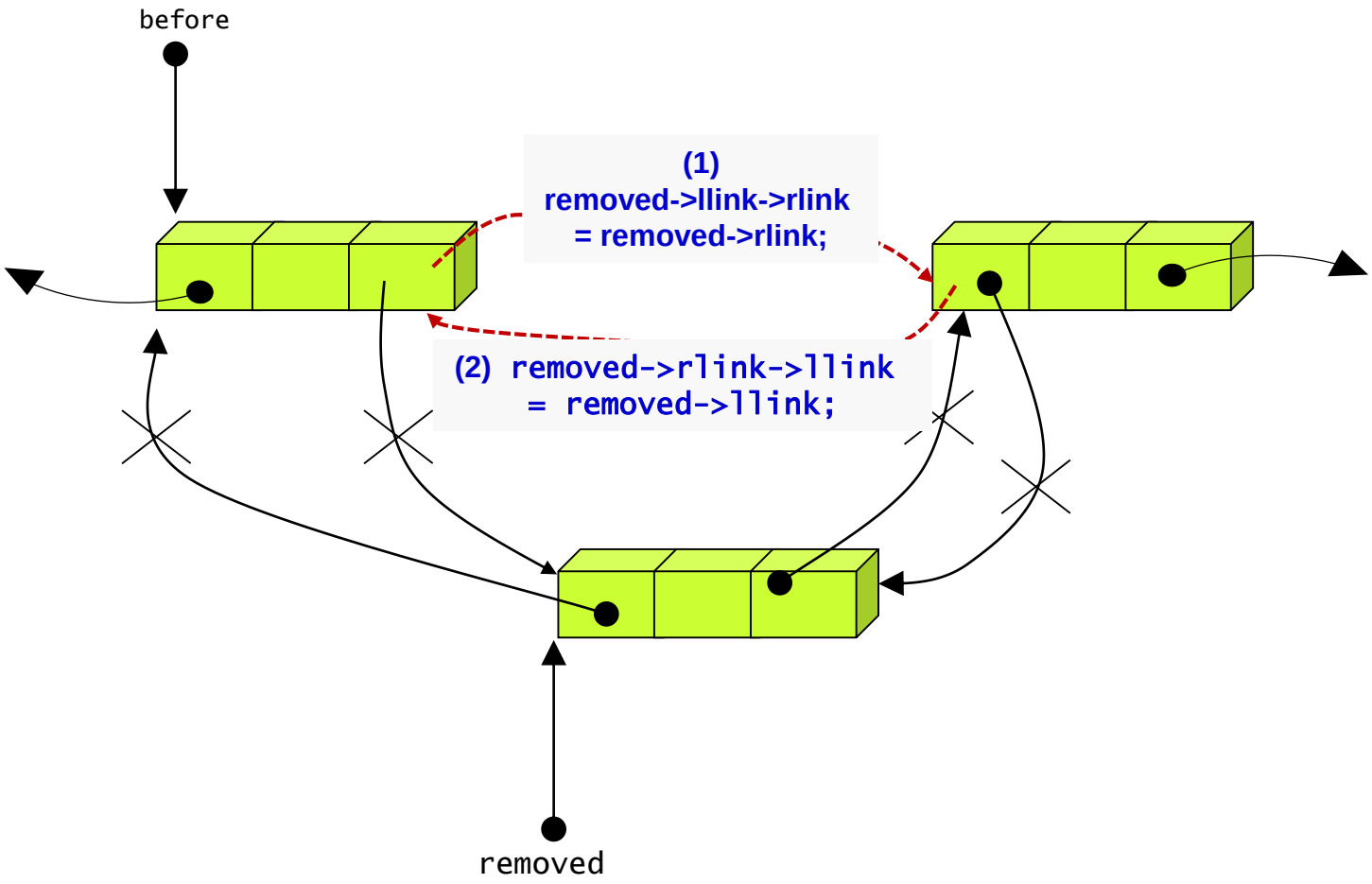


삽입 연산


before
(1)
new_node->llink = before;
(2)
new_node->rlink
= before->rlink;
(3)
before->rlink->llink = new_node;
(4)
before->rlink = new_node;
new_node

