COSE213: Data Structure

Lecture 5 - 연결 리스트 (Linked List)

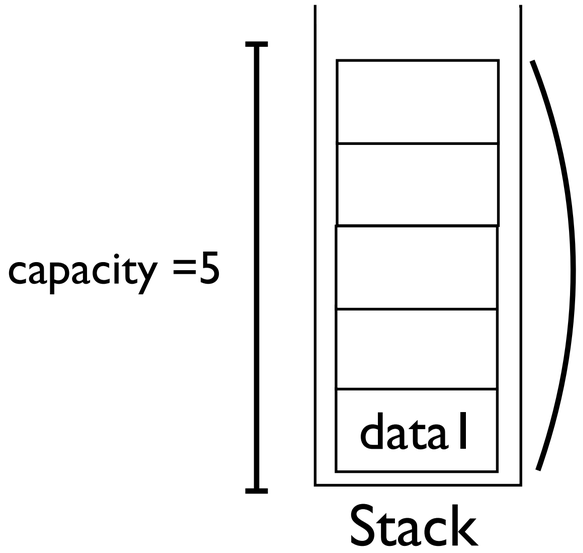
Minseok Jeon

2024 Fall

배열을 사용한 자료구조 구현의 문제점

- 동적으로 크기 조절이 불가능함

  (1) 큰 배열을 생성하고 그 중 일부만 사용한다면 메모리 공간이 낭비 됨



- 자료구조를 위해 생성된 배열이 차지하고 있는 공간 = 5
- 실제로 사용하고 있는 공간 = 1

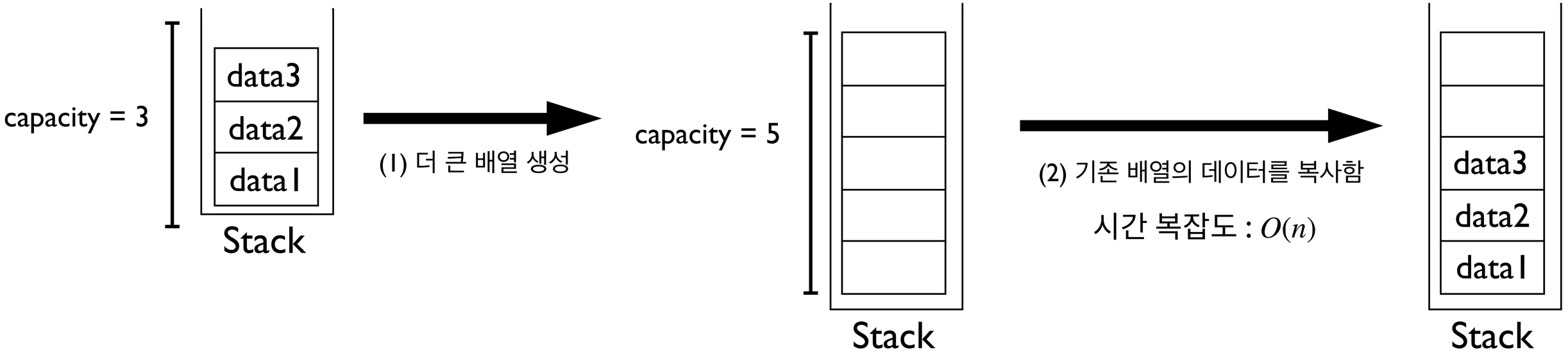
```c
#define MAX 5  // capacity

typedef struct {
    int *items;
    int top;
    int capacity;
} Stack;
```

배열을 사용한 자료구조 구현의 문제점

- 동적으로 크기 조절이 불가능함

  (1) 큰 배열을 생성하고 그 중 일부만 사용한다면 메모리 공간이 낭비 됨

  (2) 자료구조의 용량(capacity)를 늘릴 때 많은 연산이 필요함



배열을 사용한 자료구조 구현의 문제점

- 동적으로 크기 조절이 불가능함

  (1) 큰 배열을 생성하고 그 중 일부만 사용한다면 메모리 공간이 낭비 됨

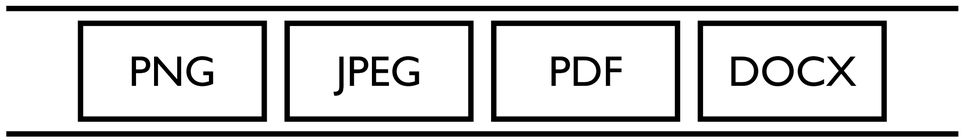
  (2) 자료구조의 용량(capacity)를 늘릴 때 많은 연산이 필요함

- 모든 데이터가 동일한 타입을 가져야 함

```c
typedef struct {
    int *items;
    int top;
    int capacity;
} Stack;
```

```c
typedef struct {
    int *items;
    int front;
    int rear;
    int size;
    int capacity;
} Queue;
```

  - 다른 타입을 가지는 데이터를 처리해야 하는 경우: 프린터 작업 대기열 (큐)



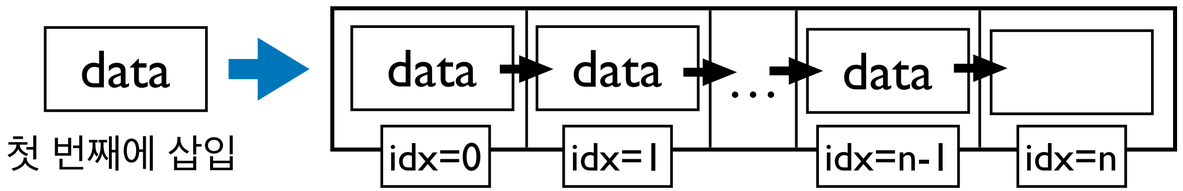
배열을 사용한 자료구조 구현의 문제점

- 동적으로 크기 조절이 불가능함

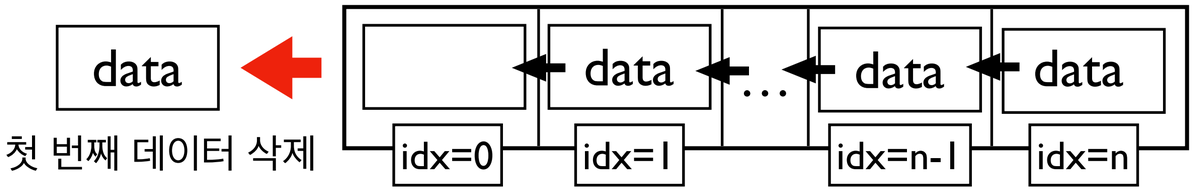
  (1) 큰 배열을 생성하고 그 중 일부만 사용한다면 메모리 공간이 낭비 됨

  (2) 자료구조의 용량(capacity)를 늘릴 때 많은 연산이 필요함

- 모든 데이터가 동일한 타입을 가져야 함
- 데이터 삽입과 삭제가 비효율적임



시간 복잡도 : $O(n)$



시간 복잡도 : $O(n)$

배열을 사용한 자료구조 구현의 문제점

- 동적으로 크기 조절이 불가능함

  (1) 큰 배열을 생성하고 그 중 일부만 사용한다면 메모리 공간이 낭비 됨

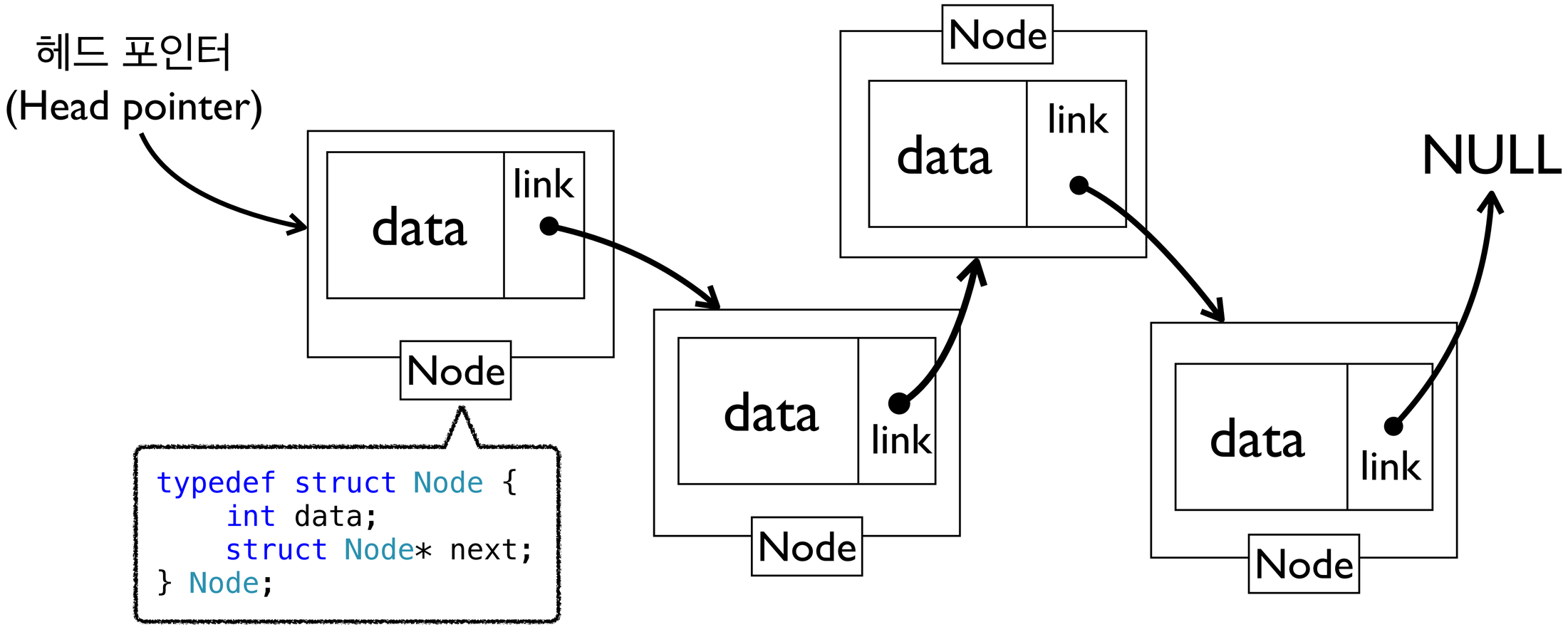
  (2) 자료구조의 용량(capacity)를 늘릴 때 많은 연산이 필요함

- 모든 데이터가 동일한 타입을 가져야 함
- 데이터 삽입과 삭제가 비효율적임
- 연속된 충분한 메모리 공간이 필요함

  - 큰 배열을 선언하려고 할 때 충분한 연속된 공간이 없다면 메모리 할당이 실패할 수 있음

연결 리스트 (Linked List)

- 연결 리스트는 데이터를 저장하고 있는 노드(Node)들의 집합이며 서로 논리적으로 연결되어 있음



연결 리스트 (Linked List)

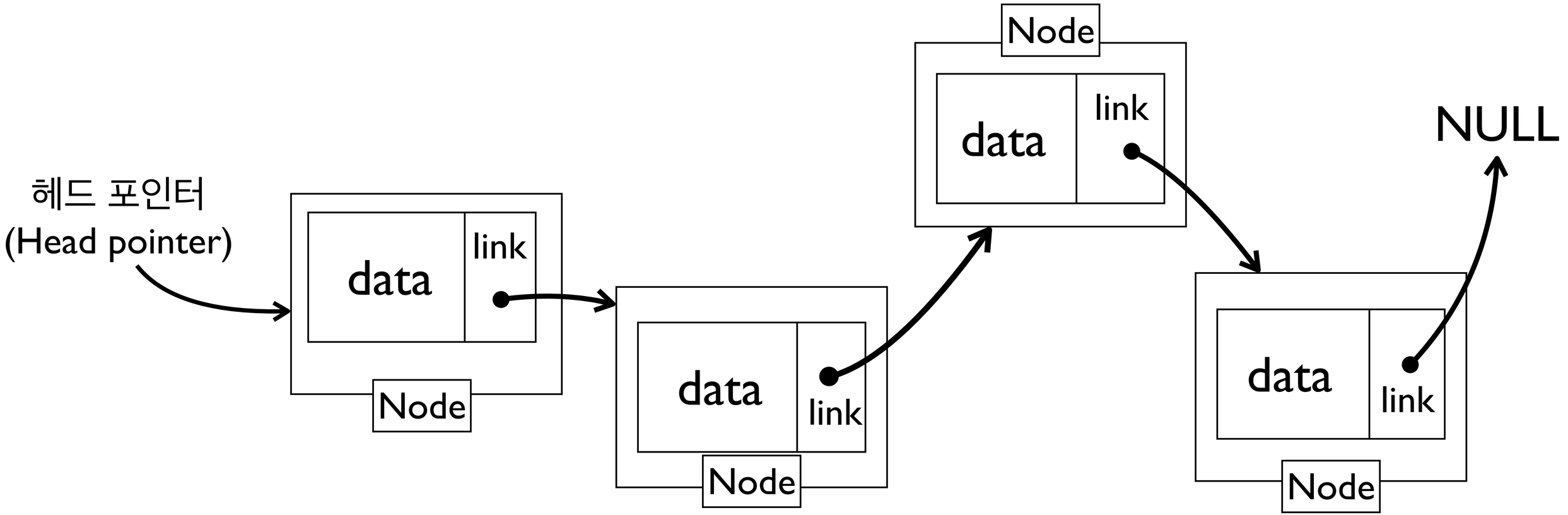
```c
#include <stdio.h>
#include <stdlib.h>

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

int main() {
    Node* head = NULL;
    Node* node1 = (Node*) malloc(sizeof(Node));
    Node* node2 = (Node*) malloc(sizeof(Node));
    node1->data = 1;
    node2->data = 2;
    node2->next = NULL;
    head = node1;
    node1->next = node2;
    printf("%d -> %d\n", head->data, head->next->data);
    printf("%p\n", &(head->data));
    printf("%p\n", &(head->next->data));

    int items[2];
    items[0] = 1;
    items[1] = 2;
    printf("%d -> %d\n", items[0], items[1]);
    printf("%p\n", &items[0]);
    printf("%p\n", &items[1]);
    return 0;
}
```

