

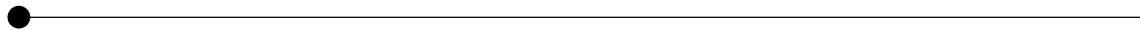
# 자료구조 – 정렬(Sort) Linked list



홍정완

# 목 차

A horizontal line with a large open circle at the left end, a small solid dot in the middle, and another large open circle at the right end.

1. 정렬의 개념
  2. 버블 정렬
  3. 삽입 정렬
  4. 합병 정렬
  5. 퀵 정렬
- 
- A horizontal line starting with a small solid dot on the left and extending to the right edge of the slide.

# 1. 정렬이란?

- 정렬은 대상을 특정한 규칙에 따라 정렬하는 것
- 정렬은 자료 탐색에 필수적이다.  
예를 들어, 영어사전에서 우리가 단어를 찾을 때, 단어들이 알파벳 순으로 정렬되어 있지 않다면???



오름차순

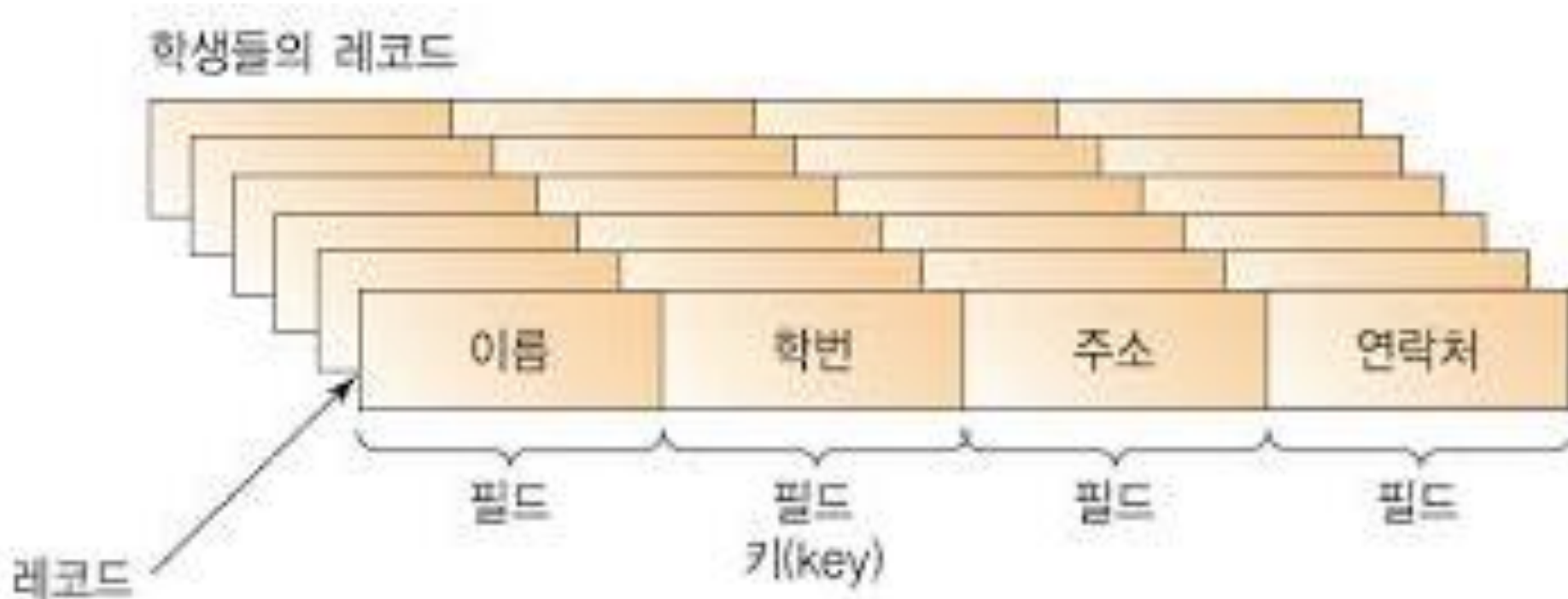


내림차순



# 1. 정렬의 대상

- 일반적으로 정렬시켜야 될 대상 : 레코드(record)
- 레코드는 다시 필드(field)라는 보다 작은 단위로 구성



# 1. 정렬 알고리즘 개요

---

- 모든 경우에 최적인 정렬 알고리즘은 없음
- 각 응용 분야에 적합한 정렬 방법을 사용해야 한다.
  - > 레코드 수의 많고 적음
  - > 레코드 크기의 크고 작음
  - > key의 특성 (문자, 정수, 실수 등)
  - > 메모리 내부/외부 정렬
- 정렬 알고리즘의 효율성 평가 기준
  - > 비교 연산 횟수
  - > 이동 연산 횟수

# 1. 정렬 알고리즘 개요

---

- 단순하지만 비효율적인 방법
  - > 삽입 정렬, 선택 정렬, 버블 정렬 등
- 복잡하지만 효율적인 방법
  - > 퀵 정렬, 힙 정렬, 합병 정렬, 기수 정렬 등
- 내부 정렬(internal sorting)
  - > 모든 데이터가 주기억장치에 저장되어진 상태에서 정렬
- 외부 정렬(external sorting)
  - > 외부기억장치에 대부분의 데이터가 있고 일부만 주기억장치에 저장된 상태에서 정렬

## 2. 버블 정렬 (bubble sort)



초기 상태



5와 3을 교환



교환 없음



8과 1을 교환



8과 2를 교환



8과 7을 교환



가장 큰 값 정렬

■ 정의 :

인접한 2개의 레코드를  
비교하여 크기가 순서대로  
되어 있지 않으면 서로 교환하는  
정렬

## 2. 버블 정렬 (bubble sort)

초기 상태	5	3	8	1	2	7
1차 스캔	3	5	1	2	7	8
2차 스캔	3	1	2	5	7	8
3차 스캔	1	2	3	5	7	8
4차 스캔	1	2	3	5	7	8
5차 스캔	1	2	3	5	7	8
정렬 완료	1	2	3	5	7	8

- 우측의 정렬이 완료된 레코드를 제외한

리스트의 왼쪽에 대하여 정렬이 완료될 때까지 반복(Loop).



## 2. 버블 정렬 - 구현 (selection)

```
int arr[] = { 5, 3, 8, 1, 2, 7 }; ①
int list_size, i;

/* start with empty linked list */
struct Node* start = NULL;

/* Create linked list from the array arr[].
Created linked list will be 1->11->2->56->12 */
for (i = 0; i < 6; i++)
    insertAtTheBegin(&start, arr[i]); ②

/* print list before sorting */
printf("\n Linked list before sorting ");
printList(start);

/* sort the linked list */
bubbleSort(start); ③

/* print list after sorting */
printf("\n Linked list after sorting ");
printList(start);
```

① 정렬 대상 레코드들

② 포인터 구조체 노드에  
레코드들 입력

③ 버블 정렬 수행

④ 데이터와 주소 구조체

```
/* 노드 정의 */ ④
typedef struct Node
{
    int data;
    struct Node* next;
};
```

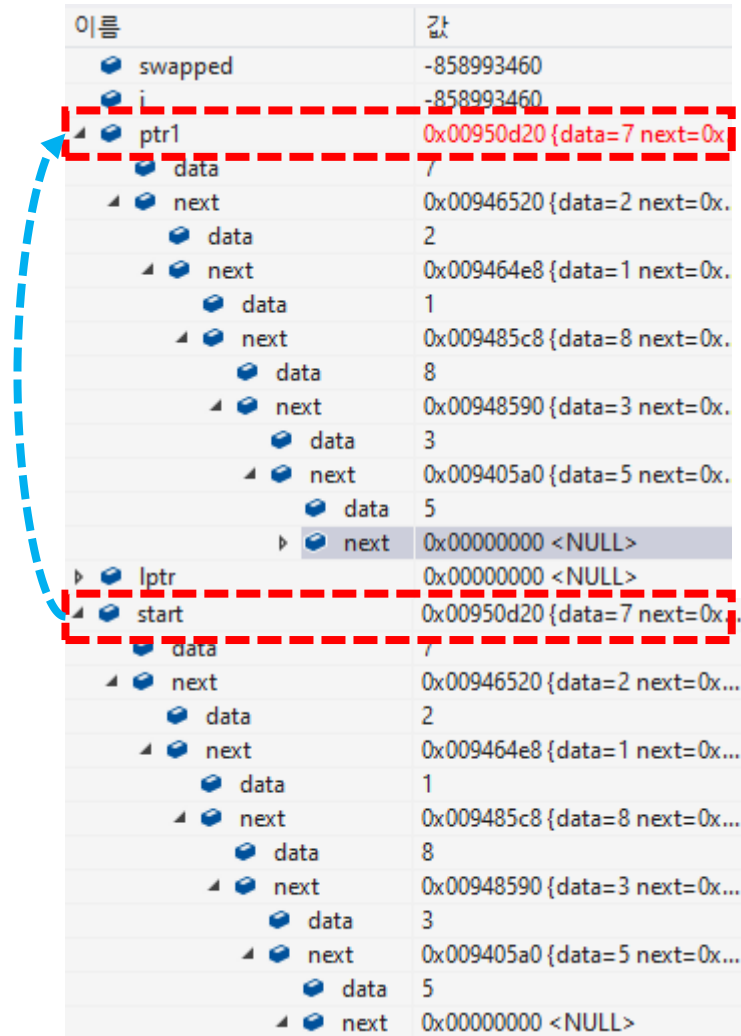
## 2. 버블 정렬 - 구현 (selection)

```
/* 버블정렬 함수 */
void bubbleSort(struct Node* start)
{
    int swapped, i;
    ① struct Node* ptr1 = NULL;
    struct Node* lptr = NULL;
    ② ptr1 = start;

    /* 리스트가 비었는지 확인 */
    if (ptr1 == NULL)
        return;
}
```

① 버블 정렬 변수 선언

② 포인터 구조체 노드에  
매개변수로 받은 노드를 복사



이름	값
swapped	-858993460
i	-858993460
ptr1	0x00950d20 {data=7 next=0x...}
data	7
next	0x00946520 {data=2 next=0x...}
data	2
next	0x009464e8 {data=1 next=0x...}
data	1
next	0x009485c8 {data=8 next=0x...}
data	8
next	0x00948590 {data=3 next=0x...}
data	3
next	0x009405a0 {data=5 next=0x...}
data	5
next	0x00000000 <NULL>
lptr	0x00000000 <NULL>
start	0x00950d20 {data=7 next=0x...}
data	7
next	0x00946520 {data=2 next=0x...}
data	2
next	0x009464e8 {data=1 next=0x...}
data	1
next	0x009485c8 {data=8 next=0x...}
data	8
next	0x00948590 {data=3 next=0x...}
data	3
next	0x009405a0 {data=5 next=0x...}
data	5
next	0x00000000 <NULL>

## 2. 버블 정렬 - 구현 (selection)

```
/* 버블정렬 함수 */
void bubbleSort(struct Node* start)
{
    int swapped, i;
    struct Node* ptr1 = NULL;
    struct Node* lptr = NULL;
    ptr1 = start;

    /* 리스트가 비었는지 확인 */
    if (ptr1 == NULL) 경과시간1ms 이하
        return;

    do
    {
        swapped = 0;
        ptr1 = start;

        ① while (ptr1->next != lptr)
        {
            ② if (ptr1->data > ptr1->next->data)
            {
                ③ swap(ptr1, ptr1->next);
                swapped = 1;
            }

            ④ ptr1 = ptr1->next;
        }

        ⑤ lptr = ptr1;
    } while (swapped);
}
```

- ① 버블 정렬 변수 선언
- ② 링크드 리스트의 노드의 데이터와 다음 노드의 데이터를 비교
- ③ 조건이 맞으면 레코드의 데이터 교환
- ④ 탐색을 위해 이동
- ⑤ 정렬된 데이터는 새로운 구조체 위치로

## 2. 버블 정렬 - 구현 (selection)

```
/* 버블정렬 함수 */
void bubbleSort(struct Node* start)
{
    int swapped, i;
    struct Node* ptr1 = NULL;
    struct Node* lptr = NULL;
    ptr1 = start;

    /* 리스트가 비었는지 확인 */
    if (ptr1 == NULL) 경과시간 1ms 이하
        return;

    do
    {
        swapped = 0;
        ptr1 = start;

        while (ptr1->next != lptr)
        {
            ② if (ptr1->data > ptr1->next->data)
            {
                ③ swap(ptr1, ptr1->next);
                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    } while (swapped);
}
```