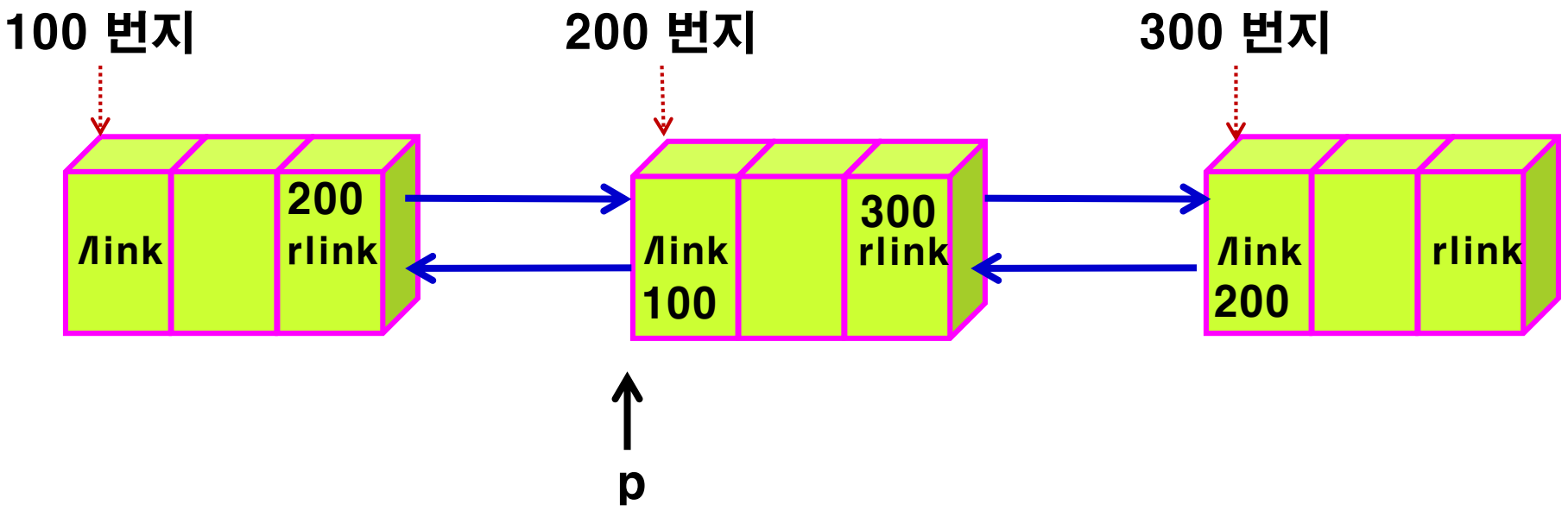
```
// 노드 new_node를 노드 before의 오른쪽에 삽입한다.
void dinsert_node(DlistNode *before, DlistNode *new_node)
{
  new_node->llink = before; // 1)
  new_node->rlink = before->rlink; // 2)
  before->rlink->llink = new_node; // 3)
  before->rlink = new_node; // 4)
}
```

삭제 연산


before
(1)
removed->llink->rlink
= removed->rlink;
(2) removed->rlink->llink
= removed->llink;
removed