연결 리스트 : 데이터를 저장할 공간이 연속되지 않아도 됨

배열 : 연속된 충분한 메모리 공간이 필요함

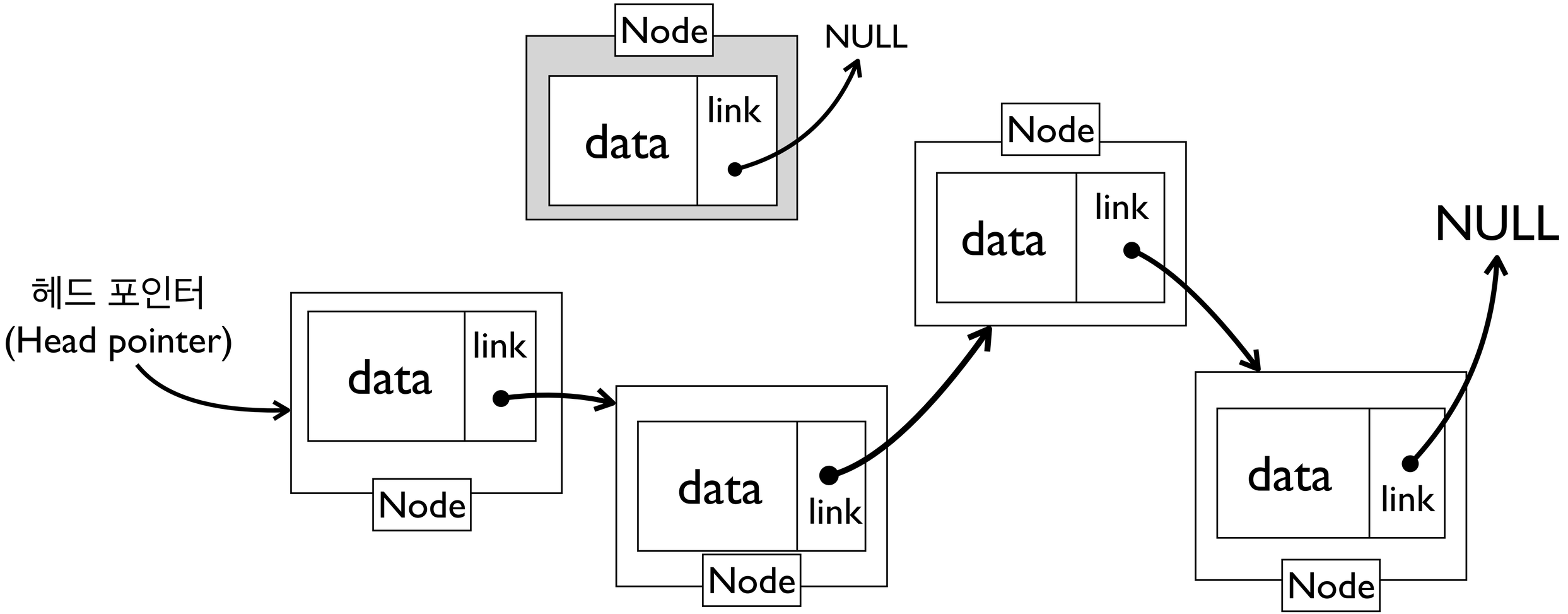
삽입 (insert)

- 첫 번째에 데이터를 삽입하는 경우



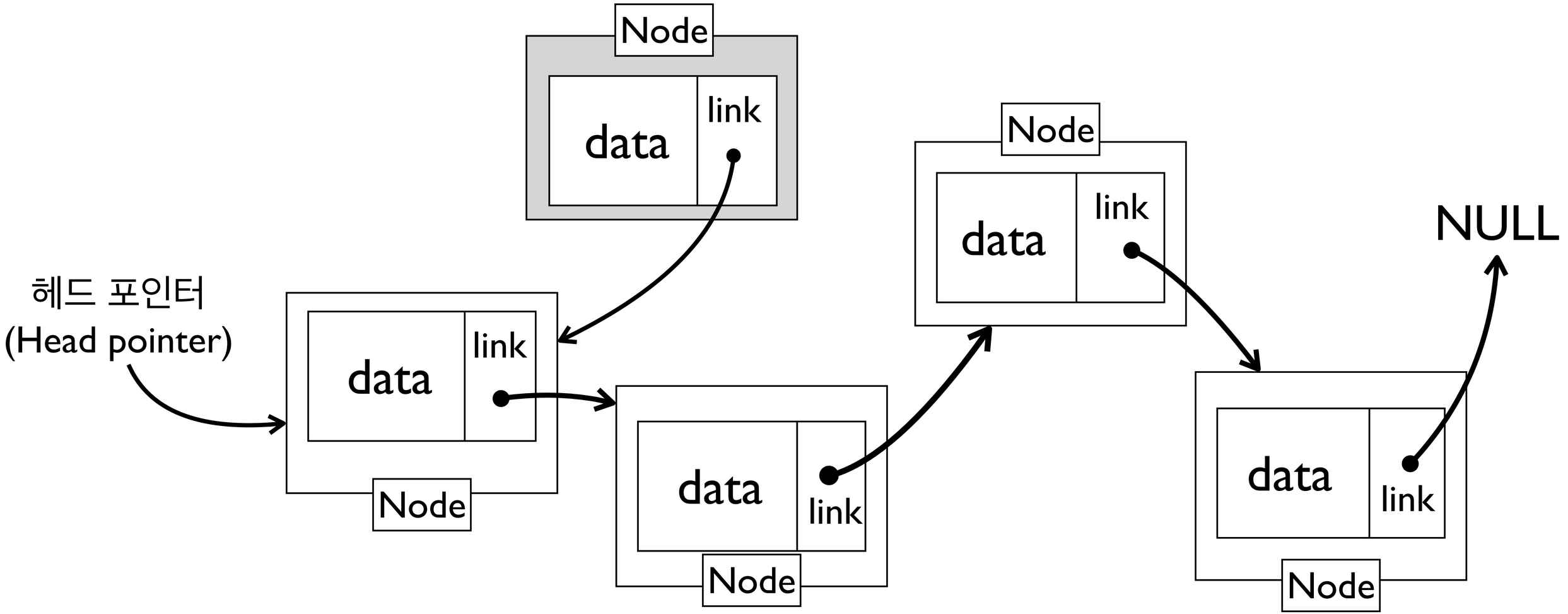
삽입 (insert)

- 첫 번째에 데이터를 삽입하는 경우



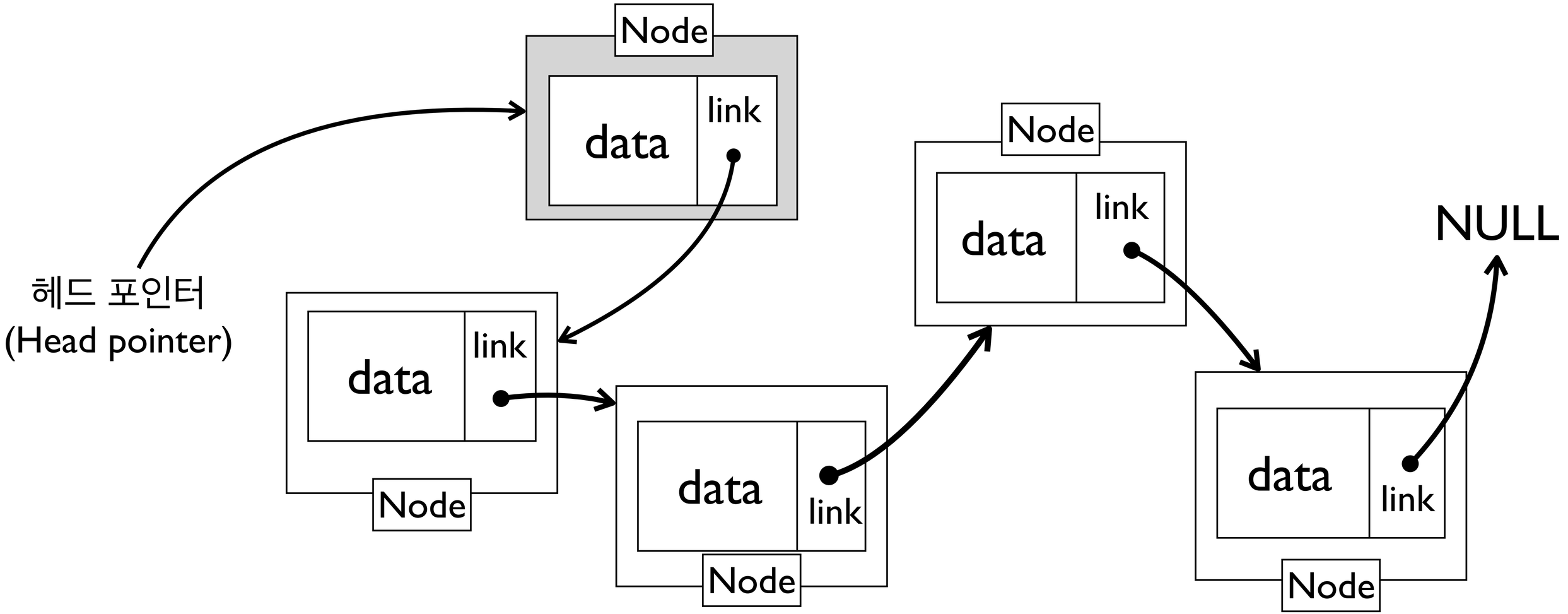
삽입 (insert)

- 첫 번째에 데이터를 삽입하는 경우



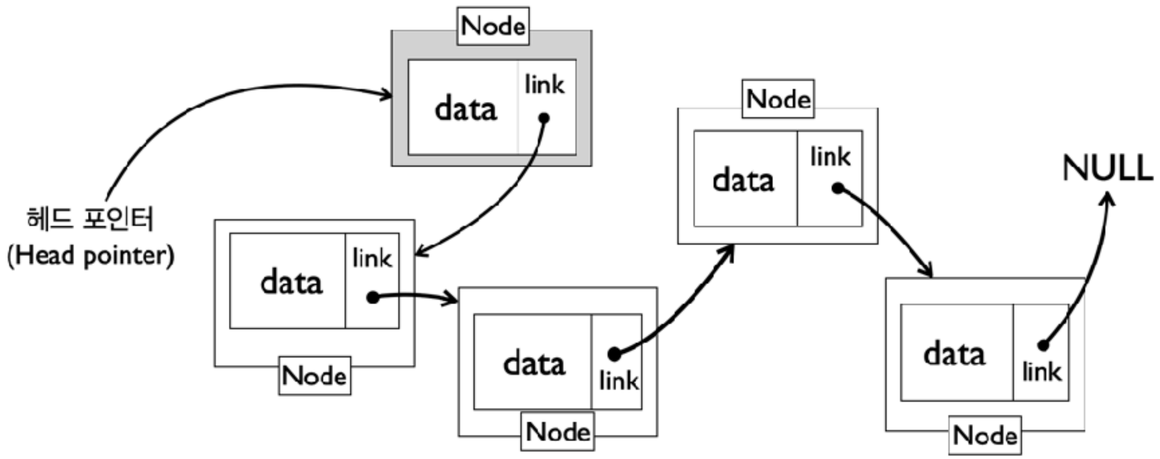
삽입 (insert)

- 첫 번째에 데이터를 삽입하는 경우

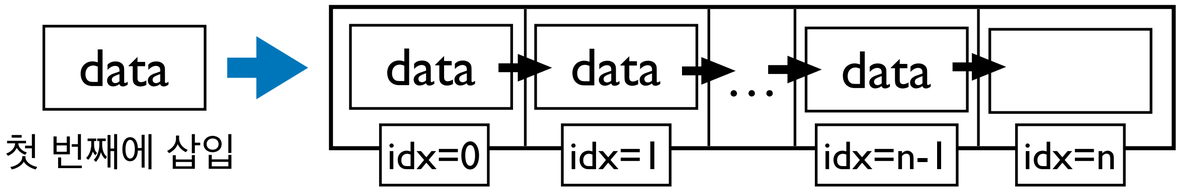


첫 번째에 데이터를 삽입하는 경우

연결 리스트 (Linked List)



배열 (Array)



첫 번째에 데이터를 삽입하는 경우

연결 리스트 (Linked List)

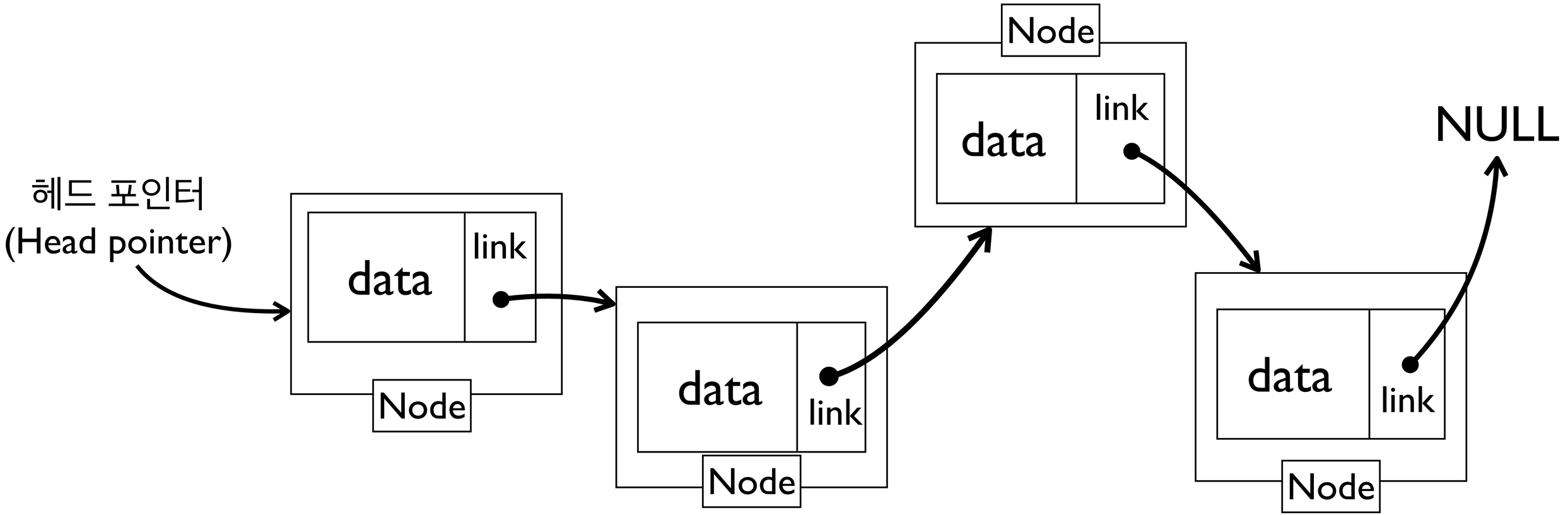
```c
#include <stdio.h>
#include <stdlib.h>

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

int main() {
    Node* head = NULL;
    Node* node1 = (Node*) malloc(sizeof(Node));
    Node* node2 = (Node*) malloc(sizeof(Node));
    node1->data = 1;
    node2->data = 2;
    node2->next = NULL;
    head = node1;
    node1->next = node2;
    printf("%d -> %d\n", head->data, head->next->data);

    Node* node3 = (Node*) malloc(sizeof(Node));
    node3 -> data = 3;
    node3 -> next = head;
    head = node3;
    printf("%d -> ", head->data);
    printf("%d -> ", head->next->data);
    printf("%d\n", head->next->next->data);
    return 0;
}
```