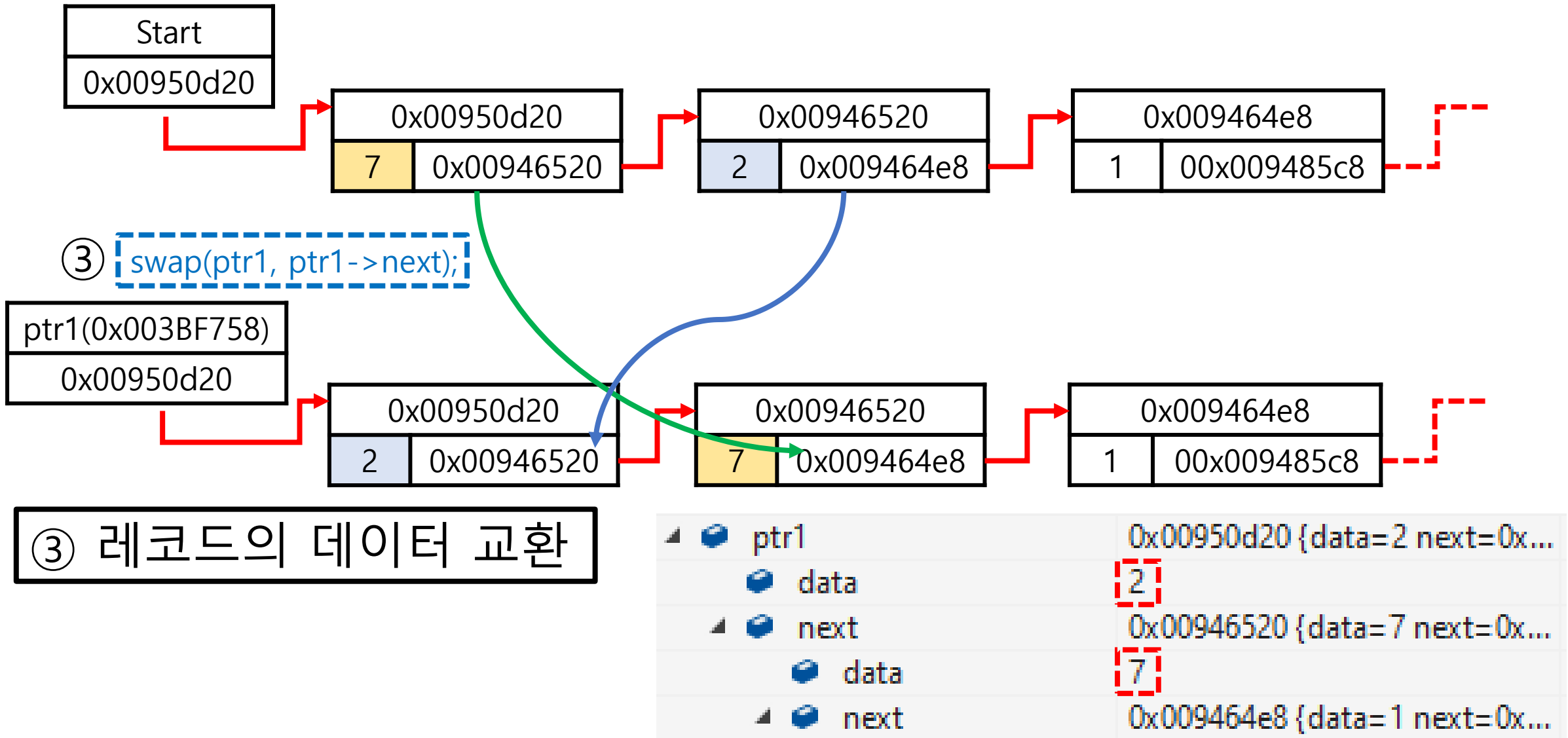
② 링크드 리스트의 노드의  
데이터와 다음 노드의  
데이터를 비교

③ 조건이 맞으면 레코드의  
데이터 교환

이름	값
ptr1->data	7
ptr1->next->data	2

이름	값
ptr1->data	2
ptr1->next->data	7

## 2. 버블 정렬 - 구현 (selection)



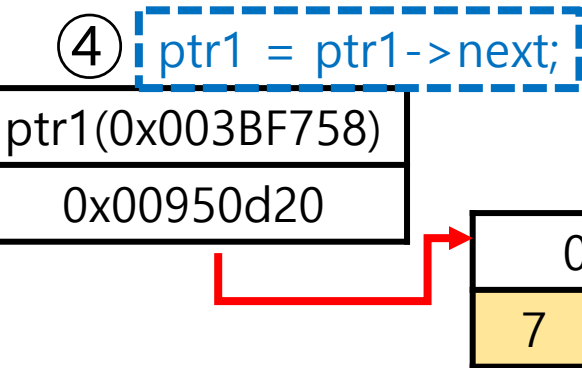
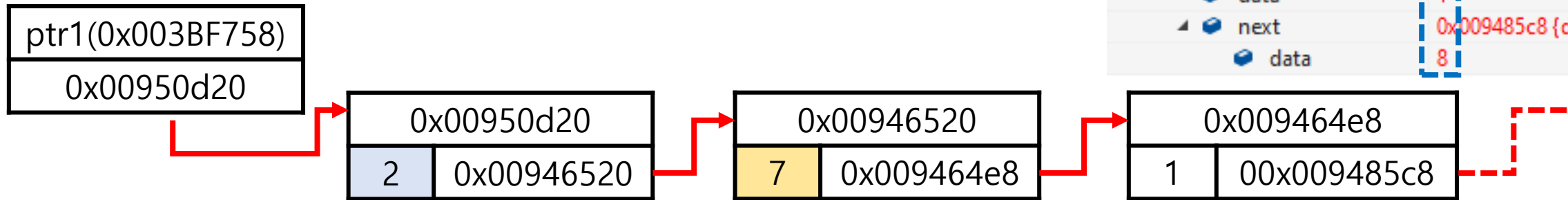
## 2. 버블 정렬 - 구현 (selection)

```
while (ptr1->next != lptr)
{
    if (ptr1->data > ptr1->next->data)
    {
        swap(ptr1, ptr1->next); 경과시간
        swapped = 1;
    }
    ④ ptr1 = ptr1->next;
}
lptr = ptr1;
```

이름	값
swapped	1
i	-858993460
ptr1	0x00950d20
data	2
next	0x00946520
data	7
next	0x009464e8
data	1
next	0x009485c8
data	8
next	0x00948590

④ 탐색을 위해 이동

이름	값
swapped	1
i	-858993460
ptr1	0x00946520 {data=7 next=0x.}
data	7
next	0x009464e8 {data=1 next=0x.}
data	1
next	0x009485c8 {data=8 next=0x.}
data	8
next	0x00948590



## 2. 버블 정렬 - 구현 (selection)

```
while (ptr1->next != lptr)
{
    if (ptr1->data > ptr1->next->data)
    {
        swap(ptr1, ptr1->next); 경과시간
        swapped = 1;
    }
    ptr1 = ptr1->next;
}
lptr = ptr1;
```

• 반복문 수행 - 교환

이름	값
swapped	1
ptr1	0x00946520 {data=1 next=0x009464e8}
data	1
next	0x009464e8 {data=7 next=0x009485c8}
data	7
next	0x009485c8 {data=8 next=0x00948590}
data	8
next	0x00948590 {data=3 next=0x009405a0}
data	3
next	0x009405a0 {data=5 next=0x00000000}
data	5
next	0x00000000 <NULL>

ptr1	0x009485c8 {data=3 next=0x00948590}
data	3
next	0x00948590 {data=8 next=0x009405a0}
data	8
next	0x009405a0 {data=5 next=0x00000000}
data	5
next	0x00000000 <NULL>

## 2. 버블 정렬 - 구현 (selection)

```
while (ptr1->next != lptr)
{
    if (ptr1->data > ptr1->next->data)
    {
        swap(ptr1, ptr1->next); 경과시간
        swapped = 1;
    }
    ptr1 = ptr1->next;
}
lptr = ptr1;
```

• 반복문 수행 - 교환

이름	값
swapped	1
ptr1	0x00948590 {data=8 next=0x...
data	8
next	0x009405a0 {data=5 next=0x...
data	5
next	0x00000000 <NULL>

이름	값
swapped	1
ptr1	0x00948590 {data=5 next=0x...
data	5
next	0x009405a0 {data=8 next=0x...
data	8
next	0x00000000 <NULL>



## 2. 버블 정렬 - 구현 (selection)

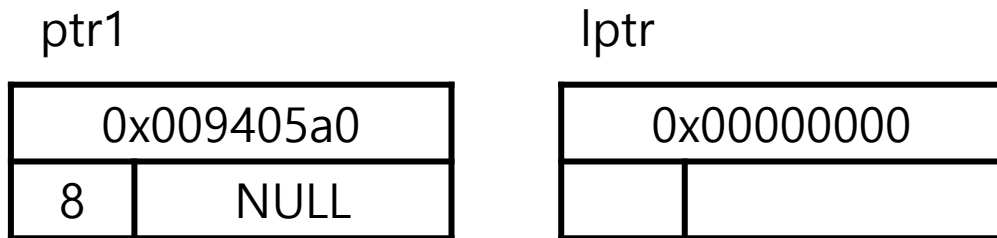
```
while (ptr1->next != lptr)
{
    if (ptr1->data > ptr1->next->data)
    {
        swap(ptr1, ptr1->next); 경과시간
        swapped = 1;
    }
    ptr1 = ptr1->next;
}
lptr = ptr1;
```

- 반복문 수행 후,  
조건 체크, 반복 종료

이름	값
ptr1->data	8
ptr1->next->data	8
ptr1->next	0x00000000 <NULL>
lptr	0x00000000 <NULL>

## 2. 버블 정렬 - 구현 (selection)

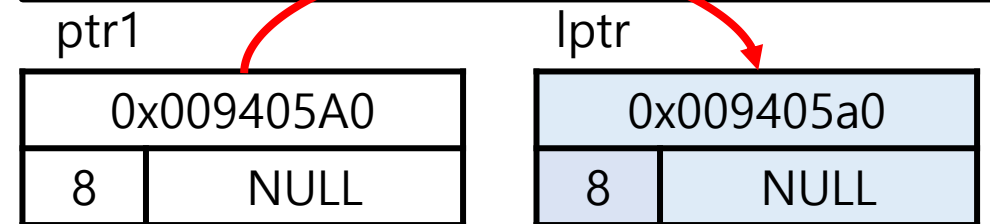
```
while (ptr1->next != lptr)
{
    if (ptr1->data > ptr1->next->data)
    {
        swap(ptr1, ptr1->next); 경과시간
        swapped = 1;
    }
    ptr1 = ptr1->next;
}
lptr = ptr1;
```



이름	값
swapped	1
ptr1	0x009405a0 {data=8 next=0x...
data	8
next	0x00000000 <NULL>
lptr	0x00000000 <NULL>

⑤ 정렬된 레코드는 완료된 구조체 배열로 복사한다

그리고 해당 위치까지 정렬 완료를 위해 반복을 다시 수행 한다.



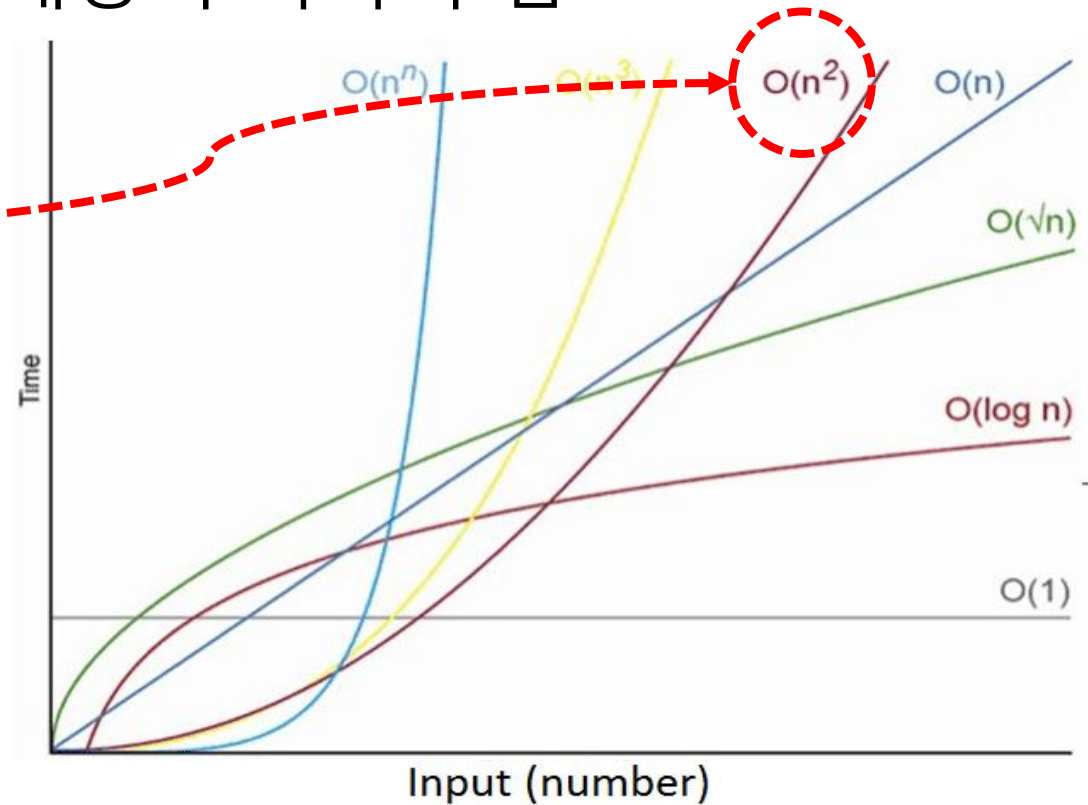
이름	값
swapped	1
ptr1	0x009405a0 {data=8 next=0x...
data	8
next	0x00000000 <NULL>
lptr	0x009405a0 {data=8 next=0x...
data	8
next	0x00000000 <NULL>