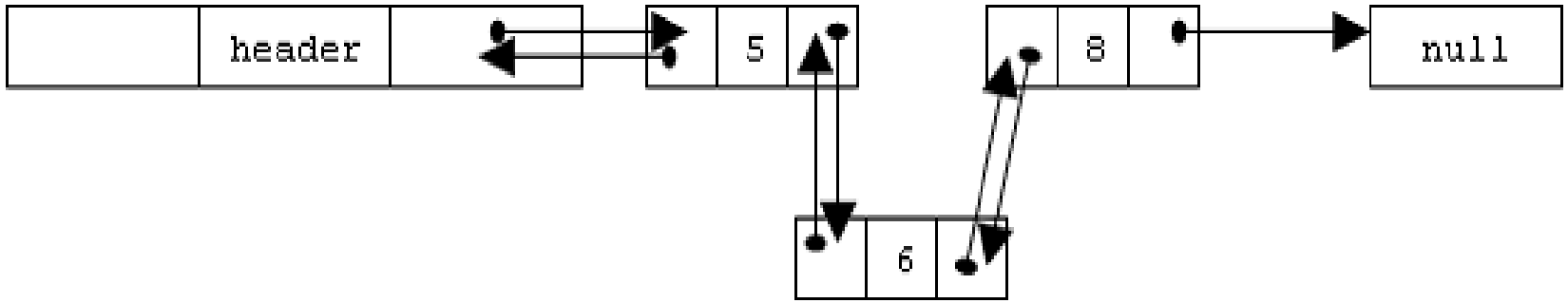
```
// 노드 removed를 삭제한다.
void dremove_node(DlistNode *phead_node,
                  DlistNode *removed)
{
   if( removed == phead_node )
       return;
   removed->llink->rlink = removed->rlink; // 1)
   removed->rlink->llink = removed->llink; // 2)
   free(removed);
}
```

p
= p->llink->rlink
= p->rlink->llink



이중 연결 리스트 insertFirst()

- **insertNode() 메소드**
  - 일반 노드 추가시 nodePrevious 참조자를 위한 과정이 추가
  - nodeNext와 nodePrevious가 다음 노드와 이전 노드를 참조하는 참조자의 역할



```
public void insertNode( int iKey ) {
    // 새로운 노드 생성
    LinkNode nodeNew = new LinkNode( iKey );
    // 첫 번째 노드부터 검색
    LinkNode current = HeaderNode.getNodeFirst();
    LinkNode previous = null;
    // 마지막 노드도 아니며, 새로운 키 값이 더 클 경우,
    while (current != null && iKey > current.getData()) {
        previous = current;
        current = current.getNodeNext();
}
```

```
// 빈 리스트일 경우
if (HeaderNode.isEmpty() == true) {
    // 새로운 노드를 첫 번째 자리로 삽입
    HeaderNode.setNodeFirst( nodeNew );
    nodeNew.setNodeNext( null );
    nodeNew.setNodePrevious( null );
}
// 새로운 노드를 첫 번째 자리로 삽입
else if (HeaderNode.getNodeFirst() == current) {
    nodeNew.setNodeNext( HeaderNode.getNodeFirst() );
    HeaderNode.setNodeFirst( nodeNew );
    nodeNew.setNodePrevious( null );
}
```

```
        // 새로운 노드를 마지막 자리로 삽입
        else if (current == null) {
            previous.setNodeNext( nodeNew );
            nodeNew.setNodePrevious( previous );
            nodeNew.setNodeNext( null );
        }
        // 새로운 노드를 중간에 삽입
        else {
            nodeNew.setNodeNext( previous.getNodeNext() );
            previous.setNodeNext( nodeNew );
            nodeNew.setNodePrevious( previous );
            nodeNew.getNodeNext().setNodePrevious( nodeNew );
        }
    }
```

이중 연결 리스트 deleteNode()