배열 (Array)

```c
#include <stdio.h>

int main() {
    int items[3];
    items[0] = 1;
    items[1] = 2;

    printf("%d -> %d\n", items[0], items[1]);

    for (int i = 2; i > 0; i--) {
        items[i] = items[i-1];
    }
    items[0] = 3;
    printf("%d -> ", items[0]);
    printf("%d -> ", items[1]);
    printf("%d\n", items[2]);
    return 0;
}
```

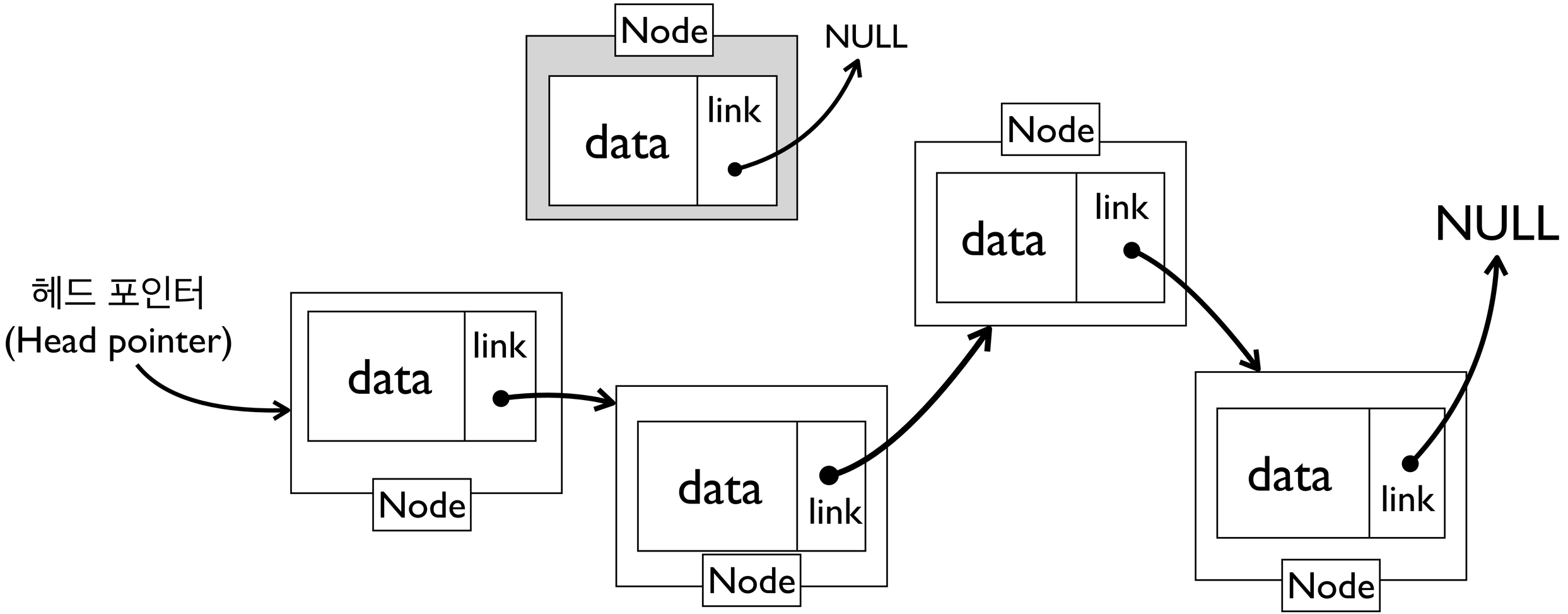
삽입 (insert)

- 연결 리스트 중간에 데이터를 삽입하는 경우



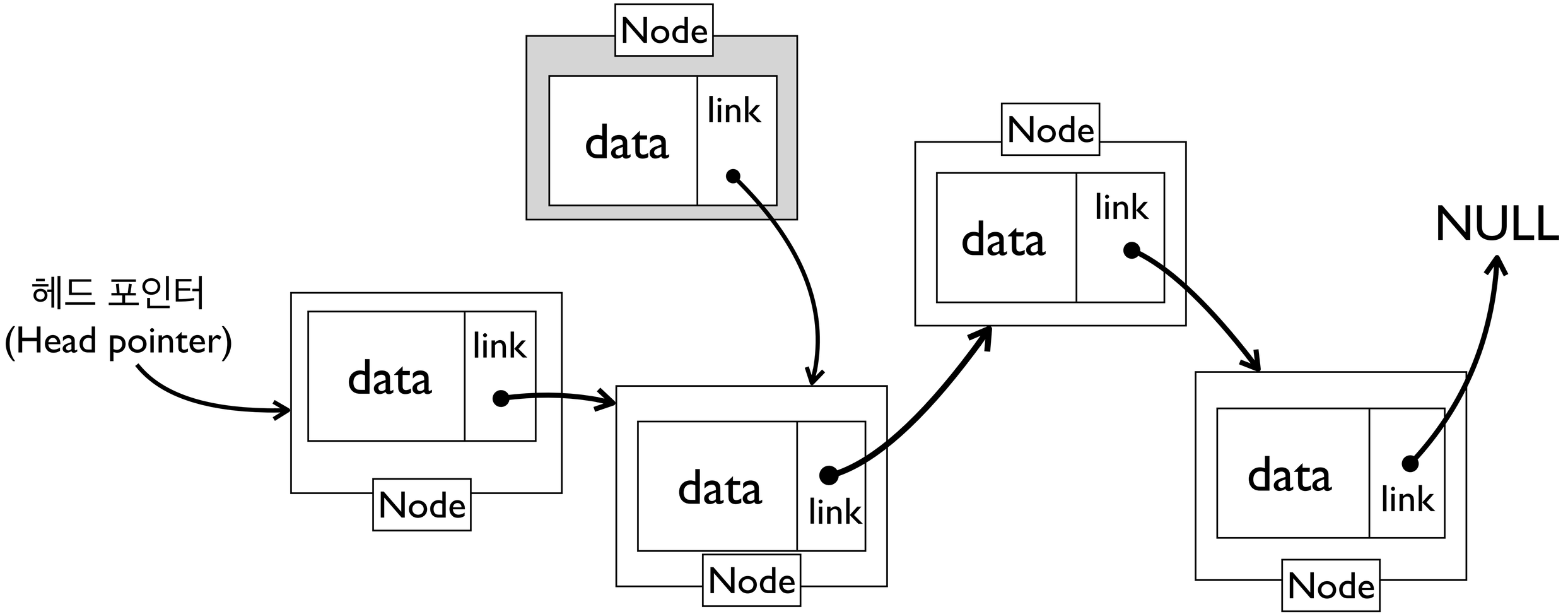
삽입 (insert)

- 연결 리스트 중간에 데이터를 삽입하는 경우



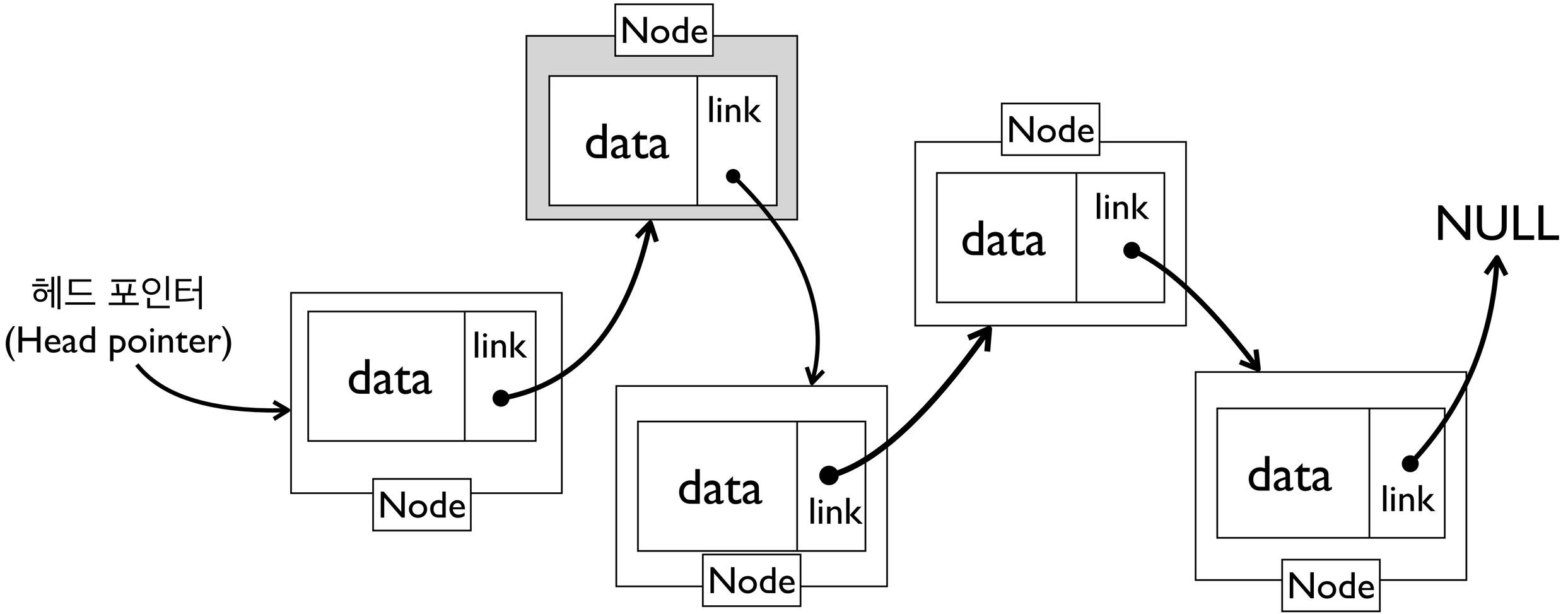
삽입 (insert)

- 연결 리스트 중간에 데이터를 삽입하는 경우



삽입 (insert)

- 연결 리스트 중간에 데이터를 삽입하는 경우

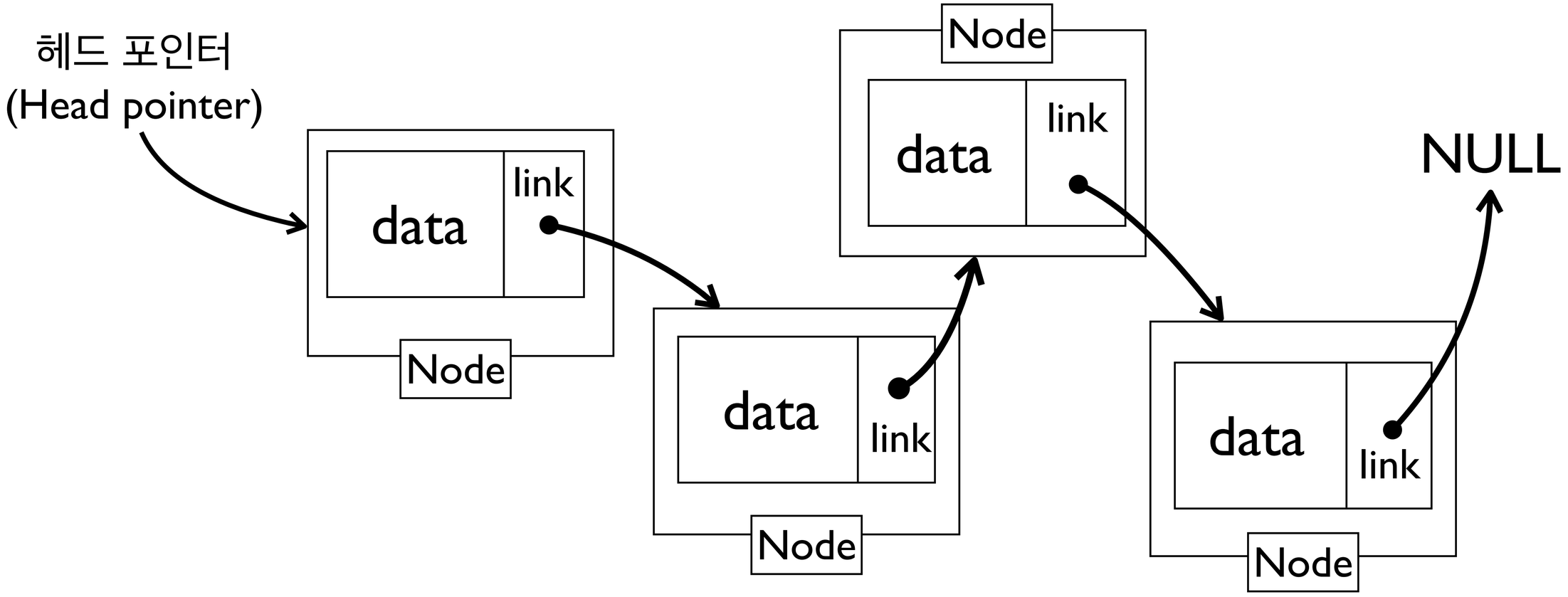


두 번째에 데이터를 삽입하는 경우

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

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

int main() {
    Node* head = NULL;
    Node* node1 = (Node*) malloc(sizeof(Node));
    Node* node2 = (Node*) malloc(sizeof(Node));
    node1->data = 1;
    node2->data = 2;
    head = node1;
    node1->next = node2;
    printf("%d -> %d\n", head->data, head->next->data);