## 2. 버블 정렬 복잡도 분석

- 비교 횟수(최상, 평균, 최악의 경우 모두 일정)
  - > 한번 반복할 때 마다 비교 대상이 하나씩 감소

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- 이동 횟수
  - > 평균의 경우 :  $O(n^2)$



- 예) 정렬할 레코드 개수가 6개 일 때, 총  $n-1$  회 반복.  
-> 즉, 5회 반복하며  $5+4+3+2+1 =$  15번 비교

### 3. 삽입 정렬 (insertion sort)



#### ■ 정의 :

-> 정렬된 데이터를  
비교하여

올바른 위치에 삽입  
하는 정렬

정렬 안된 부분

정렬된 부분

### 3. 삽입 정렬 (insertion sort)



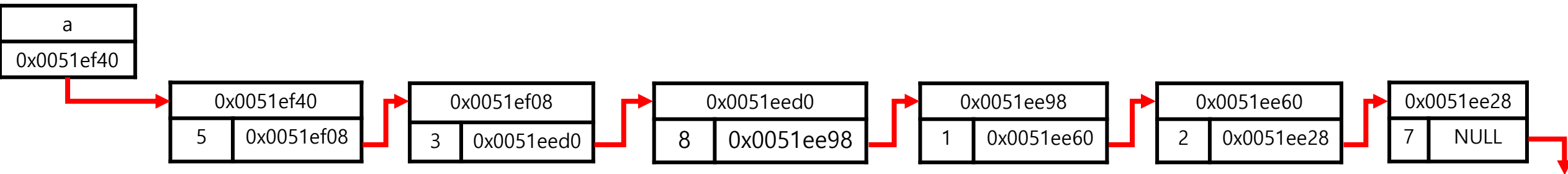
■ 예 :

-> 정렬된 데이터에서  
오름차순으로  
'1'을 삽입하는 과정

■ Key :

-> 이동할 때, 데이터를  
덮어쓰므로 임시 저  
장을 위한 변수

# 3. 삽입 정렬 (insertion sort)



int main()

{

①

```
struct Node* a = NULL;
push(&a, 7);
push(&a, 2);
push(&a, 1);
push(&a, 8);
push(&a, 3);
push(&a, 5);
```

```
printf("insertionSort before sorting %n");
printList(a);
```

②

```
insertionSort(&a);
```

```
printf("insertionSort after sorting %n");
printList(a);
```

```
return 0;
```

}

- ① 삽입 정렬 변수 선언
- ② 삽입 정렬 함수 호출

이름	값
a	0x006fef40 {data=5 next=0x006fef08 {data=3 n
data	5
next	0x006fef08 {data=3 next=0x006feed0 {data=8 n
data	3
next	0x006feed0 {data=8 next=0x006fee98 {data=1 n
data	8
next	0x006fee98 {data=1 next=0x006fee60 {data=2 n
data	1
next	0x006fee60 {data=2 next=0x006fee28 {data=7 n
data	2
next	0x006fee28 {data=7 next=0x00000000 <NULL>
data	7
next	0x00000000 <NULL>

```
insertionSort before sorting
5 3 8 1 2 7
```

# 3. 삽입 정렬 (insertion sort)

```
// function to sort a singly linked list using insertion sort
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;

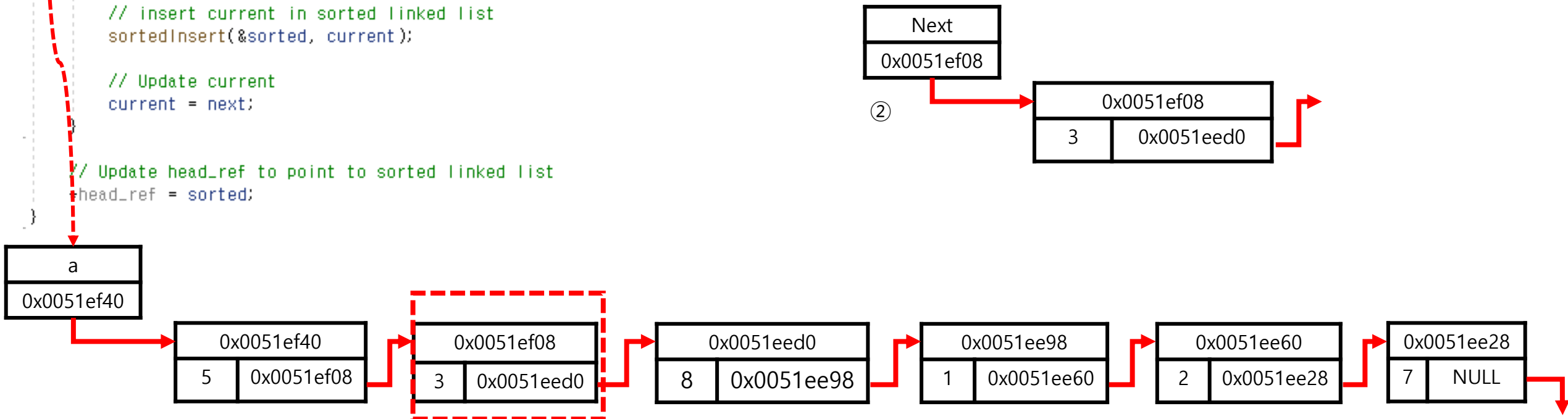
    // Traverse the given linked list and insert every
    // node to sorted
    ① struct Node* current = *head_ref;
    while (current != NULL)
    {
        // Store next for next iteration
        ② struct Node* next = current->next;

        // insert current in sorted linked list
        sortedInsert(&sorted, current);

        // Update current
        current = next;
    }

    // Update head_ref to point to sorted linked list
    *head_ref = sorted;
}
```

- ① 정렬 대상 노드 리스트를 복사 해둔다.
- ② 정렬 후, 다음 정렬을 위한 다음 노드 저장



### 3. 삽입 정렬 (insertion sort)

```
// function to sort a singly linked list using insertion sort
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;

    // Traverse the given linked list and insert every
    // node to sorted
    struct Node* current = *head_ref;
    while (current != NULL)
    {
        // Store next for next iteration
        struct Node* next = current->next;

        // insert current in sorted linked list
        ① sortedInsert(&sorted, current);

        // Update current
        current = next;
    }

    // Update head_ref to point to sorted linked list
    *head_ref = sorted;
}
```

① 삽입 정렬을 수행  
하는 함수를 호출.

sorted		0x00000000 <NULL>
current		0x0051ef40 {data=5 next=0x0051ef08 {data=3
data	5	
next	0x0051ef08 {data=3 next=0x0051eed0 {data=8	
data	3	
next	0x0051eed0 {data=8 next=0x0051ee98 {data=1	
data	8	
next	0x0051ee98 {data=1 next=0x0051ee60 {data=2	
data	1	
next	0x0051ee60 {data=2 next=0x0051ee28 {data=7	
data	2	
next	0x0051ee28 {data=7 next=0x00000000 <NULL>	
data	7	
next	0x00000000 <NULL>	



### 3. 삽입 정렬 (insertion sort)

```
void sortedInsert(struct Node** head_ref, struct Node* new_node)
{
    struct Node* current;
    /* Special case for the head end */
    ① if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        ② new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next != NULL &&
            current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
```

① 매개변수 노드가 NULL 조건

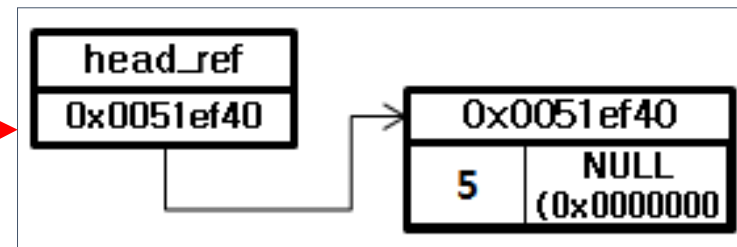
head_ref	0x0027fc20 {0x00000000 <NULL>}
new_node	0x0051ef40 {data=5 next=0x0051ef08 {data=3 next=0x0051eed0 {data=8 next=0x00000000 <NULL>}}}
data	5
next	0x0051ef08 {data=3 next=0x0051eed0 {data=8 next=0x00000000 <NULL>}}

② 정렬된 노드로 입력된 노드 입력

new_node	0x0051ef40
data	5
next	0x00000000 <NULL>

new_node	0x0051ef40 {data=5 next=0x00000000 <NULL> }
data	5
next	0x00000000 <NULL>



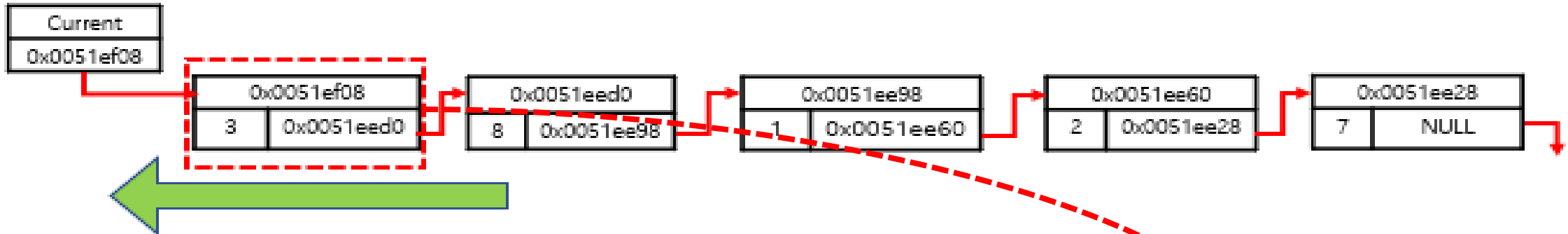
head_ref	0x0027fc20 {0x0051ef40 {data=5 next=0x00000000 <NULL>}}
data	5
next	0x00000000 <NULL>

# 3. 삽입 정렬 (insertion sort)

```
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;

    // Traverse the given linked list and insert every
```

① 매개변수 노드가 NULL 조건



```
    // Update current
    ① current = next;
```

current	0x0051ef40 {data=5 next=0x00000000}	list
data	5	
next	0x00000000 <NULL>	
next	0x0051ef08 {data=3 next=0x0051eed0}	
data	3	
next	0x0051eed0 {data=8 next=0x0051ee98}	



current	0x0051ef08 {data=3 next=0x0051eed0}
data	3
next	0x0051eed0 {data=8 next=0x0051ee98}
data	8
next	0x0051ee98 {data=1 next=0x0051ee60}

### 3. 삽입 정렬 (insertion sort)

```
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;

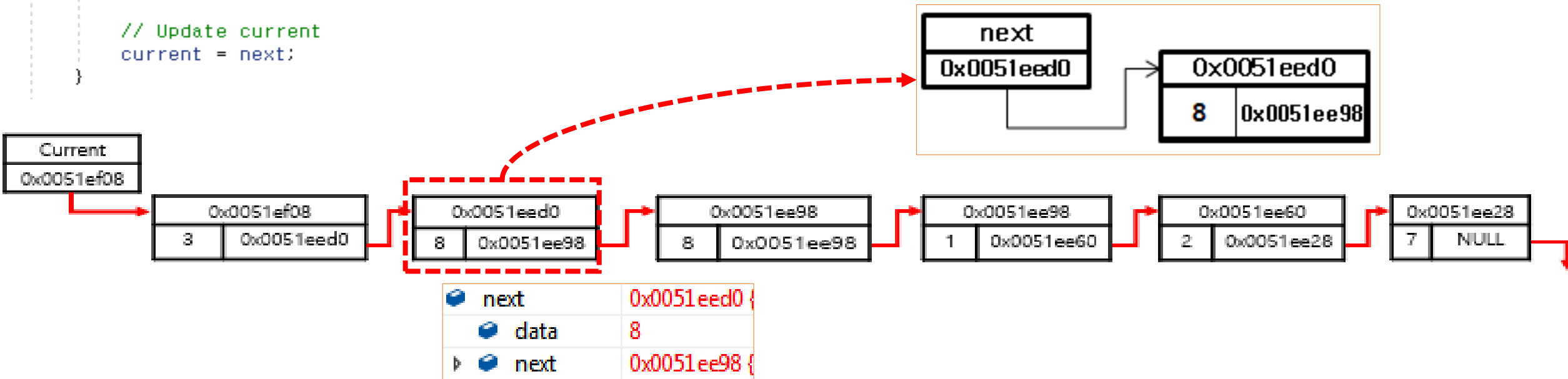
    // Traverse the given linked list and insert every
    // node to sorted
    struct Node* current = *head_ref;
    while (current != NULL)
    {
        // Store next for next iteration
        struct Node* next = current->next;

        // insert current in sorted linked list
        sortedInsert(&sorted, current);

        // Update current
        current = next;
    }
}
```