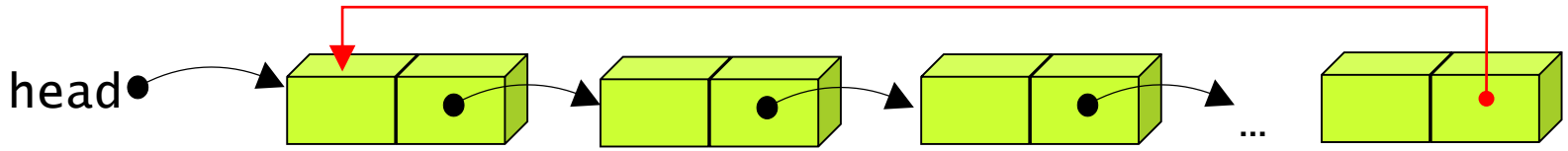
- **deleteNode() 메소드**
  - **nodePrevious 참조자**

```
public LinkNode deleteNode( int iKey ) {

    // 첫 번째 노드부터 검색
    LinkNode current = HeaderNode.getNodeFirst();
    LinkNode previous = null;

    // 빈 리스트, 마지막 노드, 찾는 키 값이 아닐 경우,
    while (current != null && iKey !=
                        current.getData())   {
        previous = current;
        current = current.getNodeNext();
    }
```

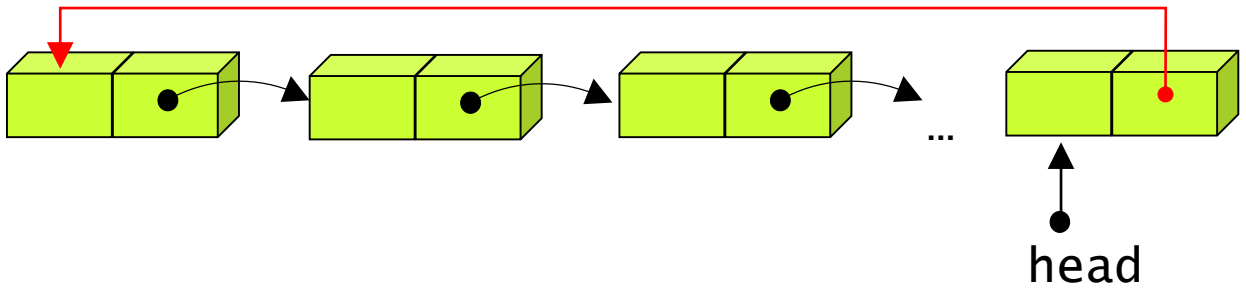
```
    // 첫 번째 노드를 삭제할 경우
    if (HeaderNode.getNodeFirst() == current) {
        HeaderNode.setNodeFirst( current.getNodeNext() );
    }
    // 마지막 노드를 삭제할 경우
    else if (current.getNodeNext() == null) {
        previous.setNodeNext( null );
    }
    // 중간 노드를 삭제할 경우
    else {
        previous.setNodeNext( current.getNodeNext() );
        current.getNodeNext().setNodePrevious( previous );
    }
    // 삭제된 노드를 반환
    return current;
}
```