    Node* node3 = (Node*) malloc(sizeof(Node));
    node3 -> data = 3;
    node3 -> next = head->next;
    head->next = node3;
    printf("%d -> ", head->data);
    printf("%d -> ", head->next->data);
    printf("%d\n", head->next->next->data);
    return 0;
}
```

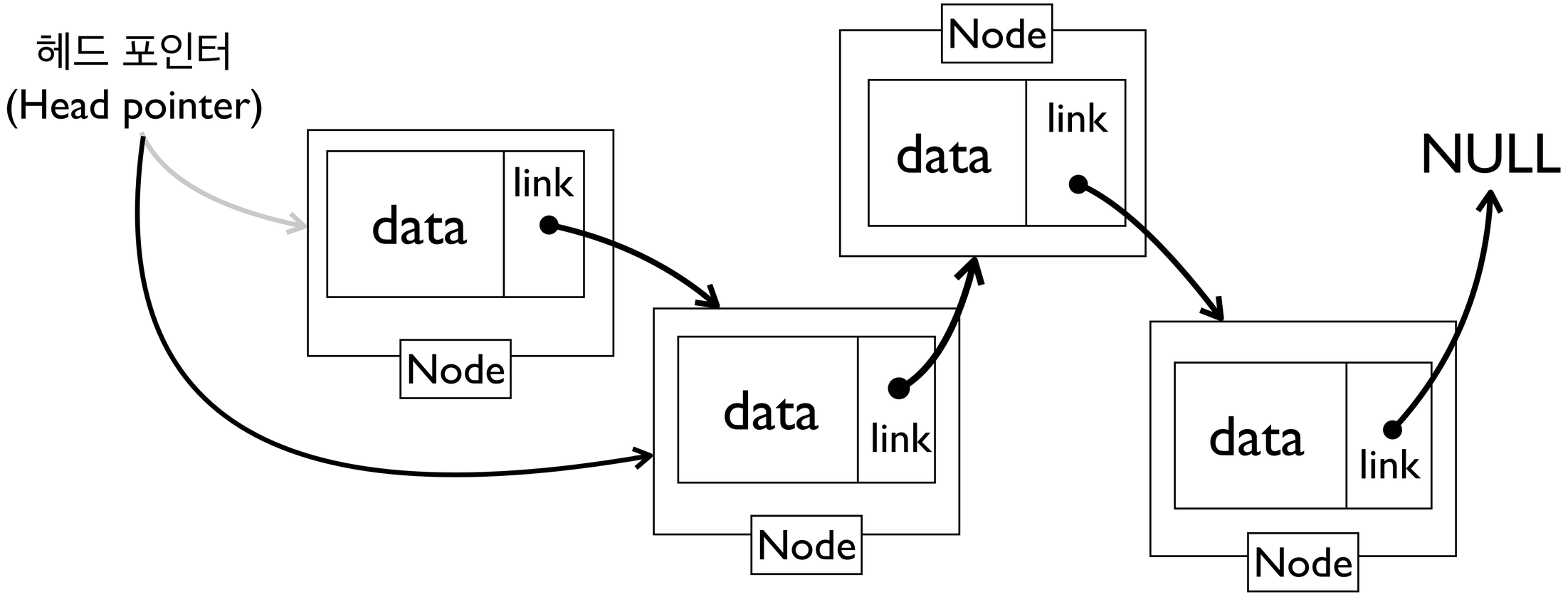
데이터 삭제 (delete)

- 첫 번째에 데이터를 삭제하는 경우



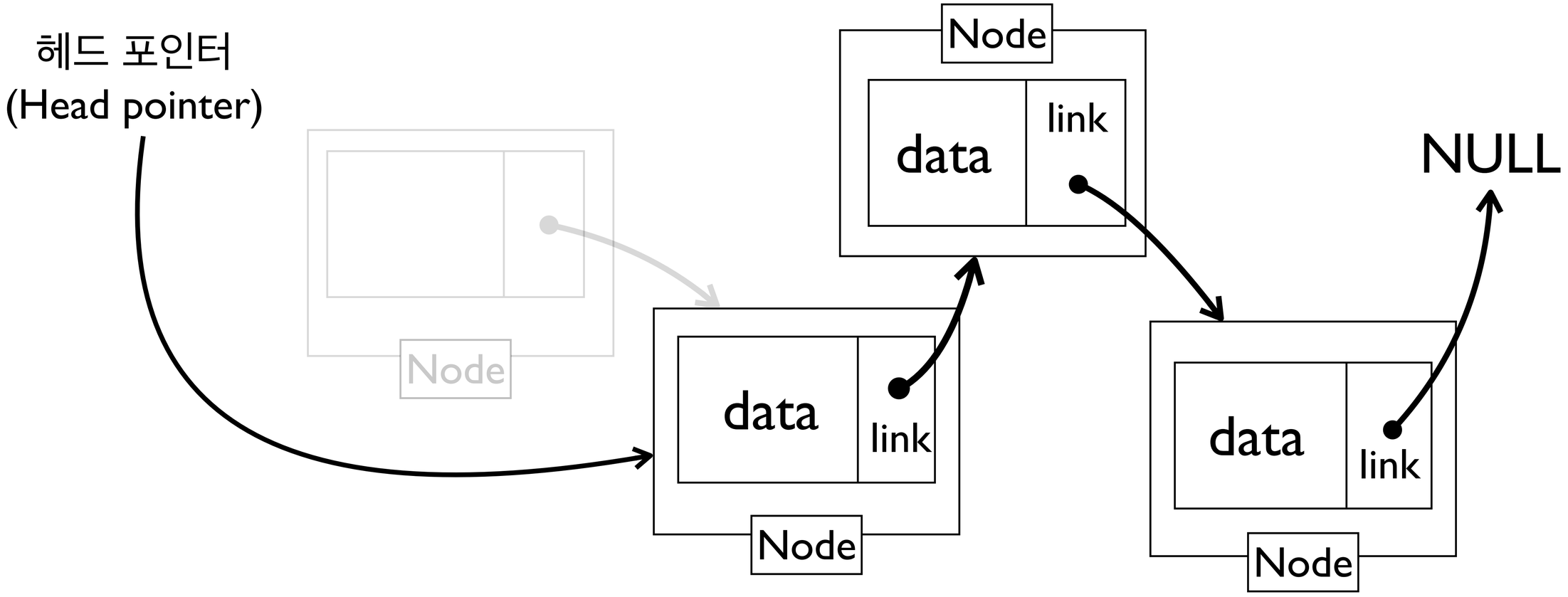
데이터 삭제 (delete)

- 첫 번째에 데이터를 삭제하는 경우



데이터 삭제 (delete)

- 첫 번째에 데이터를 삭제하는 경우



첫 번째에 데이터를 삭제하는 경우

연결 리스트 (Linked List)

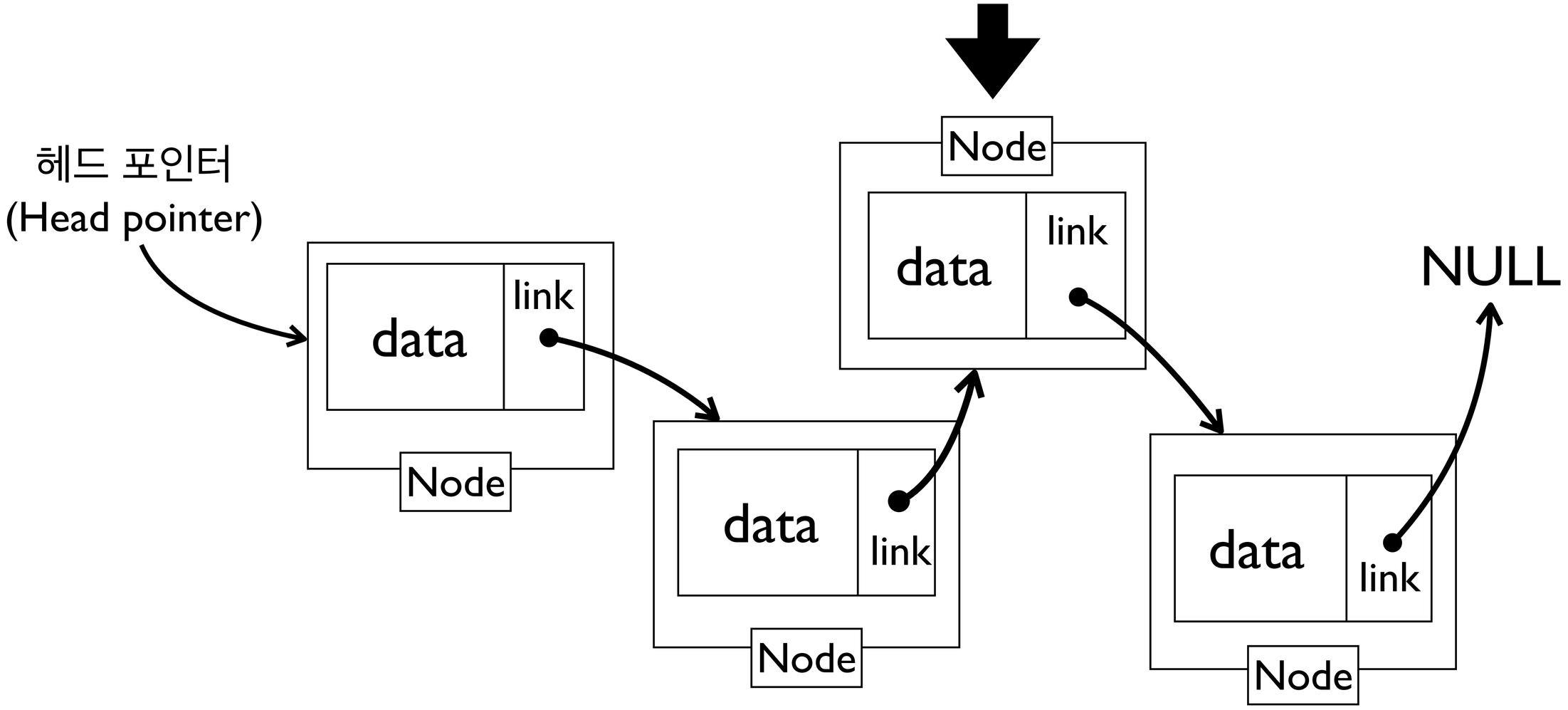
```
…
int main() {
    Node* head = NULL;
    Node* node1 = (Node*) malloc(sizeof(Node));
    Node* node2 = (Node*) malloc(sizeof(Node));
    Node* node3 = (Node*) malloc(sizeof(Node));

    node1->data = 1;
    node2->data = 2;
    node3->data = 3;

    head = node1;
    head->next = node2;
    head->next->next = node3;

    printf("%d -> ", head->data);
    printf("%d -> ", head->next->data);
    printf("%d\n", head->next->next->data);

    Node* temp = head;
    head = head->next;
    free(temp);
    temp = NULL;

    printf("%d -> ", head->data);
    printf("%d\n", head->next->data);
    return 0;
}
```

배열 (Array)

```
#include <stdio.h>

int main() {
    int items[3];
    items[0] = 1;
    items[1] = 2;
    items[2] = 3;

    printf("%d -> ", items[0]);
    printf("%d -> ", items[1]);
    printf("%d\n", items[2]);