① 매개변수 노드가  
NULL 조건

② 정렬 후, 다음 정렬을 위한  
다음 노드 저장



# 3. 삽입 정렬 (insertion sort)

```
void sortedInsert(struct Node** head_ref, struct Node* new_node)
```

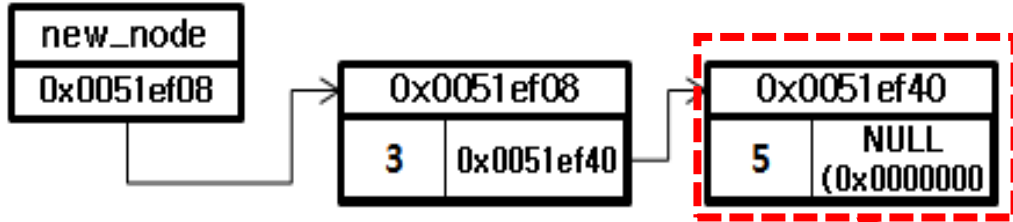
```
{  
    struct Node* current;  
    /* Special case for the head end */  
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)  
    {  
        new_node->next = *head_ref;  
        *head_ref = new_node;  
    }  
}
```

①

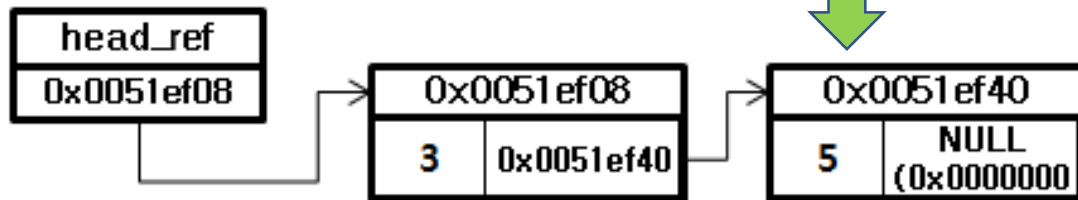
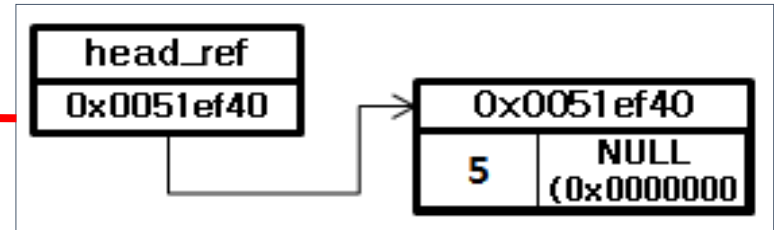
②

① 조건이 ( $5 \geq 3$ ) 참이므로 수행

② 노드 3의 다음 노드를 기존의 노드로 **복사**한다.



new_node	0x0051ef08 {data=3 next=0x0051ef40 {data=5 next=0x00000000 <NULL>}}
data	3
next	0x0051ef40 {data=5 next=0x00000000 <NULL>}
data	5
next	0x00000000 <NULL>



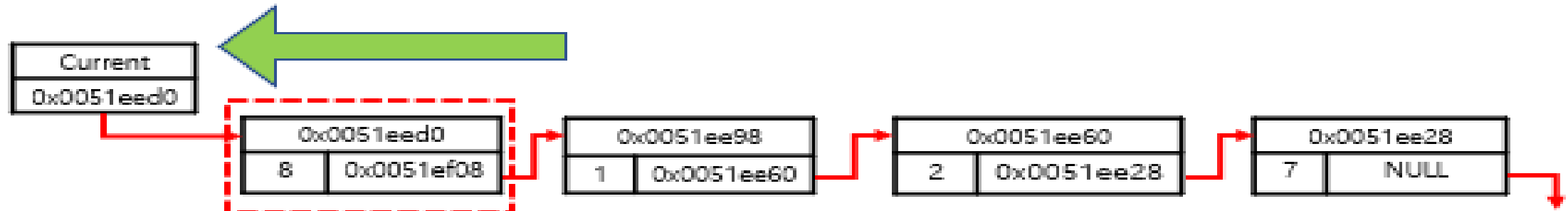
head_ref	0x0027fc20 {0x0051ef08 {data=3 next=0x0051ef40 {data=5 next=0x00000000 <NULL>}}}
data	3
next	0x0051ef40 {data=5 next=0x00000000 <NULL>}
data	5
next	0x00000000 <NULL>

### 3. 삽입 정렬 (insertion sort)

```
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;

    // Traverse the given linked list and insert every
```

① 다음 노드 정렬을 위해 위치를 이동한다.



```
    ① // Update current
    current = next;
}
```

current	0x0051ef08 {data=3 next=0x0051ef40 }
data	3
next	0x0051ef40 {data=5 next=0x00000000 }
next	0x0051eed0 {data=8 next=0x0051ee98 }
data	8
next	0x0051ee98 {data=1 next=0x0051ee60 }



current	0x0051eed0 {data=8 next=0x0051ee98 }
data	8
next	0x0051ee98 {data=1 next=0x0051ee60 }

### 3. 삽입 정렬 (insertion sort)

```
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;

    // Traverse the given linked list and insert every
    // node to sorted
    struct Node* current = *head_ref;
    ① while (current != NULL)
    {
        ② // Store next for next iteration
        struct Node* next = current->next;

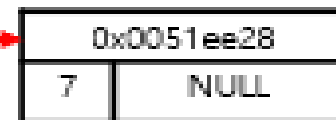
        // insert current in sorted linked list
        ③ sortedInsert(&sorted, current);
    }
}
```

① 정렬 대상 노드 리스트를 복사 해둔다

② 정렬 후, 다음 정렬을 위한 다음 노드 저장



②



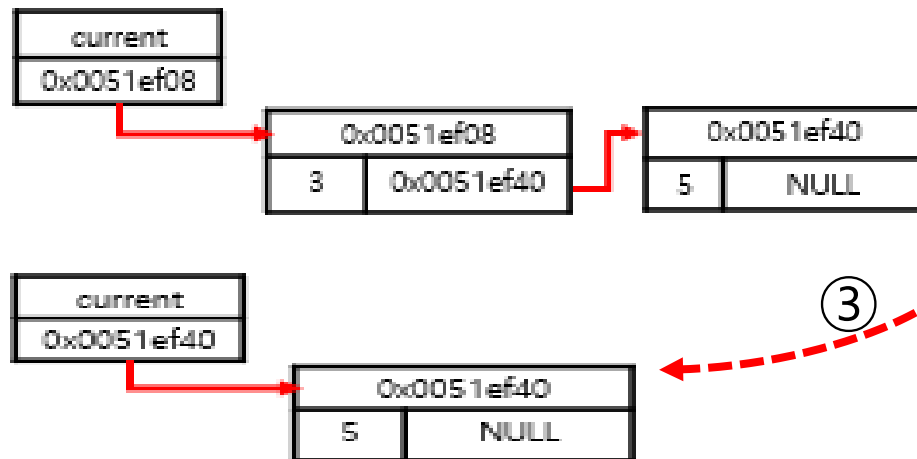
current	0x0051eed0
data	8
next	0x0051ee98
next	0x0051ee98
data	1
next	0x0051ee60

sorted	0x0051ef08
data	3
next	0x0051ef40
data	5
next	0x00000000
current	0x0051eed0
data	8
next	0x0051ee98

# 3. 삽입 정렬 (insertion sort)

```
void sortedInsert(struct Node** head_ref, struct Node* new_node)
{
    struct Node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        ① current = *head_ref;
        ② while (current->next != NULL &&
            current->next->data < new_node->data)
        {
            ③ current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
```

current	0xffffffff
head_ref	0x0027fc20
data	3
next	0x0051ef40
data	5
next	0x00000000
new_node	0x0051eed0
data	8
next	0x0051ee98



- ① 정렬 대상 노드를 복사 해 둔다
- ② 다음 노드가 비어 있지 않고 새 노드(new)보다 작을 때까지 반복
- ③ 비교를 위해 탐색

current	0x0051ef08
data	3
next	0x0051ef40
data	5
next	0x00000000

current	0x0051ef40
data	5
next	0x00000000

### 3. 삽입 정렬 (insertion sort)

```
void sortedInsert(struct Node** head_ref, struct Node* new_node)
```

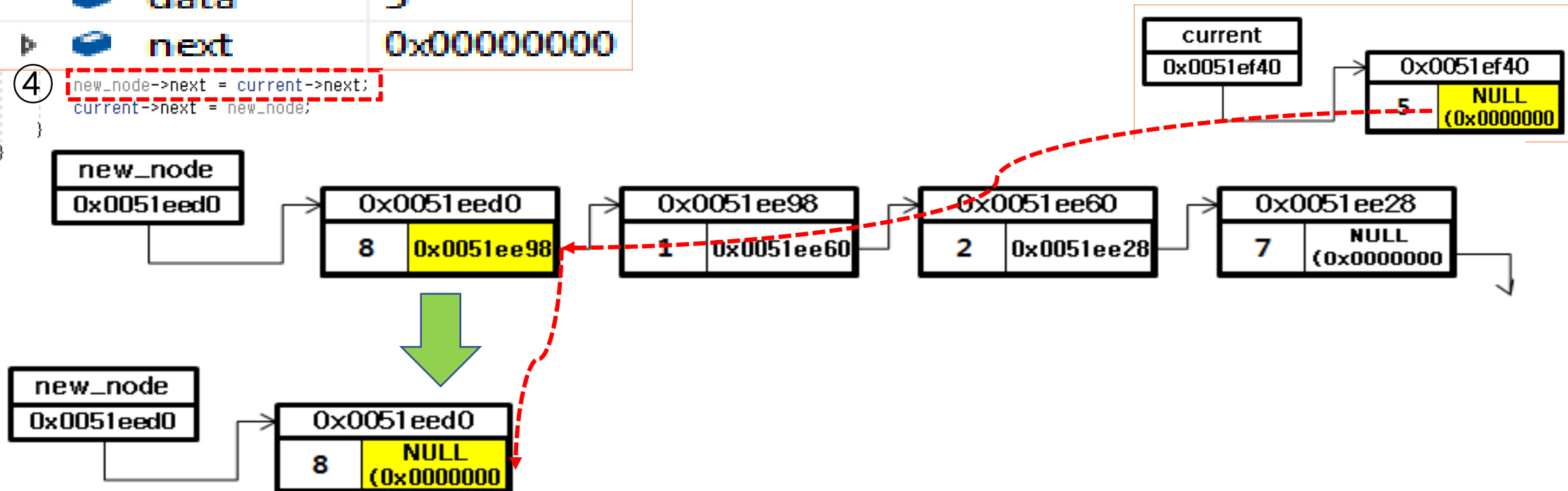
new_node	0x0051eed0
data	8
next	0x0051ee98
data	1
next	0x0051ee60
current	0x0051ef40
data	5
next	0x00000000



new_node	0x0051eed0
data	8
next	0x00000000

④ 현재 노드의 다음 참조 주소를 새로운 노드의 다음 주소로 복사

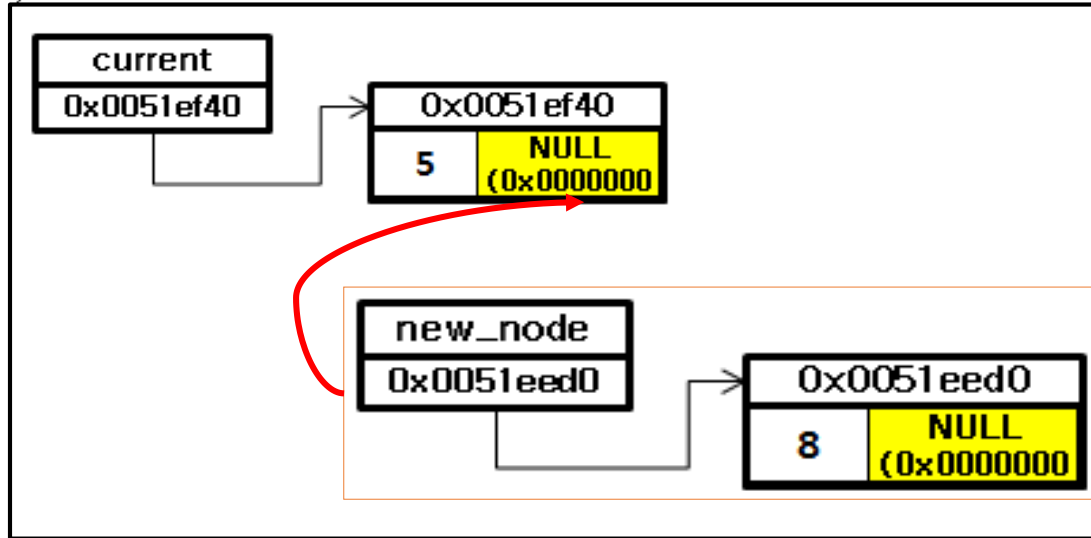
④ `new_node->next = current->next;`  
`current->next = new_node;`