6. 원형 연결 리스트

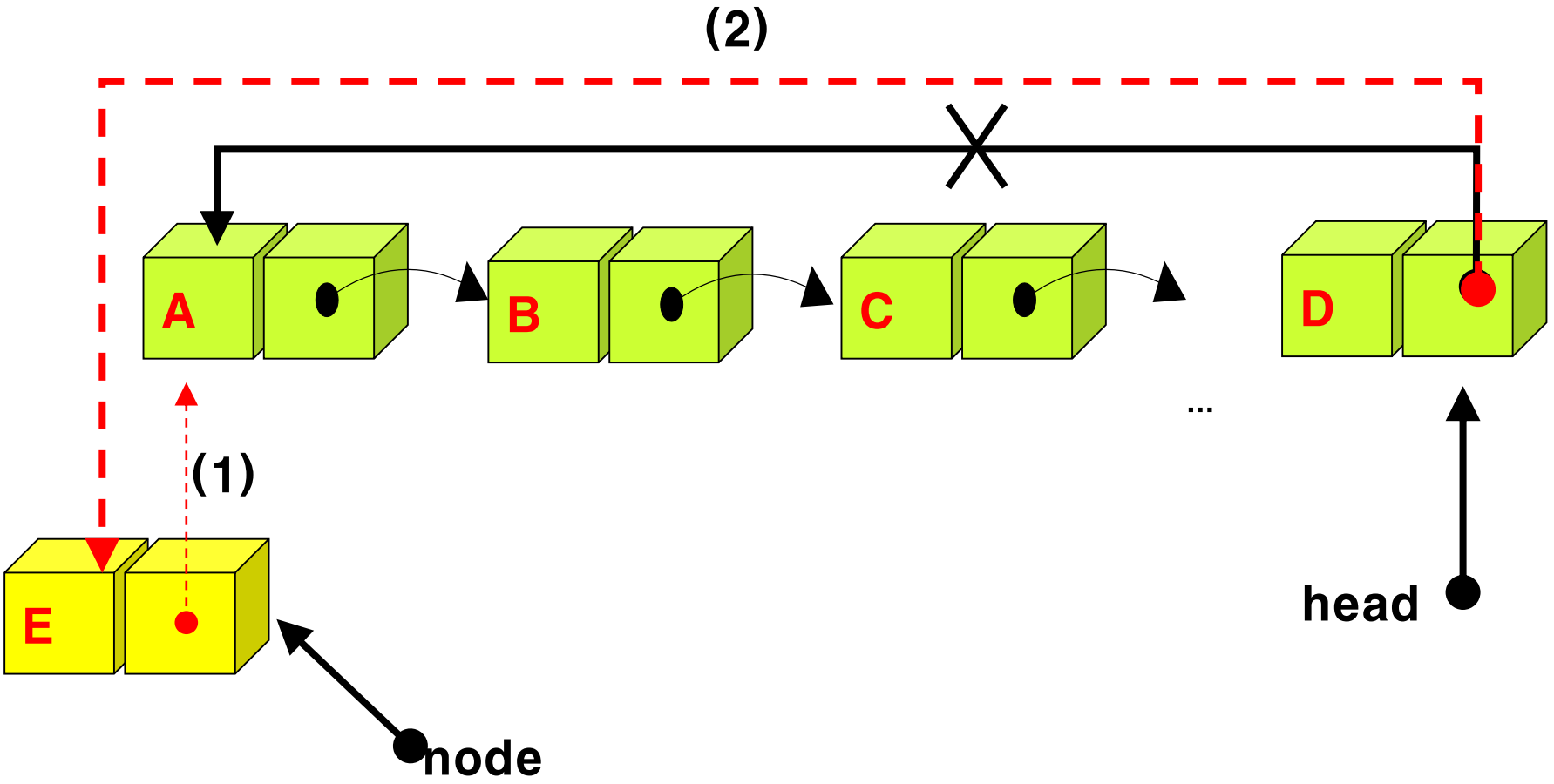
- 원형 연결 리스트(Circular linked list)
  - 마지막 노드의 링크가 첫번째 노드를 가리키는 리스트
  - 한 노드에서 다른 모든 노드로의 접근이 가능



  - 보통 헤드포인터가 마지막 노드를 가리키게끔 구성하면 리스트의 처음이나 마지막에 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이