    for (int i = 0; i < 2; i++) {
        items[i] = items[i+1];
    }
    printf("%d -> ", items[0]);
    printf("%d\n", items[1]);
    return 0;
}
```

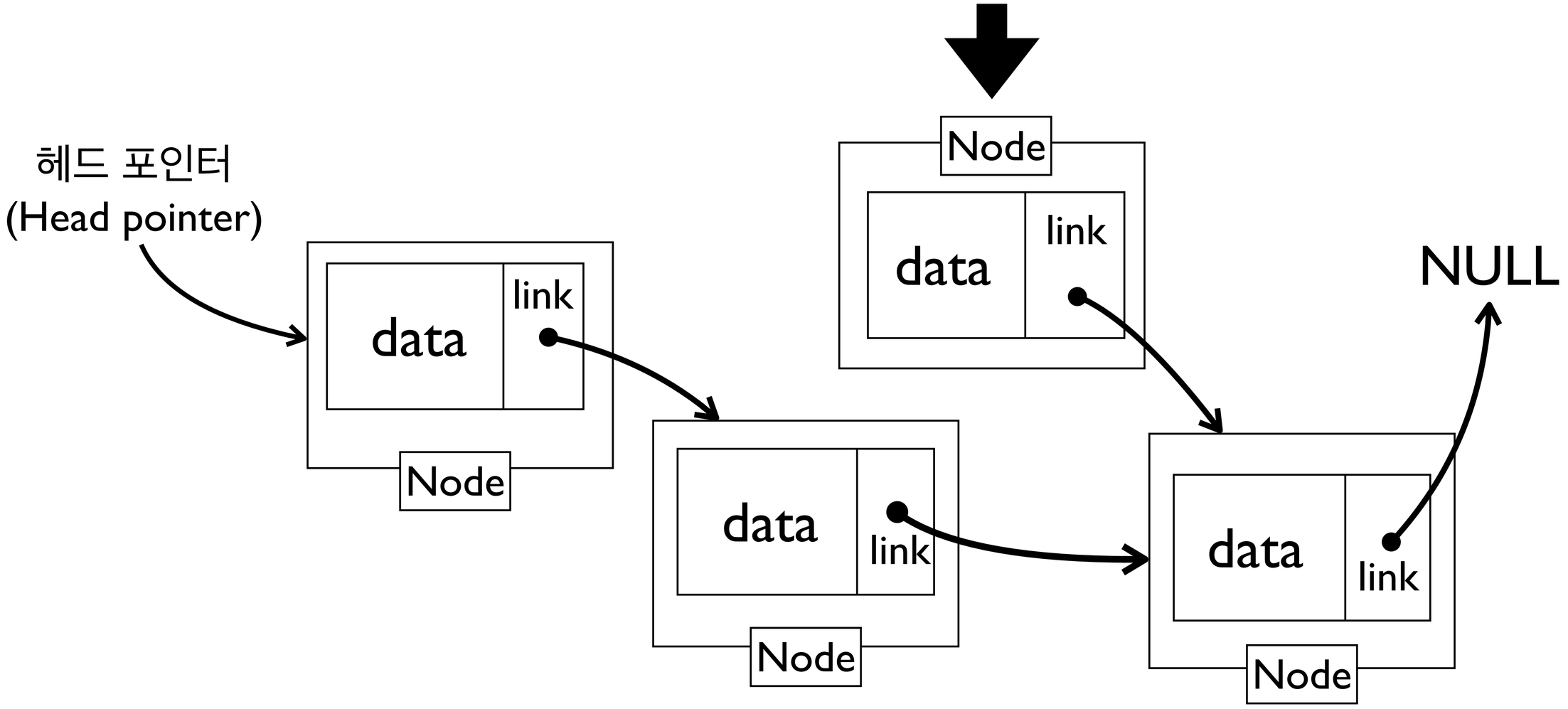
데이터 삭제 (delete)

- 연결 리스트 중간의 데이터를 삭제하는 경우



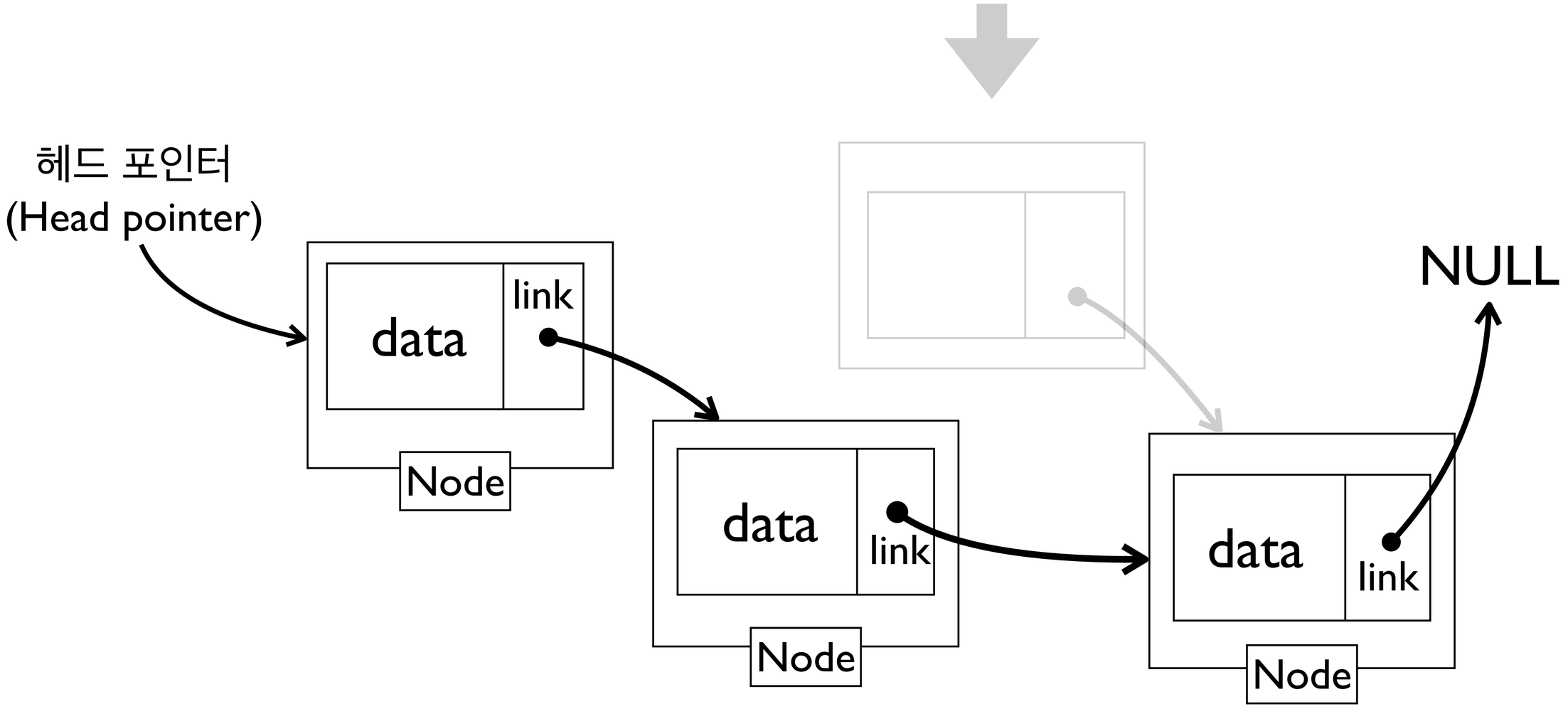
데이터 삭제 (delete)

- 연결 리스트 중간의 데이터를 삭제하는 경우



데이터 삭제 (delete)

- 연결 리스트 중간의 데이터를 삭제하는 경우



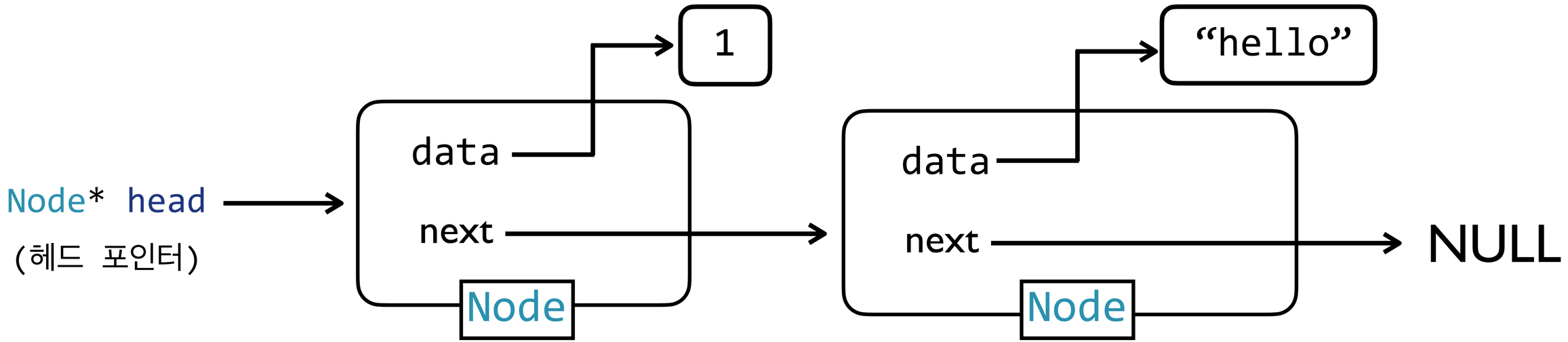
두 번째에 데이터를 삭제하는 경우

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

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

int main() {
    Node* head = NULL;
    Node* node1 = (Node*) malloc(sizeof(Node));
    Node* node2 = (Node*) malloc(sizeof(Node));
    Node* node3 = (Node*) malloc(sizeof(Node));

    node1->data = 1;
    node2->data = 2;
    node3->data = 3;

    head = node1;
    head->next = node2;
    head->next->next = node3;

    printf("%d -> ", head->data);
    printf("%d -> ", head->next->data);
    printf("%d\n", head->next->next->data);

    Node* temp = head->next;
    head->next = head->next->next;
    free(temp);
    temp = NULL;
    printf("%d -> ", head->data);
    printf("%d\n", head->next->data);
    return 0;
}
```

연결 리스트

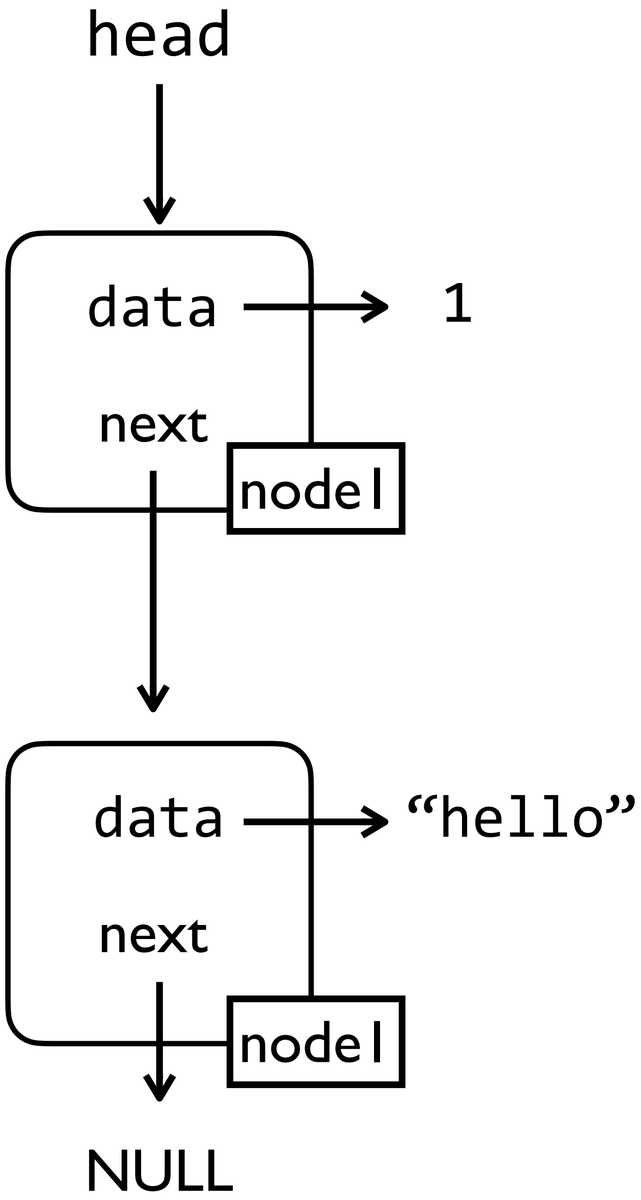
- 아래와 같이 구현할 경우 다양한 타입을 데이터로 가지는 자료구조를 구현할 수 있음

```
typedef struct Node {
    void *data;
    struct Node* next;
} Node;
```



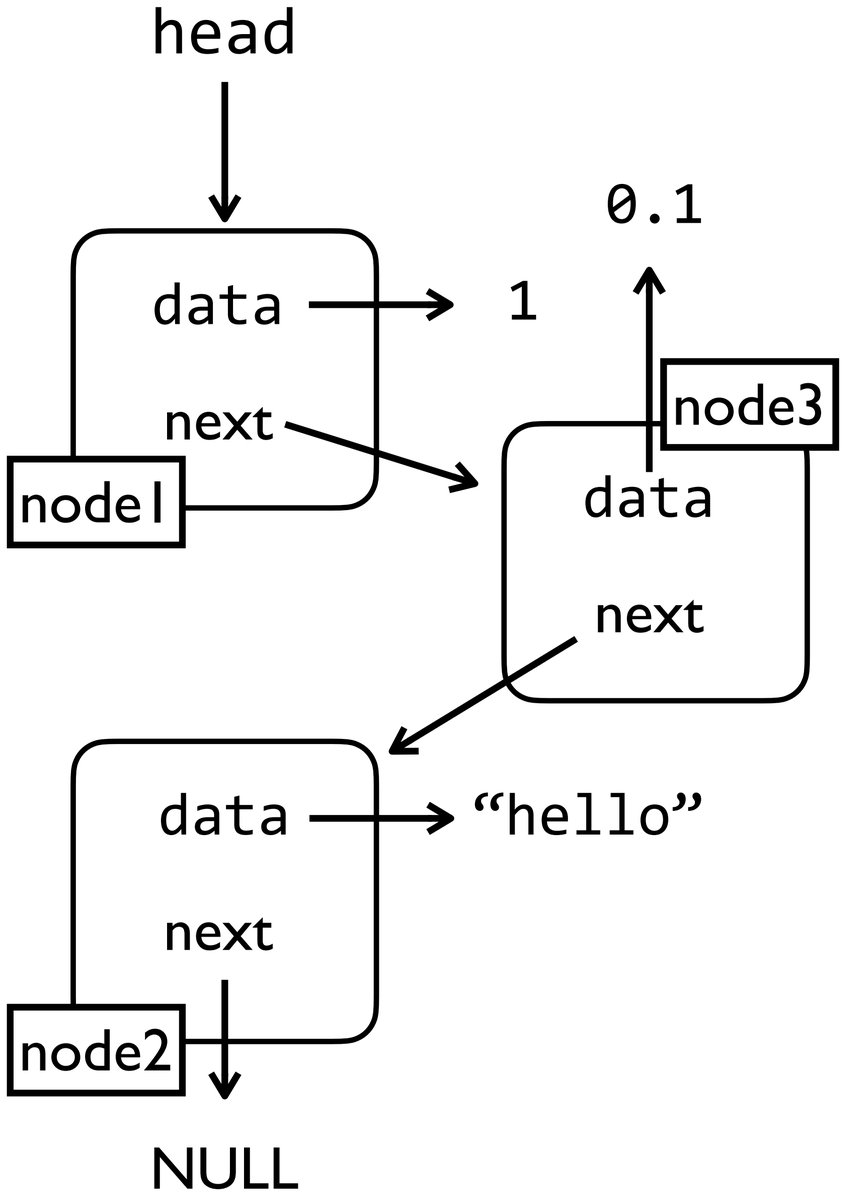
연결 리스트 사용 예시

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

typedef struct Node {
    void *data;
    struct Node* next;
} Node;

int main() {
    Node* head = NULL;

    int data1 = 1;
    Node* node1 = (Node*)malloc(sizeof(Node));
    node1 -> data = &data1;
    node1 -> next = NULL;
    head = node1;

    char data2[] = "hello";
    Node* node2 = (Node*)malloc(sizeof(Node));
    node1 -> next = node2;
    node2 -> data = data2;
    node2 -> next = NULL;

    printf("%d\n", *(int *)(head->data));
    printf("%s\n", (char *)(head->next->data));

    return 0;
}
```



연결 리스트 사용 예시

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

typedef struct Node {
    void *data;
    struct Node* next;
} Node;

int main() {
    Node* head = NULL;

    int data1 = 1;
    Node* node1 = (Node*)malloc(sizeof(Node));
    node1 -> data = &data1;
    node1 -> next = NULL;
    head = node1;

    char data2[] = "hello";
    Node* node2 = (Node*)malloc(sizeof(Node));
    node1 -> next = node2;
    node2 -> data = data2;
    node2 -> next = NULL;

    printf("%d\n", *(int *)(head->data));
    printf("%s\n", (char *)(head->next->data));

    ?

    printf("%f\n", *(double *)(head->next->data));

    return 0;
}
```



연결 리스트 사용 예시

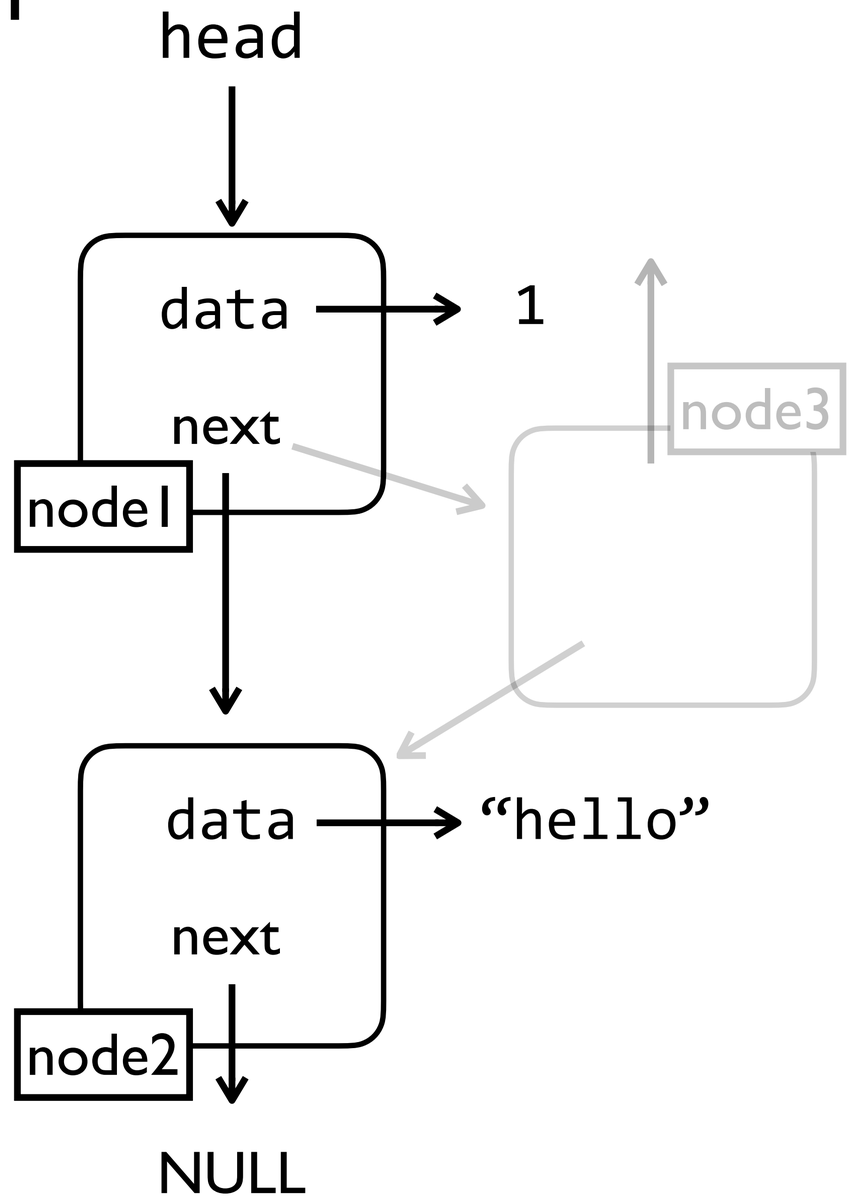
```
…
int main() {
    Node* head = NULL;

    int data1 = 1;
    Node* node1 = (Node*)malloc(sizeof(Node));
    node1 -> data = &data1;
    node1 -> next = NULL;
    head = node1;

    char data2[] = "hello";
    Node* node2 = (Node*)malloc(sizeof(Node));
    node1 -> next = node2;
    node2 -> data = data2;
    node2 -> next = NULL;

    printf("%d\n", *(int *)(head->data));
    printf("%s\n", (char *)(head->next->data));
```