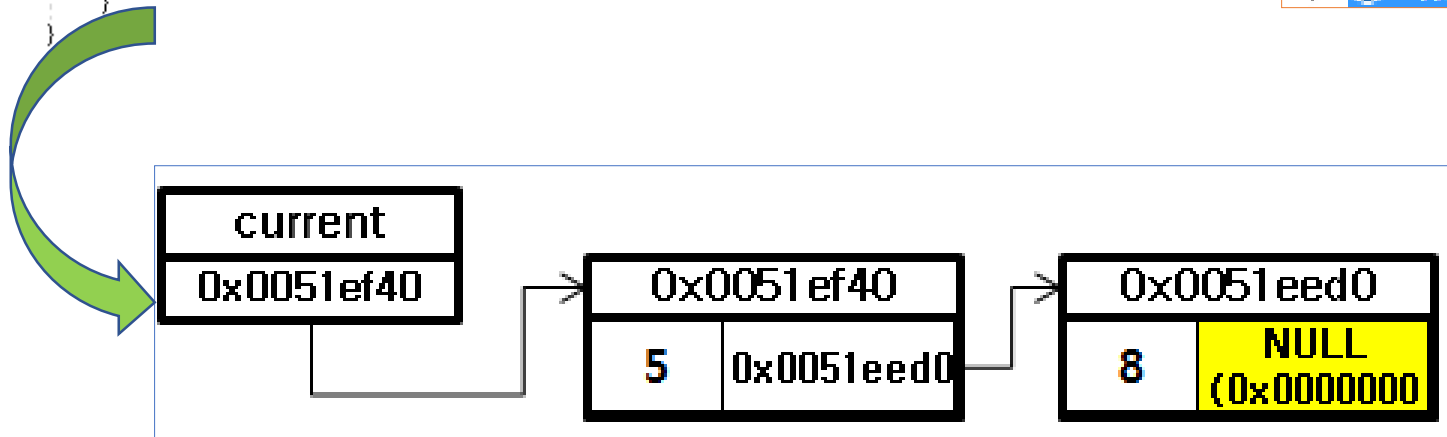


### 3. 삽입 정렬 (insertion sort)

```
void sortedInsert(struct Node** head_ref, struct Node* new_node)
```



⑤ `new_node->next = current->next;`  
`current->next = new_node;`



⑤ 현재 노드의 다음 참조 주소를 새로운 노드의 다음 주소로 복사

current	0x0051ef40
data	5
next	0x00000000

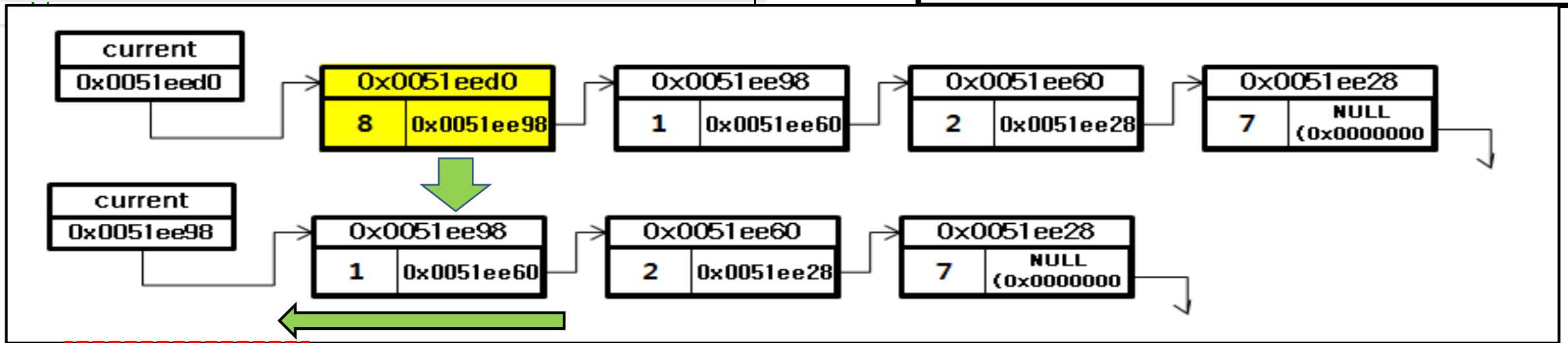


current	0x0051ef40
data	5
next	0x0051eed0
data	8
next	0x00000000

### 3. 삽입 정렬 (insertion sort)

```
void insertionSort(struct Node** head_ref)
{
    // Initialize sorted linked list
    struct Node* sorted = NULL;
```

⑤ 다음 노드 정렬을 위해 위치를 이동한다.



① `current = next;`

current	0x0051eed0
data	8
next	0x00000000
next	0x0051ee98 {
data	1
next	0x0051ee60 {



current	0x0051ee98 {
data	1
next	0x0051ee60 {

### 3. 삽입 정렬 (insertion sort)

```
void insertionSort(struct Node** head_ref)
```

```
void sortedInsert(struct Node** head_ref, struct Node* new_node)
```

- ① 삽입함수에서는 정렬할 노드를 탐색
- ② 정렬된 노드와 비교대상 노드를 호출하여 값의 비교를 통해 노드 위치를 비교 후, 변경한다.



### 3. 삽입 정렬 복잡도 분석

- 최선의 경우  $O(n)$ : 이미 정렬되어 있는 경우

-> 비교 :  $n-1$  번

- 최선의 경우  $O(n^2)$ : 이미 정렬되어 있는 경우

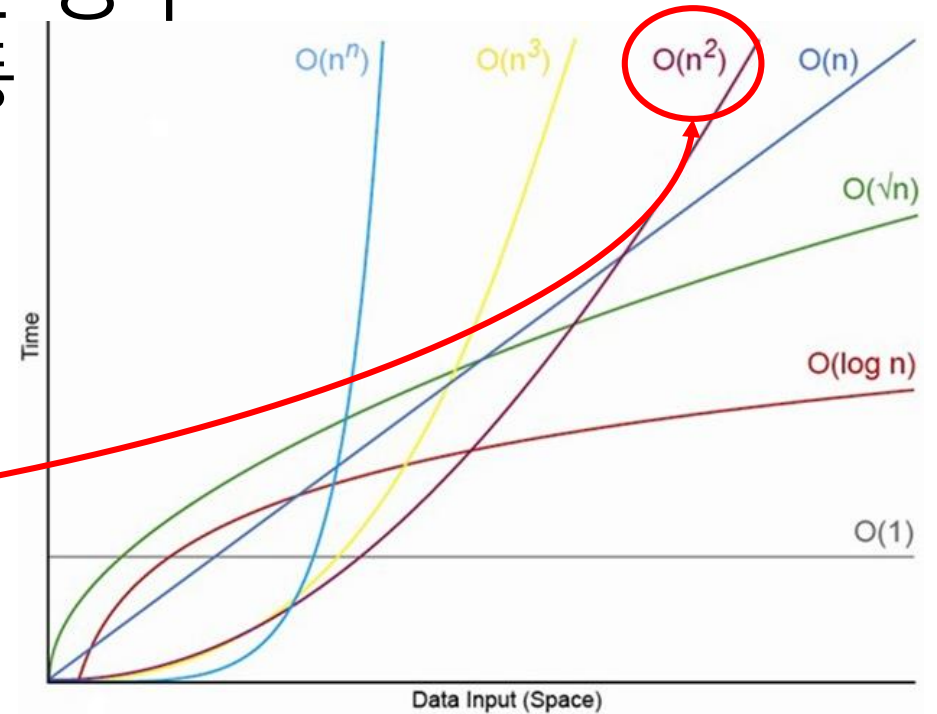
-> 모든 단계에서 앞에 놓인 자료 전부 이동

-> 비교 :  $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$

-> 이동 :  $\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$

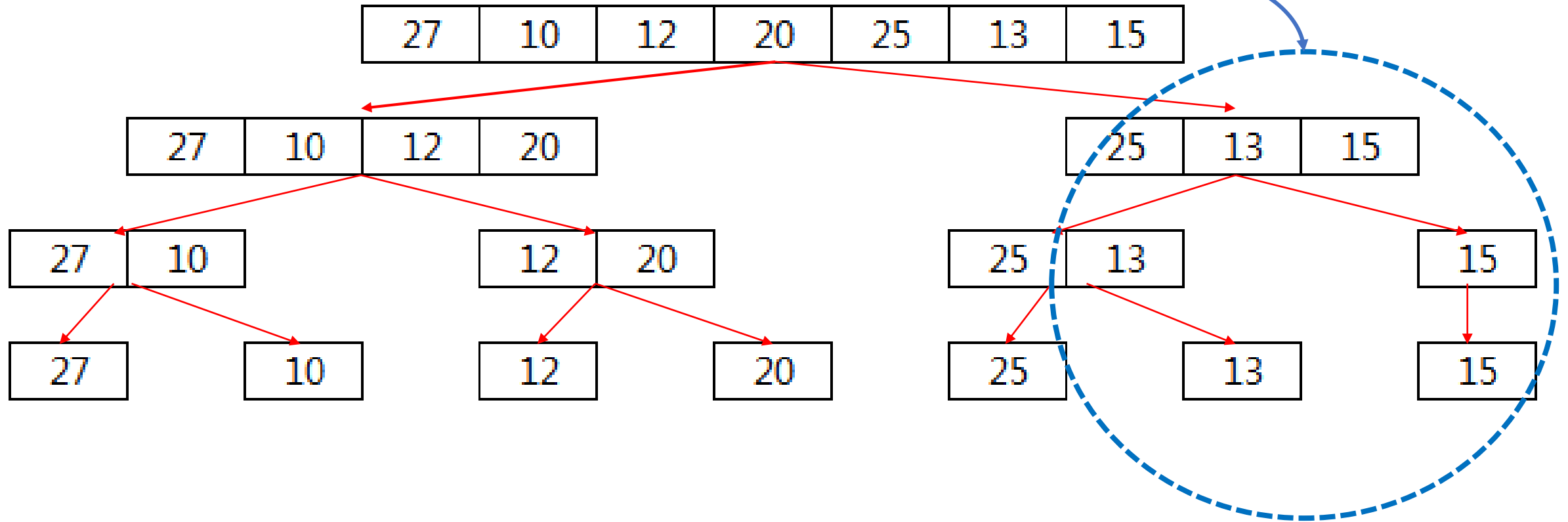
- 평균의 경우  $O(n^2)$

- 대부분 정렬되어 있으면 매우 효율적



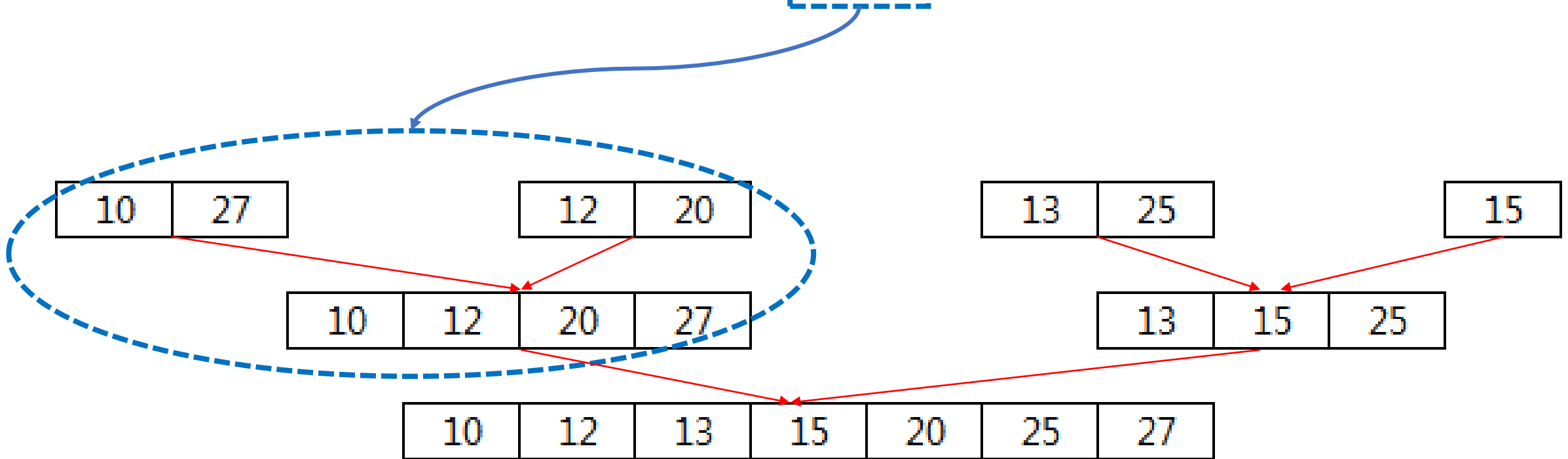
## 4. 합병 정렬 (Merge sort)