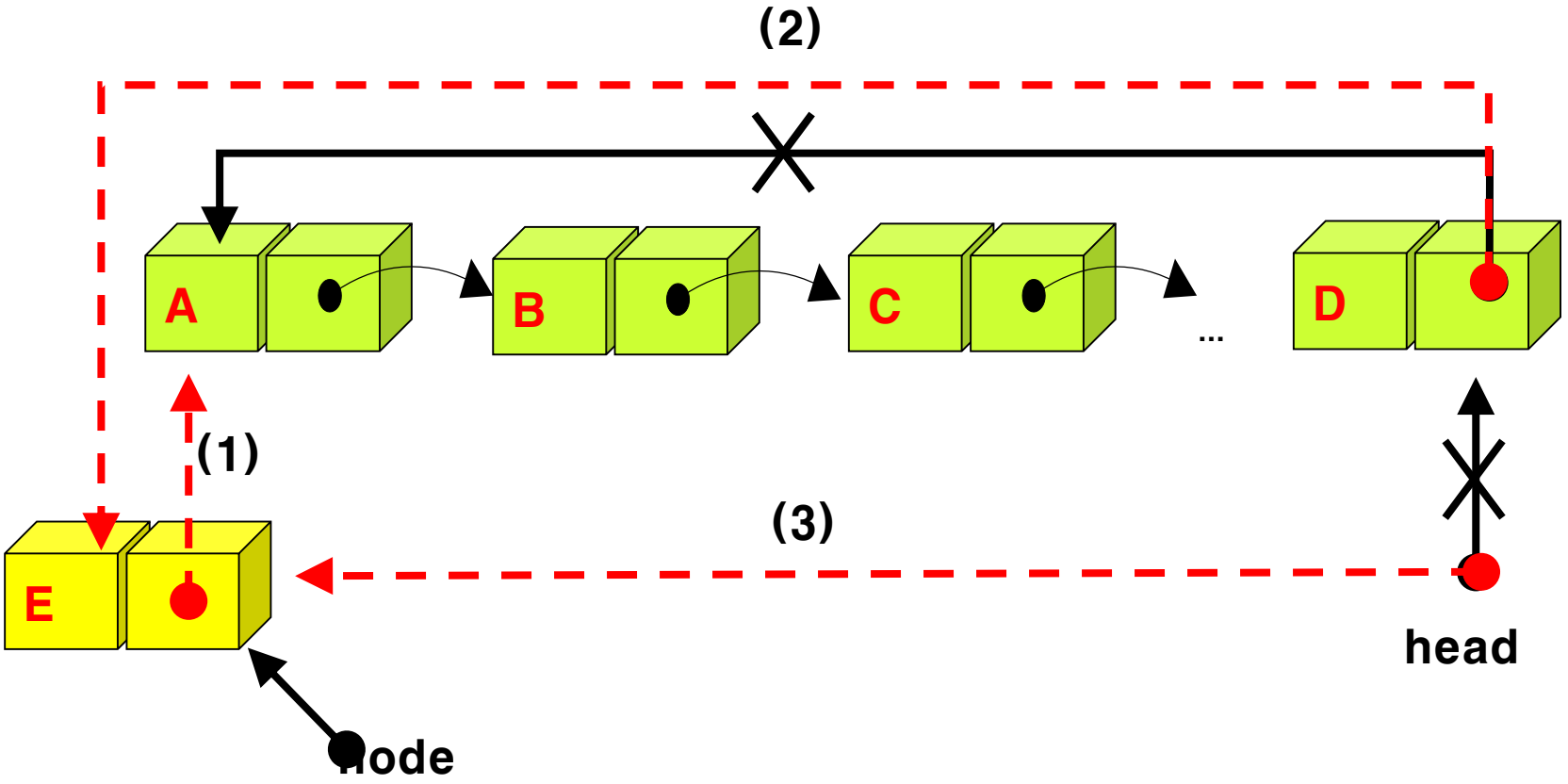


원형 연결 리스트의 처음에 삽입



```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드, // node : 삽입될 노드
void insert_first(ListNode **phead,     ListNode *node) {
   if( *phead == NULL ) {
        *phead = node;
        node->link = node;
   }
   else {
        node->link = (*phead)->link;
        (*phead)->link = node;
   }
}
```

원형 연결 리스트의 끝에 삽입



```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드, // node : 삽입될 노드
void insert_last(ListNode **phead, ListNode *node) {
   if( *phead == NULL ){
        *phead = node;
        node->link = node;
   }
   else {
        node->link = (*phead)->link;
        (*phead)->link = node;
        *phead = node;
   }
}
```