?

```
   printf("%f\n", *(double *)(head->next->data));
```

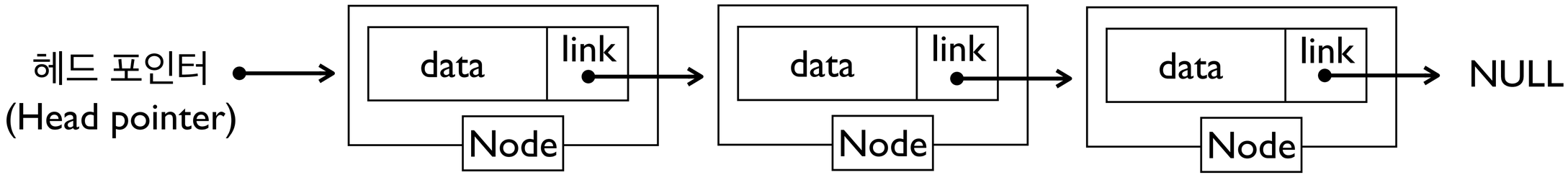
?

```
   printf("%s\n", (char *)(head->next->data));

    return 0;
}
```



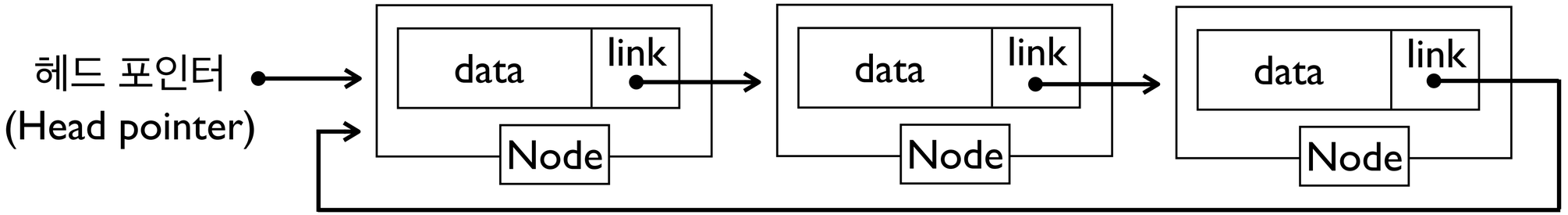
연결 리스트의 종류

(1) 단순 연결 리스트:



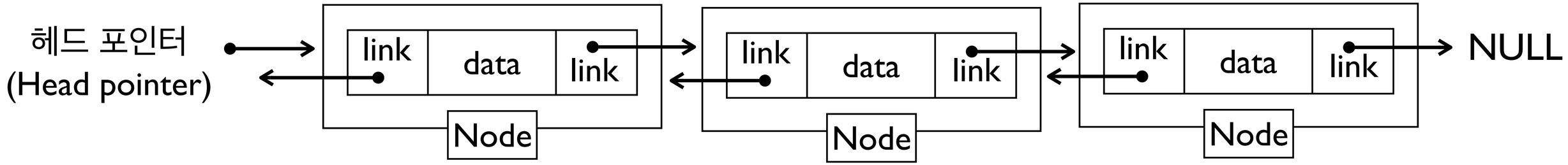
연결 리스트의 종류

(2) 원형 연결 리스트:



연결 리스트의 종류

(3) 이중 연결 리스트:



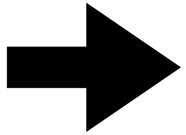
논의 (Discussion)

- 배열을 사용한 자료구조를 구현할 경우 단점
  - 동적으로 크기 조절이 불가능함

    (1) 큰 배열을 생성하고 그 중 일부만 사용한다면 메모리 공간이 낭비 됨

    (2) 자료구조의 용량(capacity)를 늘릴 때 많은 연산이 필요함

  - 모든 데이터가 동일한 타입을 가져야 함
  - 데이터 삽입과 삭제가 비효율적임
  - 연속된 (충분한) 메모리 공간이 필요함



연결 리스트를 사용하여 해결 가능!

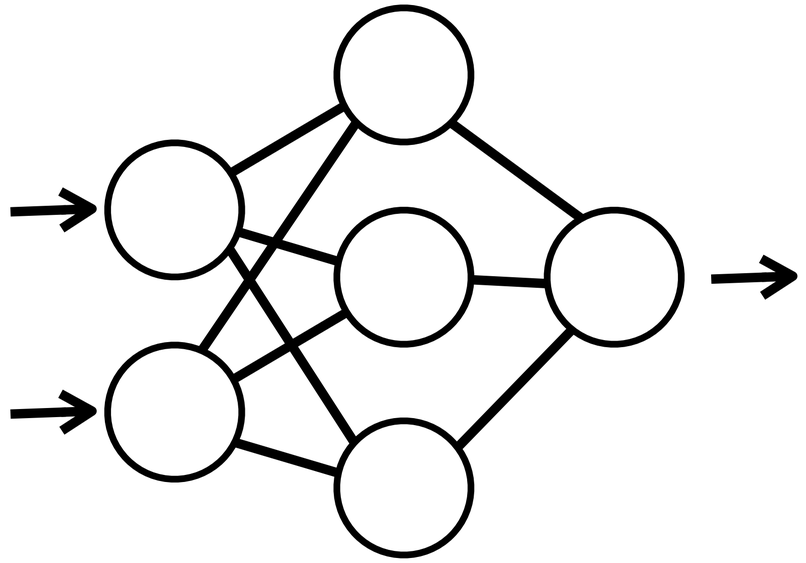
논의 (Discussion)

- 배열을 사용한 자료구조가 적합한 경우
  - (1) 동적으로 크기 조절이 필요 없고 (2) 모든 데이터가 동일한 타입을 가지며 (3) 삽입 삭제가 필요 없고 (4)공간이 충분할 경우
- 실제 예시: 기계학습 데이터

```
‘중간고사 점수': [34, 50, 45, 61, 75]
‘기말고사 점수’: [78, 80, 65, 56, 52]
    '합격여부': [ 0,  0,  0,  1,  1]
```

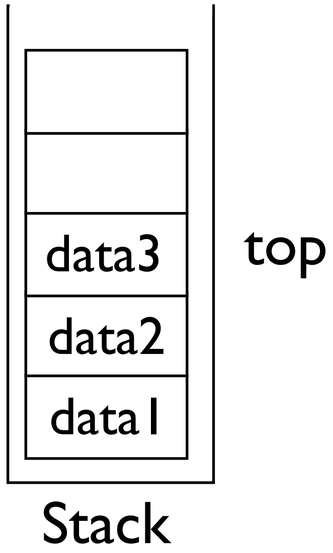
기계학습 데이터

기계학습

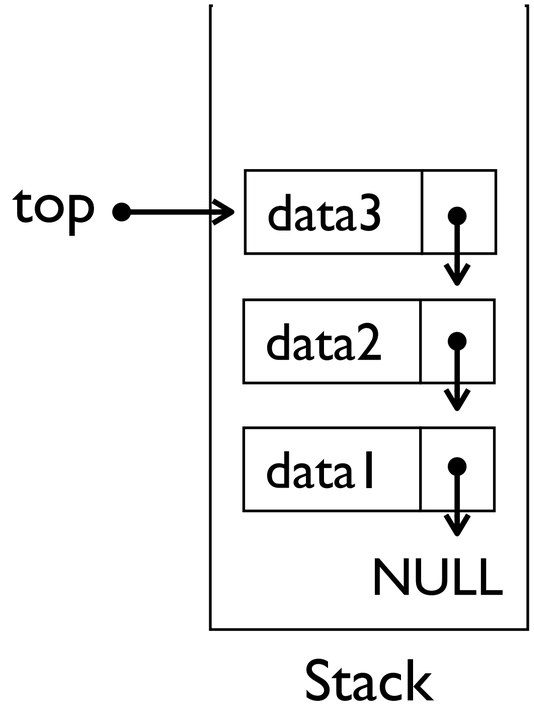


학습된 모델

배열과 연결리스트를 사용해 구현한 스택(Stack)



배열로 구현한 스택



연결 리스트로 구현한 스택

연결 리스트로 구현한 스택

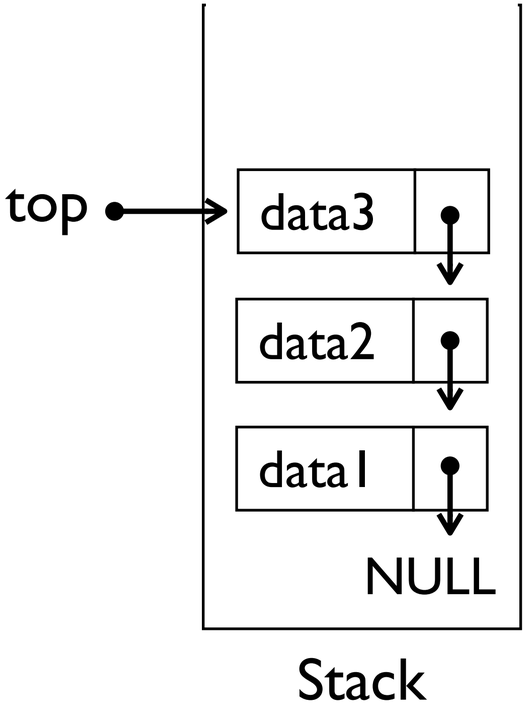
- 스택(Stack): 후입선출(LIFO: Last In, First Out) 원칙을 따르는 자료구조
- 스택의 추상 자료형:
  - create() : 비어있는 스택을 생성 후 반환
  - `isEmpty(s)` : 스택 s가 비어있는지 확인함
  - `push(s,x)` : 스택 s의 가장 위에 주어진 새로운 데이터 x를 추가
  - `pop(s)` : 스택 s의 가장 위에 있는 데이터를 삭제하고 반환
  - `peek(s)` : 스택 s의 가장 위 데이터를 제거하지 않고 반환
  - destroy(s) : 스택이 차지하고 있는 메모리를 해제함

연결 리스트로 구현한 스택

- 스택(Stack)은 다음과 같은 정보를 가지는 자료구조임

```
typedef struct Stack {
    Node* top;
} Stack;
```

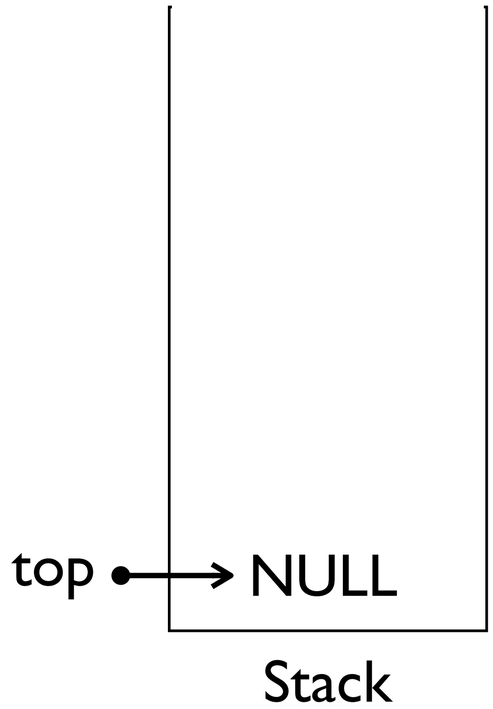
```
typedef struct Node {
    int data;
    struct Node* next;
} Node;
```



연결 리스트로 구현한 스택

- create : 비어있는 스택을 생성 후 반환

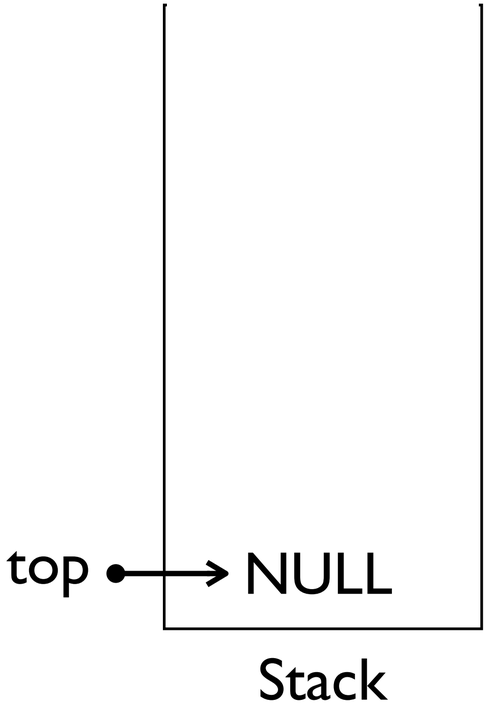
```
procedure create():
  stack ← allocateStack()
  stack.top ← NULL
  return stack
end procedure
```