- 전체의 레코드를 하나의 단위로 **분할**하고

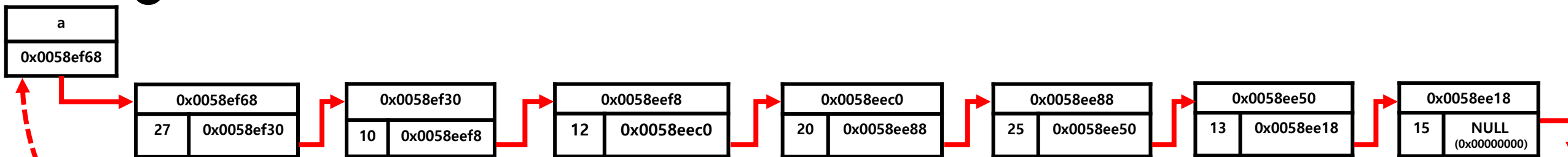


## 4. 합병 정렬 (Merge sort)

- 다시 분할한 레코드를 **합병**하는 정렬



# 4. 합병 정렬 - 구현 (selection)



```
int main()
{
    /* Start with the empty list */
    struct Node* res = NULL;
    struct Node* a = NULL;

    /* Let us create a unsorted linked lists to test the functions
    Created lists shall be a: 2->3->20->5->10->15 */
    push(&a, 15);
    push(&a, 13);
    push(&a, 25);
    push(&a, 20);
    push(&a, 12);
    push(&a, 10);
    push(&a, 27);

    printf("\n Not Sorted Linked List is: \n");
    printList(a);

    /* Sort the above created Linked List */
    MergeSort(&a);

    printf("\n Sorted Linked List is: \n");
    printList(a);

    //getchar();
    return 0;
}
```

합병 정렬 수행!!

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

# 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

-- 데이터 나누기!!

이름	값
▶ a	0xc0000000 {data=??? next=??? }
▶ b	0xc0000000 {data=??? next=??? }
◀ head	0x0058ef68 {data=27 next=0x0058ef30 }
data	27
▶ next	0x0058ef30 {data=10 next=0x0058eef8 }
data	10
▶ next	0x0058eef8 {data=12 next=0x0058eec0 }
data	12
▶ next	0x0058eec0 {data=20 next=0x0058ee88 }
data	20
▶ next	0x0058ee88 {data=25 next=0x0058ee50 }
data	25
▶ next	0x0058ee50 {data=13 next=0x0058ee18 }
data	13
▶ next	0x0058ee18 {data=15 next=0x00000000 }
data	15
▶ next	0x00000000 <NULL>
data	<메모리를 읽을 수 없음>
next	<메모리를 읽을 수 없음>
◀ headRef	0x0020fc14 {0x0058ef68 {data=27 ne 0x0058ef68 {data=27 next=0x0058ef30 }
data	27
▶ next	0x0058ef30 {data=10 next=0x0058eef8 }
data	10
▶ next	0x0058eef8 {data=12 next=0x0058eef8 }



# 4. 합병 정렬 - 구현 (selection)

```
void FrontBackSplit(struct Node* source, struct Node** frontRef, struct Node** backRef)
```

① 리스트를 나눈다(분리)

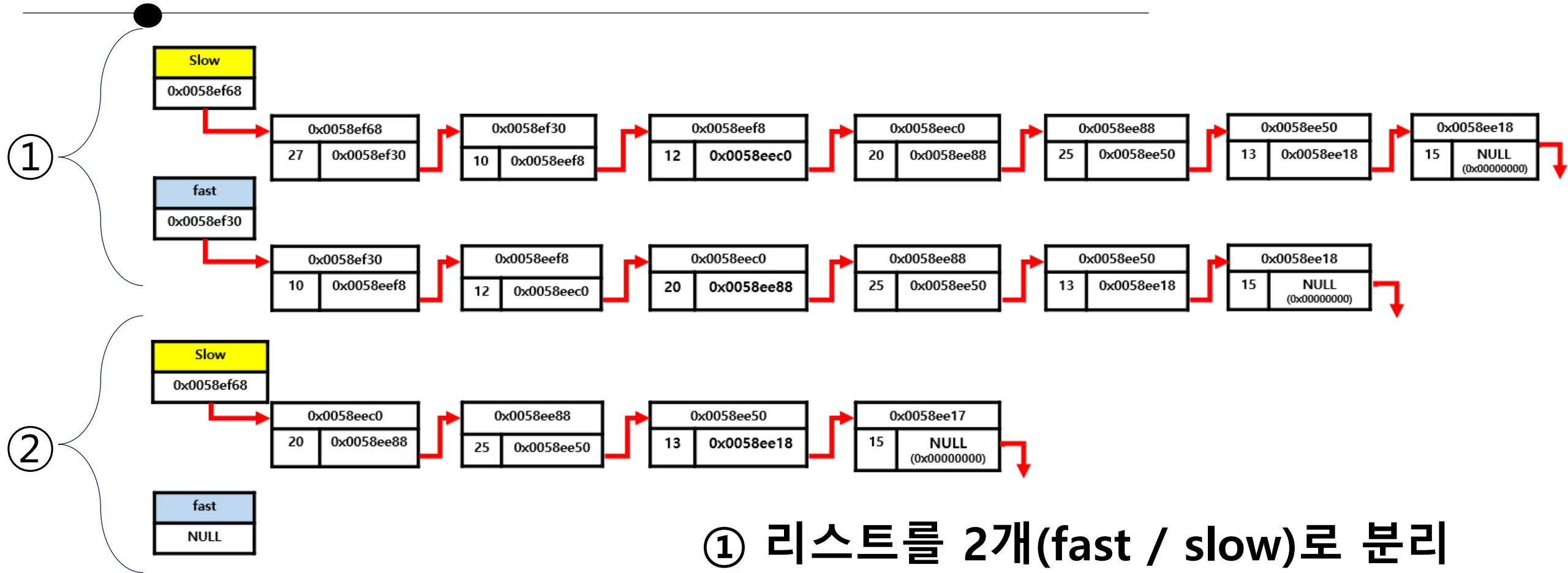
② 현재 리스트를 절반으로  
나눌 때까지 수행  
(즉,  $n/2$ 로 나눌 때 까지 수행)

```
{
    struct Node* fast;
    struct Node* slow;
    if (source == NULL || source->next == NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
    else
    {
        slow = source;
        fast = source->next;

        /* Advance 'fast' two nodes, and advance 'slow' one node */
        while (fast != NULL)
        {
            fast = fast->next;
            if (fast != NULL)
            {
                slow = slow->next;
                fast = fast->next;
            }
        }

        /* 'slow' is before the midpoint in the list, so split it in two
        at that point. */
        *frontRef = source;
        *backRef = slow->next;
        slow->next = NULL;
    }
}
```

## 4. 합병 정렬 - 구현 (selection)



① 리스트를 2개(fast / slow)로 분리

② 분리는 반으로 나눌 때 까지 반복  
(코드에서는 한쪽(fast)이 NULL까지)

# 4. 합병 정렬 - 구현 (selection)

```
void FrontBackSplit(struct Node* source, struct Node** frontRef, struct Node** backRef)
```

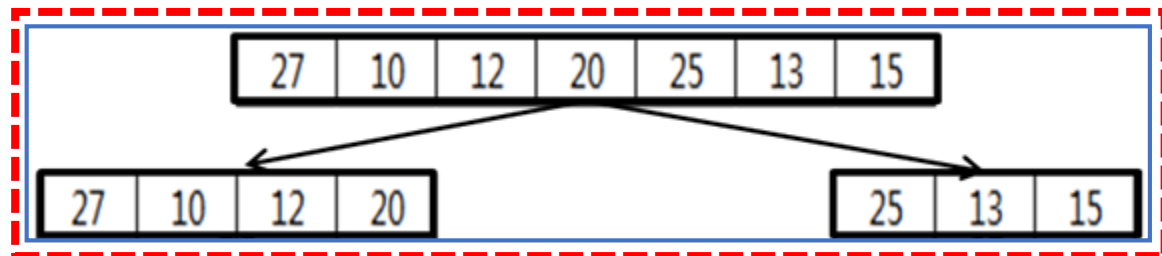
```
{
    struct Node* fast;
    struct Node* slow;
    if (source == NULL || source->next == NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
    else
    {
        slow = source;
        fast = source->next;

        /* Advance 'fast' two nodes, and advance 'slow' one node */
        while (fast != NULL)
        {
            fast = fast->next;
            if (fast != NULL)
            {
                slow = slow->next;
                fast = fast->next;
            }
        }

        /* 'slow' is before the midpoint in the list, so split it in two
        at that point. */
        *frontRef = source;
        *backRef = slow->next;
        slow->next = NULL;
    }
}
```

③ 정렬 대상 노드 리스트를 절반 지점 ( $n/2$ )을 맞추고

매개변수 메모리에 2개의 레코드의 시작점으로 분할 처리

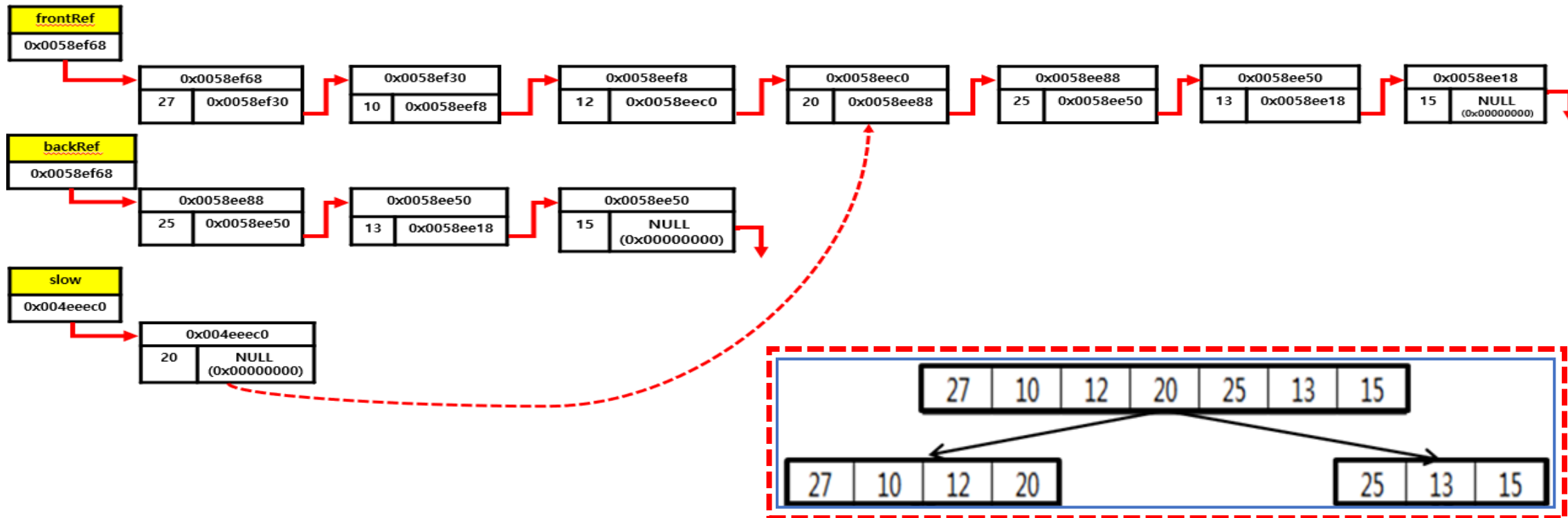


③

## 4. 합병 정렬 - 구현 (selection)

```
*frontRef = source; ③  
*backRef = slow->next;  
slow->next = NULL;
```

③ 2 개의 레코드의 시작점으로 분할  
처리 확인



# 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    ① FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    ② MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

① 분할 함수 실행 후,  
나누어진 데이터 확인

이름	값
▲ a	0x004eef68 {data=27 next=0x004eef30 {data=10
data	27
▲ next	0x004eef30 {data=10 next=0x004eeef8 {data=12
data	10
▲ next	0x004eeef8 {data=12 next=0x004eeec0 {data=20
data	12
▲ next	0x004eeec0 {data=20 next=0x00000000 <NULL>
data	20
▶ next	0x00000000 <NULL>
▲ b	0x004eee88 {data=25 next=0x004eee50 {data=13
data	25
▲ next	0x004eee50 {data=13 next=0x004eee18 {data=15
data	13
▲ next	0x004eee18 {data=15 next=0x00000000 <NULL>
data	15
▶ next	0x00000000 <NULL>

② 다시 재귀호출을 통해  
merge 함수 호출(분할)

# 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

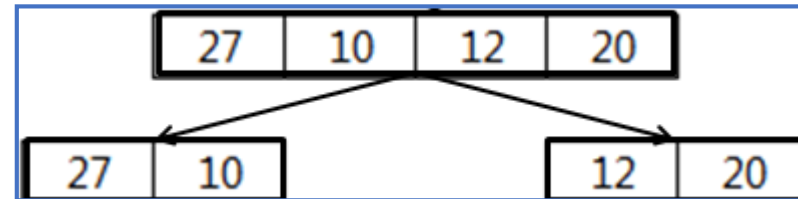
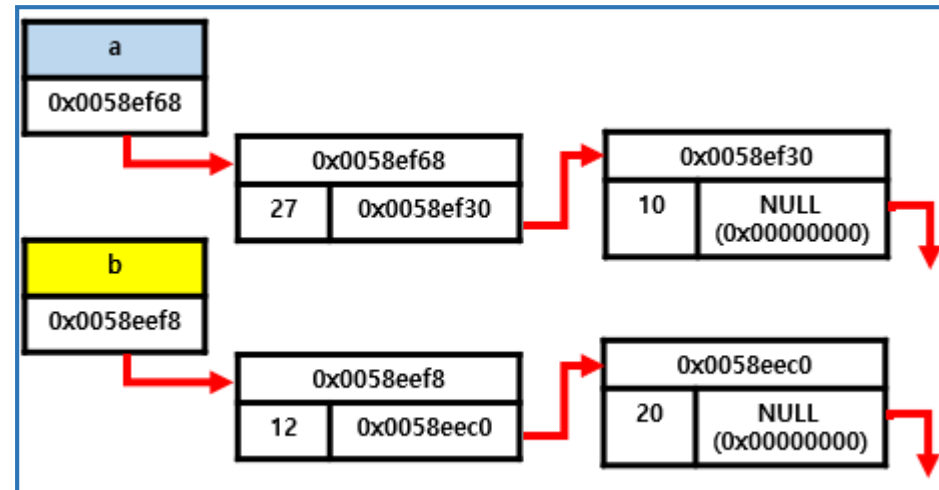
    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

① 다시 재귀호출을 통해  
합병 정렬 함수 호출(분할)



## 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

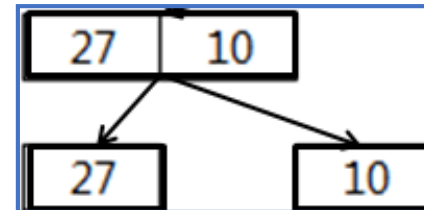
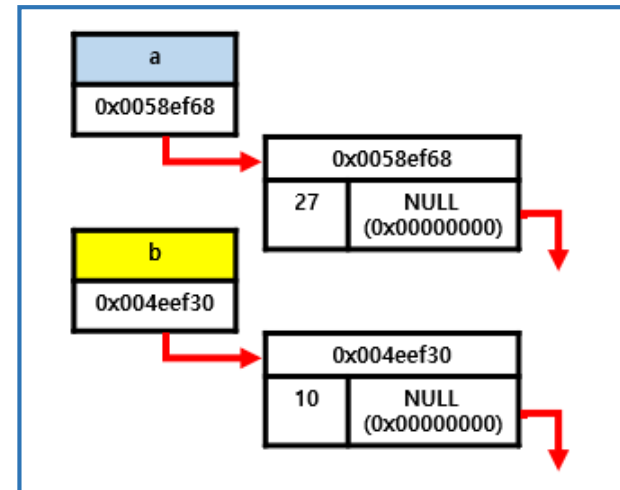
    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

① 다시 재귀호출을 통해  
합병 정렬 함수 호출(분할)

(데이터가 1개가 될 때 까지)



## 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    ② *headRef = SortedMerge(a, b);
}
```

② 분리된 데이터를 다시  
합친다 (합병)

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b == NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}
```



# 4. 합병 정렬 - 구현 (selection)

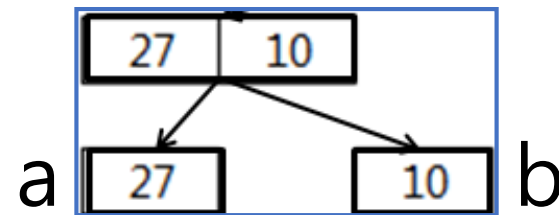
```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;

    /* Base cases */
    ① if (a == NULL)
        return(b);
    else if (b == NULL)
        return(a);

    /* Pick either a or b, and recur */
    ② if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}
```

① 비교대상이 비어 있는지 확인한다(NULL 체크)

② 매개변수 데이터들의 크기를 비교 한 후, 결과 노드로 메모리 저장



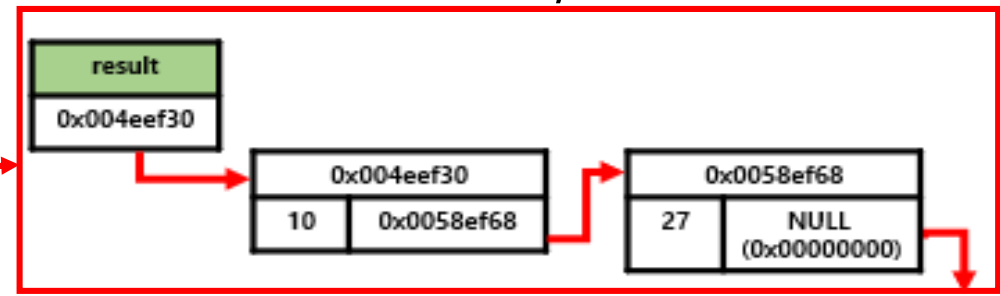
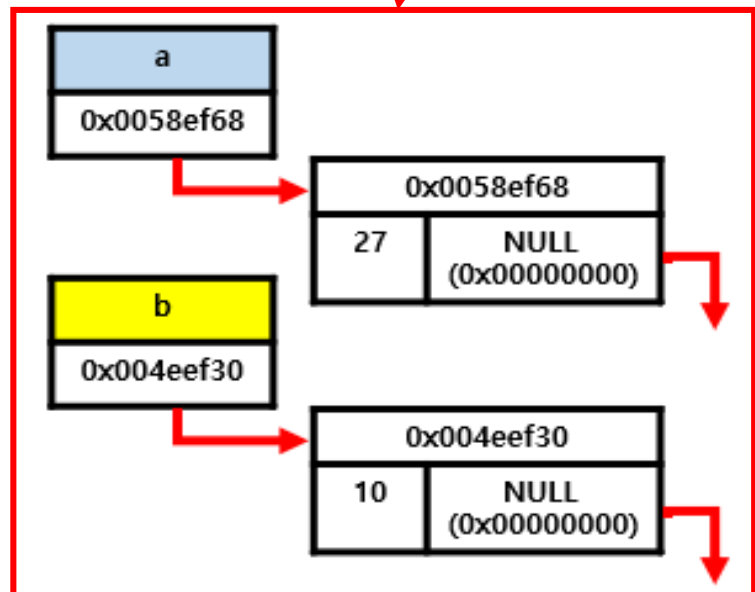
이름	값
▲ a	0x004eef68 {data=27 next=0x0000}
data	27
next	0x00000000 <NULL>
▲ b	0x004eef30 {data=10 next=0x0000}
data	10
next	0x00000000 <NULL>
▲ result	0xffffffff {data=??? next=??? }
data	<메모리를 읽을 수 없음>
next	<메모리를 읽을 수 없음>

# 4. 합병 정렬 - 구현 (selection)

```
if (a->data <= b->data)
{
    result = a;
    result->next = SortedMerge(a->next, b);
}
else
{
    result = b;
    result->next = SortedMerge(a, b->next);
}
return(result);
```

② 분리된 데이터를 다시 합친다 (합병)

합병 처리 함수 또한 필요 시, 재귀 호출



이름	값
a	0x004eef68 {data=27 next=0x00000000 <NULL>}
data	27
next	0x00000000 <NULL>
b	0x004eef30 {data=10 next=0x004eef68 {data=27 next=0x00000000 <NULL>}}
data	10
next	0x004eef68 {data=27 next=0x00000000 <NULL>}
result	0x004eef30 {data=10 next=0x004eef68 {data=27 next=0x00000000 <NULL>}}
data	10
next	0x004eef68 {data=27 next=0x00000000 <NULL>}
data	27
next	0x00000000 <NULL>

# 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

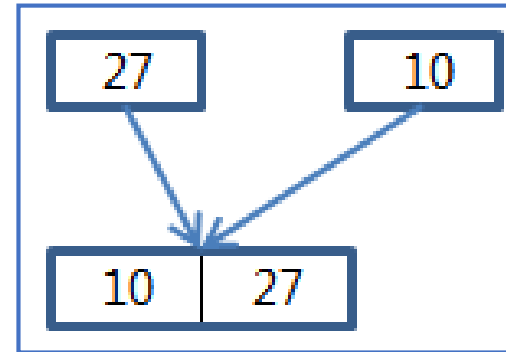
    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

③ 합병하고 반환한  
결과 데이터를 기존  
포인터 변수에 적용



이름	값
a	0x004eef68 {data=27 next=0x00000000}
b	0x004eef30 {data=10 next=0x004eef68}
head	0x004eef68 {data=27 next=0x00000000}
headRef	0x003ff6c4 {0x004eef30 {data=10 next=0x004eef68}, 0x004eef68 {data=27 next=0x00000000}}
data	10
next	0x004eef68 {data=27 next=0x00000000}
data	27
next	0x00000000 <NULL>

# 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

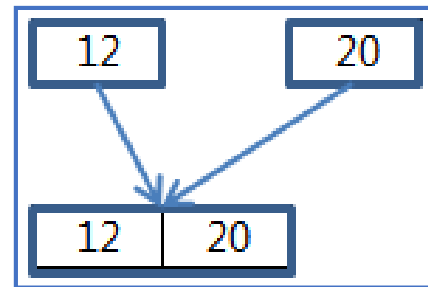
    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

③ 합병하고 반환한  
결과 데이터를 기존  
포인터 변수에 적용



이름	값
▲ a	0x004eeef8 {data=12 next=0x004eeec0 {data=20
data	12
next	0x004eeec0 {data=20 next=0x00000000 <NULL>
▲ b	0x004eeec0 {data=20 next=0x00000000 <NULL>
data	20
next	0x00000000 <NULL>
▶ head	0x004eeef8 {data=12 next=0x004eeec0 {data=20
▶ headRef	0x003ff6b8 {0x004eeef8 {data=12 next=0x004eeec0 {data=20
data	0x004eeef8 {data=12 next=0x004eeec0 {data=20
next	0x004eeec0 {data=20 next=0x00000000 <NULL>
data	20
next	0x00000000 <NULL>

## 4. 합병 정렬 - 구현 (selection)

```
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

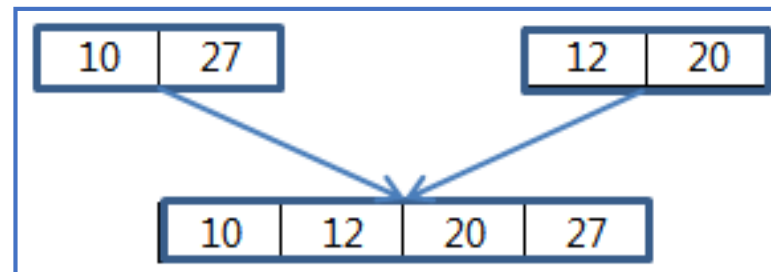
    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

③ 합병한 2개의 노드를  
다시 하나의 노드로  
합친다 ('10, 27' | '12, 20')



이름	값
▲ a	0x004eef30 {data=10 next=0x004eef68 {data=27
data	10
▲ next	0x004eef68 {data=27 next=0x00000000 <NULL>
data	27
▶ next	0x00000000 <NULL>
▲ b	0x004eef68 {data=12 next=0x004eeec0 {data=20
data	12
▲ next	0x004eeec0 {data=20 next=0x00000000 <NULL>
data	20
▶ next	0x00000000 <NULL>

## 4. 합병 정렬 - 구현 (selection)

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b == NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return(result);
}
```