연결 리스트로 구현한 스택

- `isEmpty` : 스택이 비어있는지 확인함

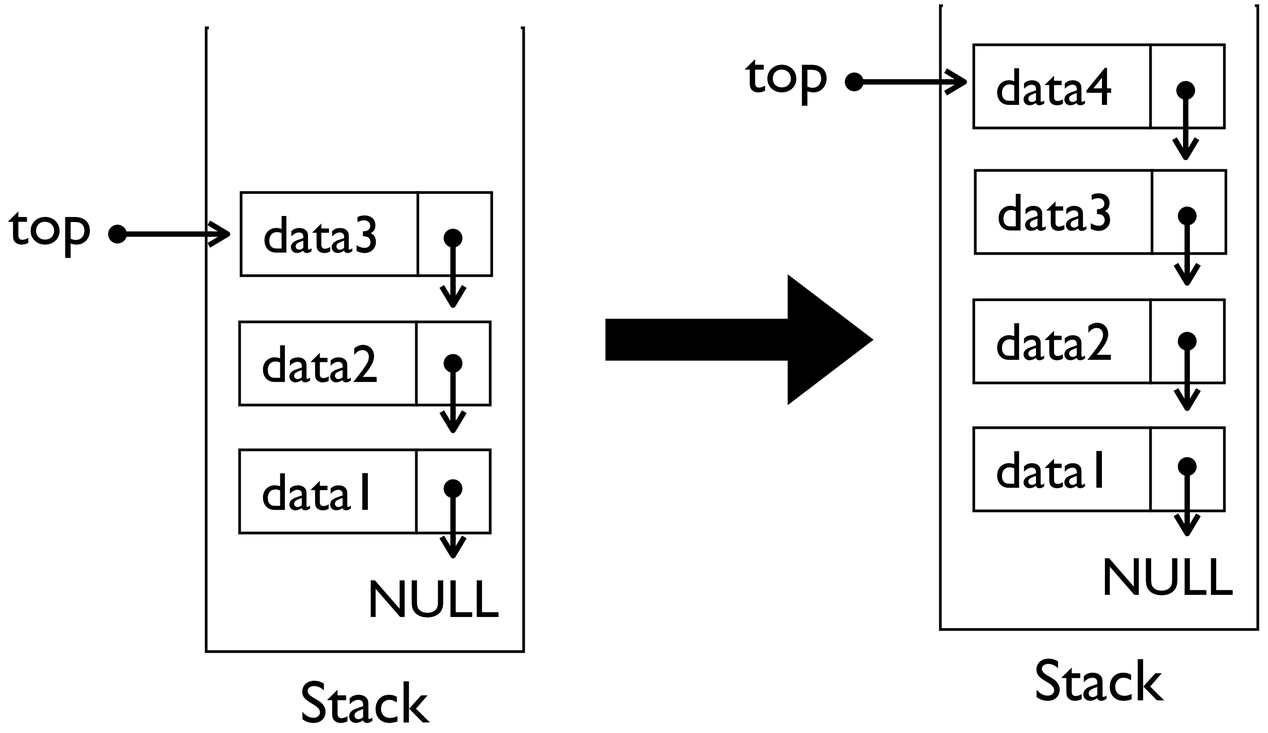
```
procedure isEmpty(stack)
    if stack.top = NULL then
        return true
    else
        return false
  end if
end procedure
```



연결 리스트로 구현한 스택

- push : 스택의 맨 위에 주어진 새로운 데이터를 추가

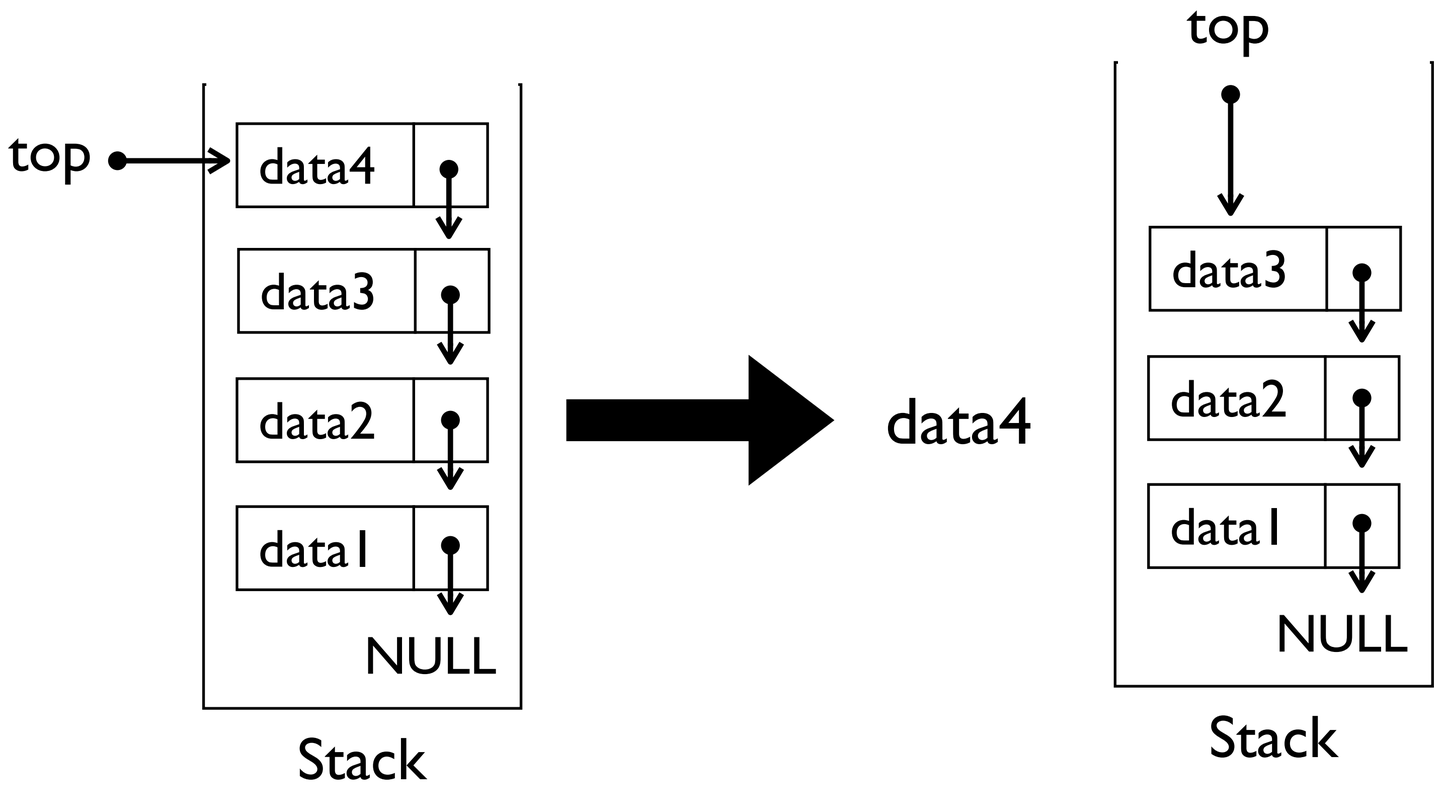
```
procedure push(stack, data)
    node ← allocateNode()
    node.data ← data
    node.next ← stack.top
    stack.top ← node
    return stack
end procedure
```



연결 리스트로 구현한 스택

- pop : 스택의 가장 위에 있는 데이터를 삭제하고 반환

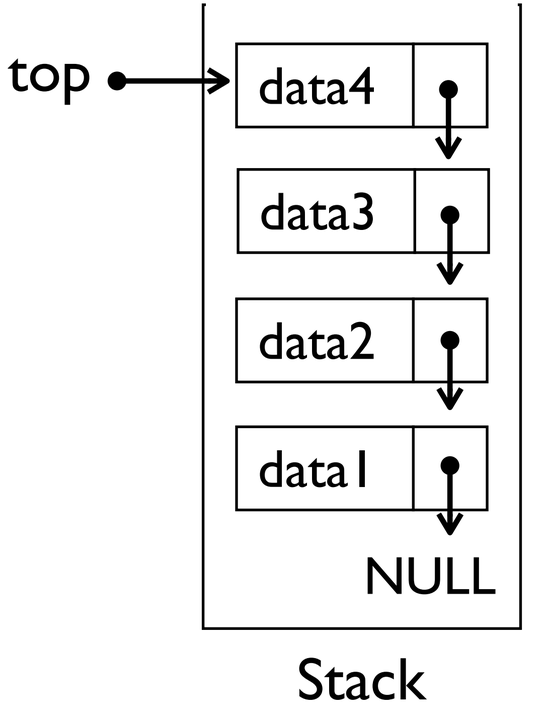
```
procedure pop(stack)
    if isEmpty(stack) then
        print (“Cannot pop. Stack is empty.”)
        return error()                          ▷ failed
    else
        node ← stack.top
        data ← node.data
        stack.top ← stack.top.next
        free(node)
        return data
end procedure
```



연결 리스트로 구현한 스택

- peek : 스택의 맨 위 데이터를 제거하지 않고 반환

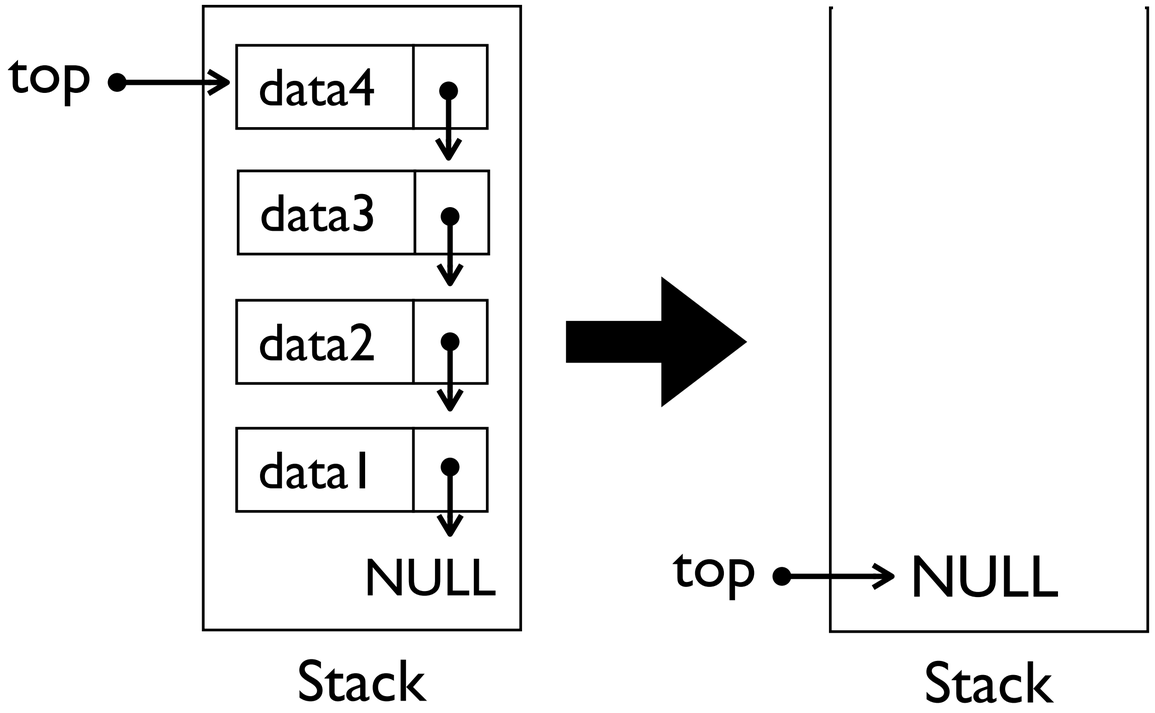
```
procedure peek(stack)
    if isEmpty(stack) then
        print (“Cannot peek. Stack is empty”)
        return error()                          ▷ failed
    else
        return stack.top.data
  end if
end procedure
```



연결 리스트로 구현한 스택

- destroy : 스택이 차지하고 있는 메모리를 해제함

```
procedure destroy(stack)
    while stack.top ≠ NULL do
        node ← stack.top
        stack.top ← stack.top.next
        free(node)
    end while
    free(stack)
    return true
end procedure
```



연결 리스트로 구현한 큐 (Queue)

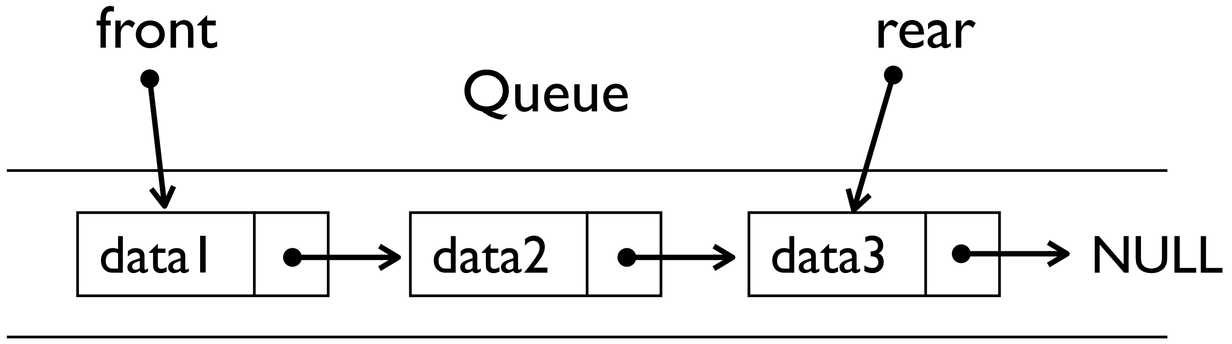
- 큐(Queue)는 선입선출(FIFO: Fist In, First Out) 원칙을 따르는 자료구조
- 큐의 추상 자료형:
  - create() : 비어있는 큐를 생성 후 반환
  - `isEmpty(q)` : 큐 q가 비어있으면 `true`를 아니면 `false`를 반환
  - `enqueue(q, e)` : 큐 q에서 주어진 데이터 e를 큐의 맨 뒤에 추가
  - `dequeue(q)` : 큐 q가 비어있지 않으면 맨 앞 데이터를 삭제하고 반환
  - `peek(q)` : 큐 q가 비어있지 않으면 맨 앞 데이터를 제거하지 않고 반환
  - `destroy(q)` : 큐가 차지하고 있는 메모리를 해제함

연결 리스트로 구현한 큐 (Queue)

- 큐(Queue)는 다음과 같은 정보를 가지는 자료구조임

```
typedef struct Queue {
    Node* front;
    Node* rear;
} Queue;
```

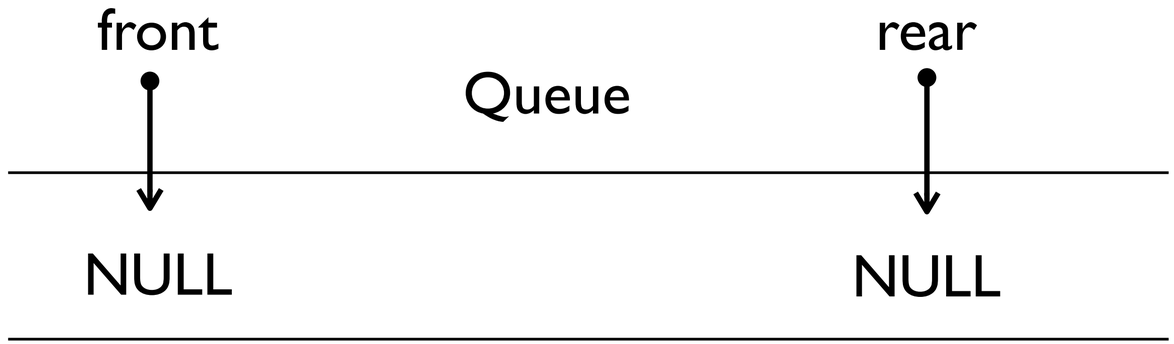
```
typedef struct Node {
    int data;
    struct Node* next;
} Node;
```



연결 리스트로 구현한 큐 (Queue)

- create : 비어있는 큐를 생성 후 반환

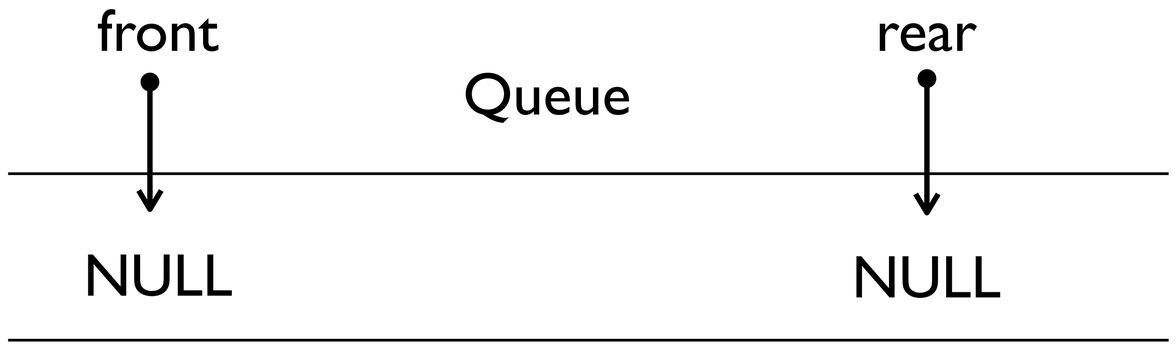
```
procedure create():
  queue ← allocateQueue()
  queue.front ← NULL
  queue.rear ← NULL
  return queue
end procedure
```



연결 리스트로 구현한 큐 (Queue)

- `isEmpty` : 큐가 비어있는지 확인함

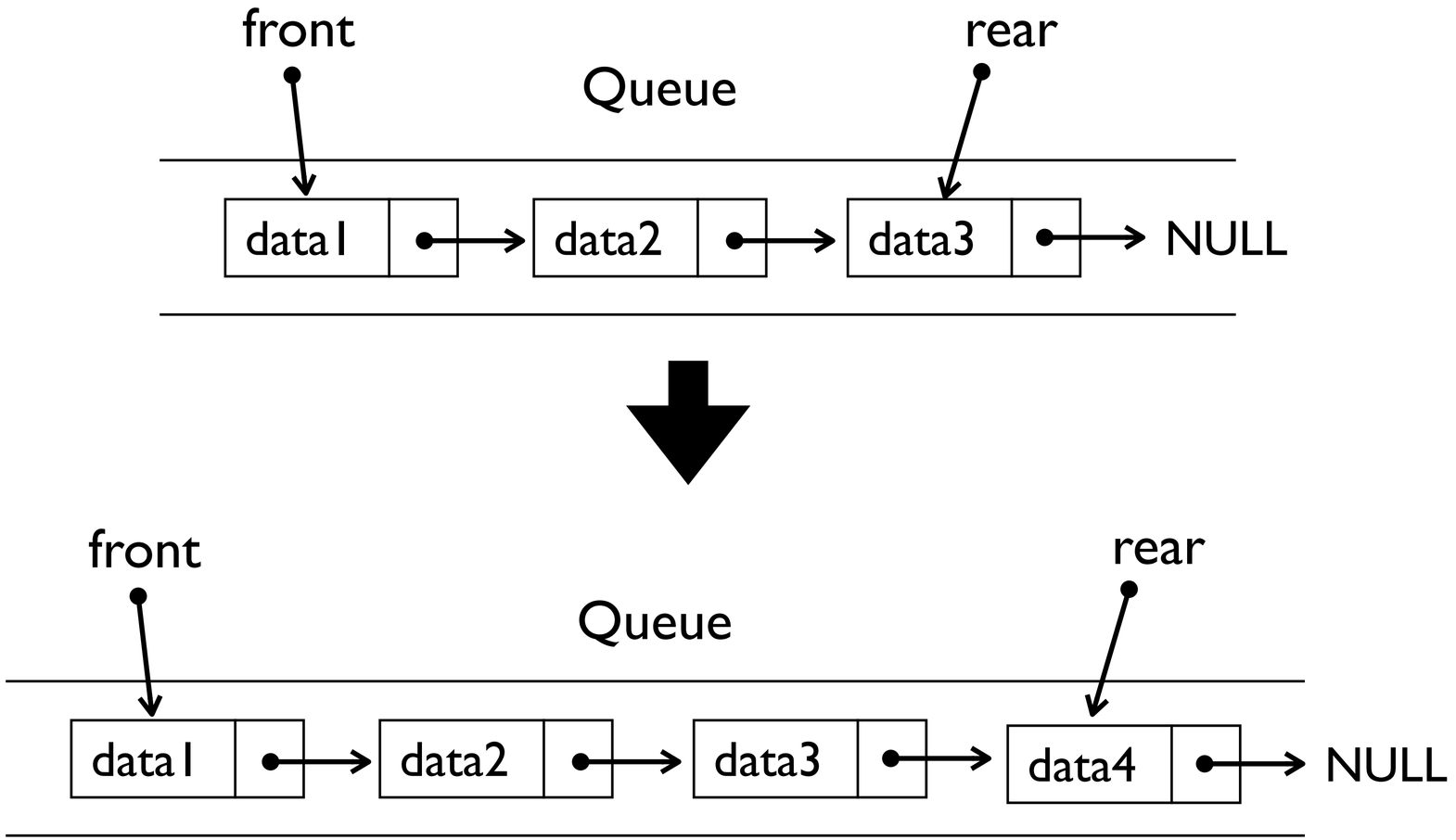
```
procedure isEmpty(queue)
    if queue.front = NULL then
        return true
    else
        return false
  end if
end procedure
```



연결 리스트로 구현한 큐 (Queue)

- enqueue: 큐에서 주어진 데이터를 맨 뒤에 추가

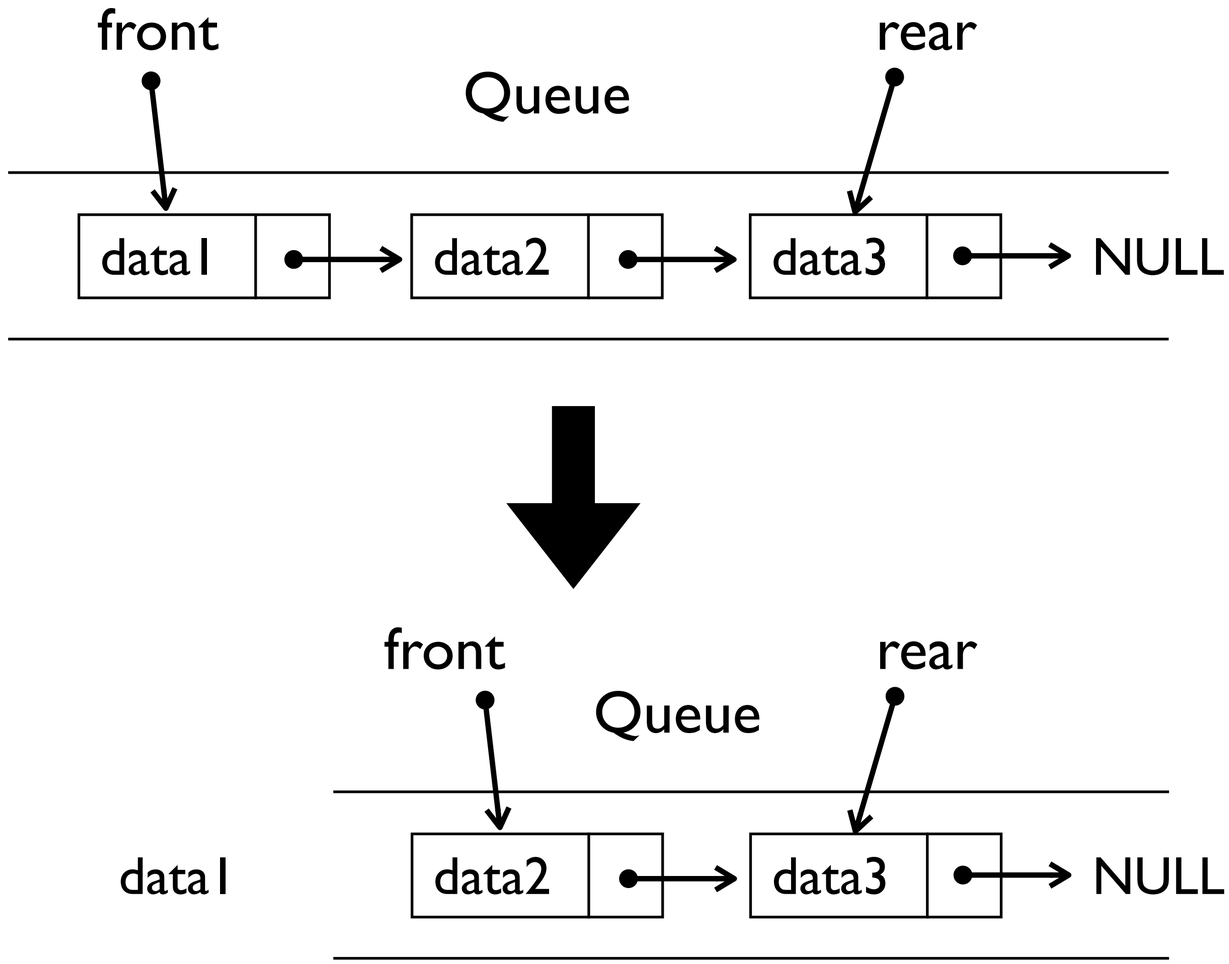
```
procedure enqueue(queue, data)
    node ← allocateNode()
    node.data ← data
    node.next ← NULL
    if isEmpty(queue) then
        queue.rear ← node
        queue.front ← node
    else
        queue.rear.next ← node
        queue.rear ← node
    end if
 return queue
end procedure
```



연결 리스트로 구현한 큐 (Queue)

- dequeue : 큐의 가장 앞에 있는 데이터를 삭제하고 반환

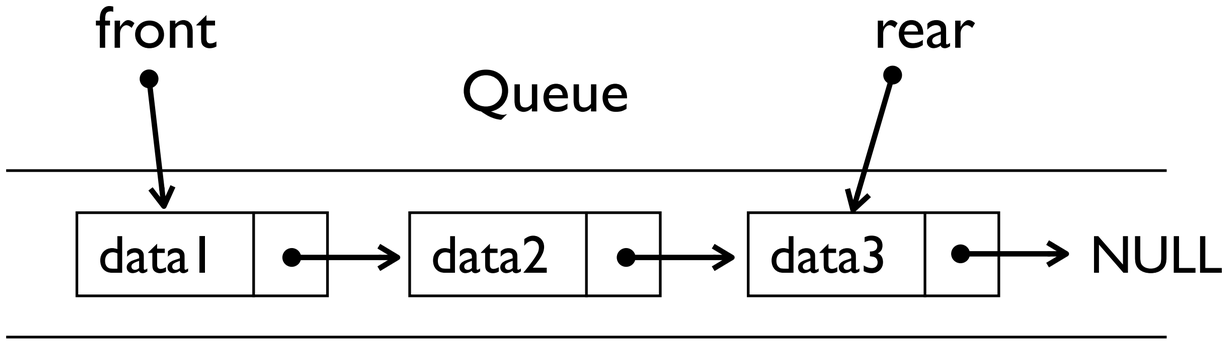
```
procedure dequeue(queue)
    if isEmpty(queue) then
        print("Cannot dequeue. Queue is empty.")
        return error()                                  ▷ failed
    end if
    node ← queue.front
    data ← node.data
    queue.front  ← queue.front.next
    if queue.front = NULL then
        queue.rear ← NULL
    free(node)
    end if
    return data                                         ▷ succeed
end procedure
```



연결 리스트로 구현한 큐 (Queue)

- peek : 큐의 가장 앞에 있는 데이터를 삭제하고 반환

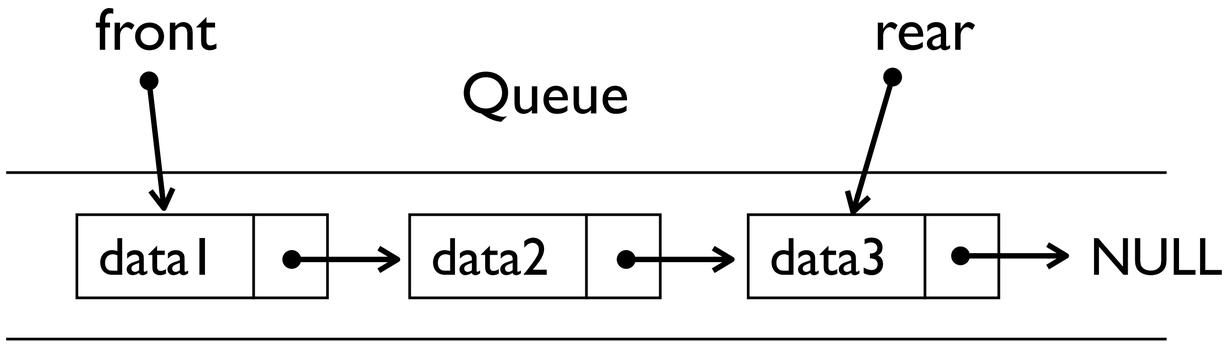
```
procedure peek(queue)
    if isEmpty(queue) then
        print("Cannot peek. Queue is empty.")
        return error()                          ▷ failed
    end if
    return queue.front.data.                    ▷ succeed
end procedure
```



연결 리스트로 구현한 큐 (Queue)

- `destroy` : 큐가 차지하고 있는 메모리를 해제함

```
procedure destroy(queue)
    while queue.front ≠ NULL do
        node ← queue.front
        queue.front ← queue.front.next
        free(node)
    end while
    free(queue)
return true                          ▷ succeed
end procedure
```



마무리 (Wrap-up)

- 문제: 배열(Array)로 자료구조를 구현할 시 아래와 같은 문제점이 있음
  - 동적으로 크기 조절이 불가능함
  - 모든 데이터가 동일한 타입을 가져야 함
  - 데이터 삽입과 삭제가 비효율적임
  - 연속된 (충분한) 메모리 공간이 필요함
- 해결책: 연결 리스트 (Linked List)
  - 연결 리스트는 데이터를 저장하고 있는 노드(Node)들의 집합이며 서로 연결되어 있음