①

/\* Base cases \*/  
if (a == NULL)  
 return(b);  
else if (b == NULL)  
 return(a);

②

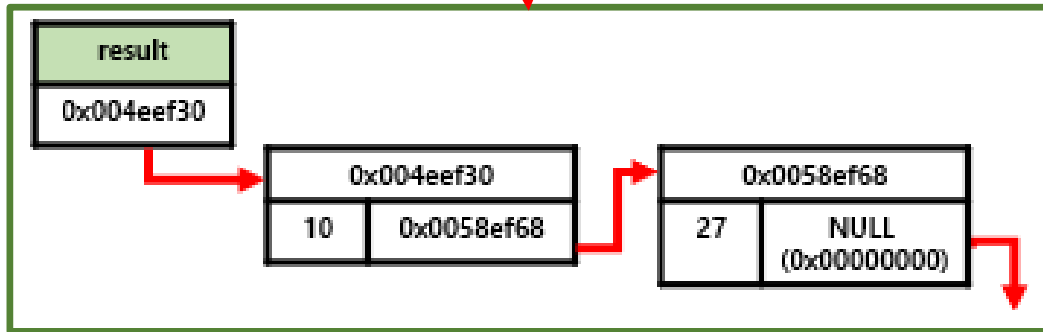
/\* Pick either a or b, and recur \*/  
if (a->data <= b->data)  
{  
 result = a;  
 result->next = SortedMerge(a->next, b);  
}  
else  
{  
 result = b;  
 result->next = SortedMerge(a, b->next);  
}

① 다시 합칠 데이터가 없는 경우 나머지를 반환한다

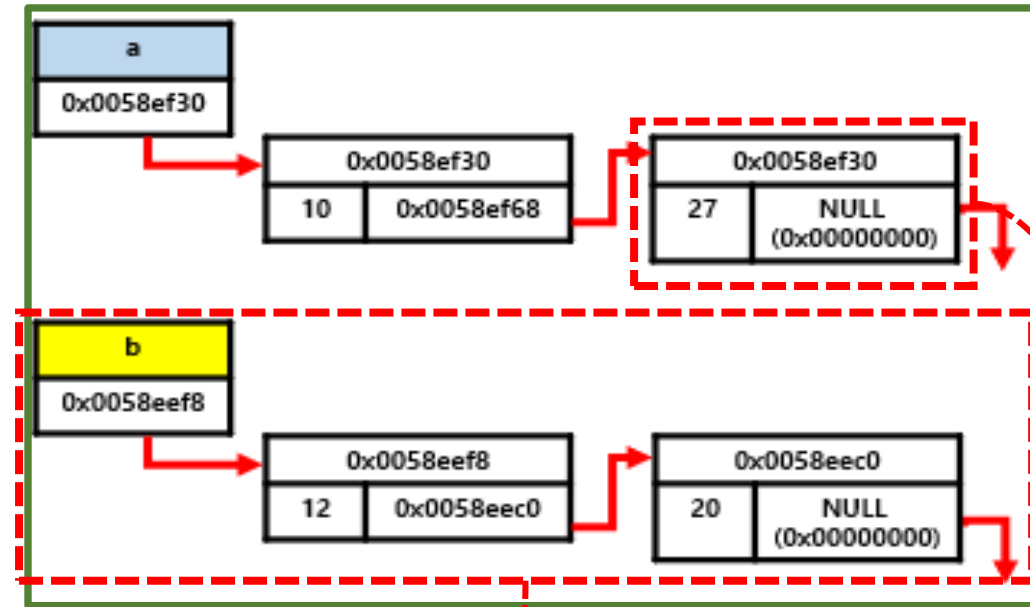
② 매개변수 데이터들의 크기 비교 후, 결과 노드로 메모리 저장 (처리할 데이터가 있으면 재귀호출로 수행)

## 4. 합병 정렬 - 구현 (selection)

```
if (a->data <= b->data)
{
    ② result = a;
    result->next = SortedMerge(a->next, b);
}
else ③
{
    result = b;
    result->next = SortedMerge(a, b->next);
}
return(result);
```



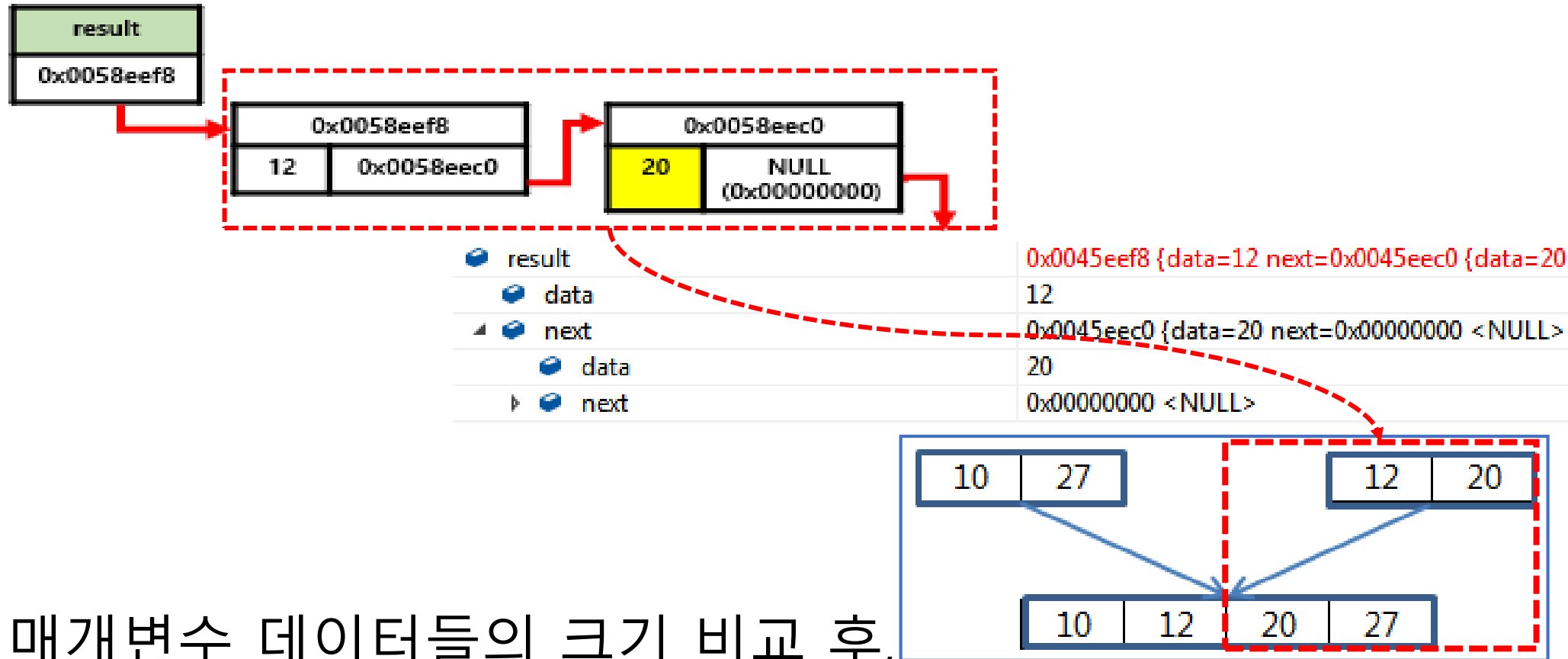
③ 다시 합치기 위해  
합병하는 함수 호출



```
struct Node* SortedMerge(struct Node* a, struct Node* b)
```

## 4. 합병 정렬 - 구현 (selection)

② `result = b;`



② 매개 변수 데이터들의 크기 비교 후,  
결과 노드로 메모리 저장



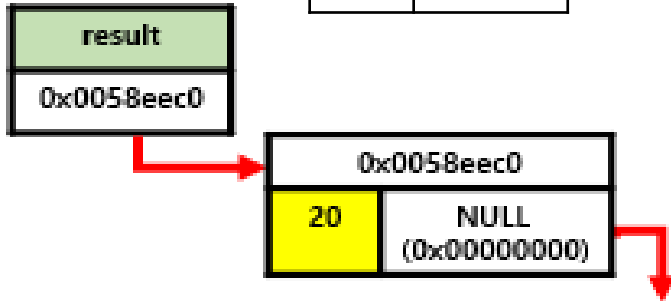
## 4. 합병 정렬 - 구현 (selection)

```
result->next = SortedMerge(a, b->next);
```

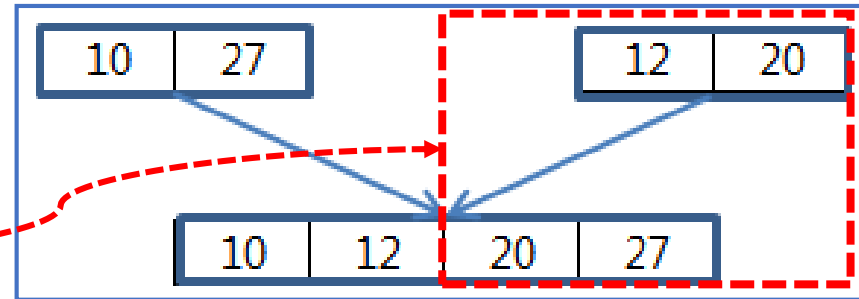
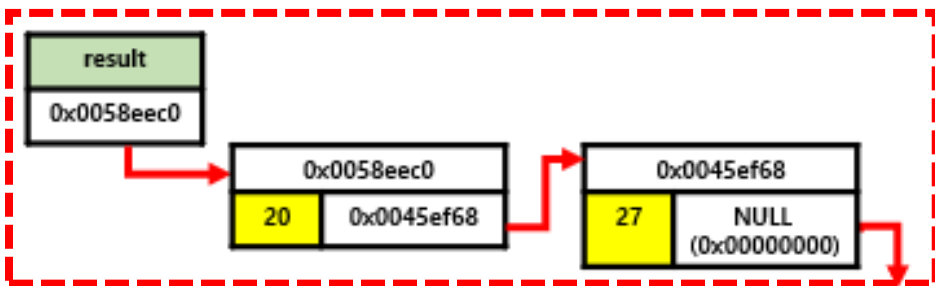
다시 SortedMerge 함수를 호출한다 (27과 20을 비교 !!!)

③ 다시 합치기 위해  
합병하는 함수 호출

a	27
b	20



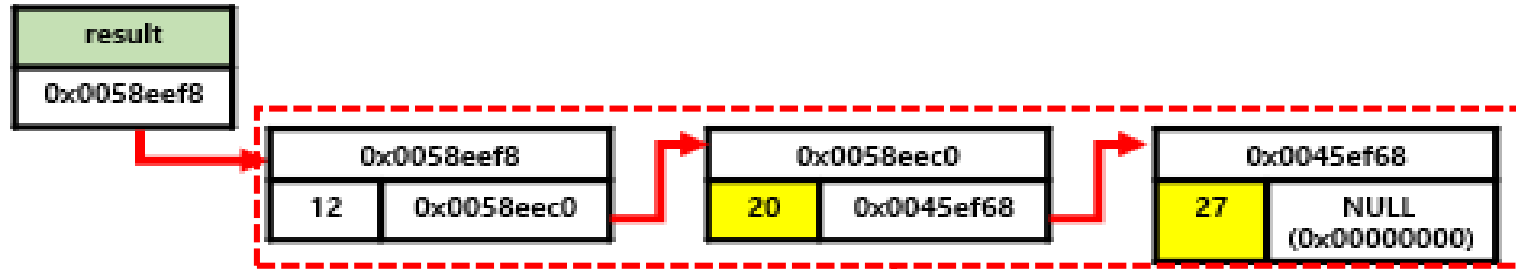
다시 SortedMerge 함수를 호출한다 (27과 null을 비교 !!!)  
27을 반환한다.



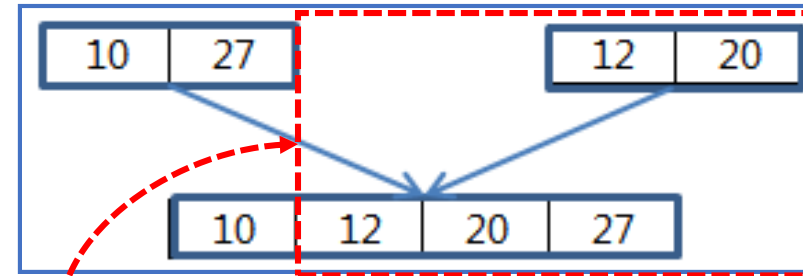
이 결과값을 다시 return (정렬해야 할 노드들 중 큰 값부터 정렬되어 반환하고 있다).

## 4. 합병 정렬 - 구현 (selection)

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
```



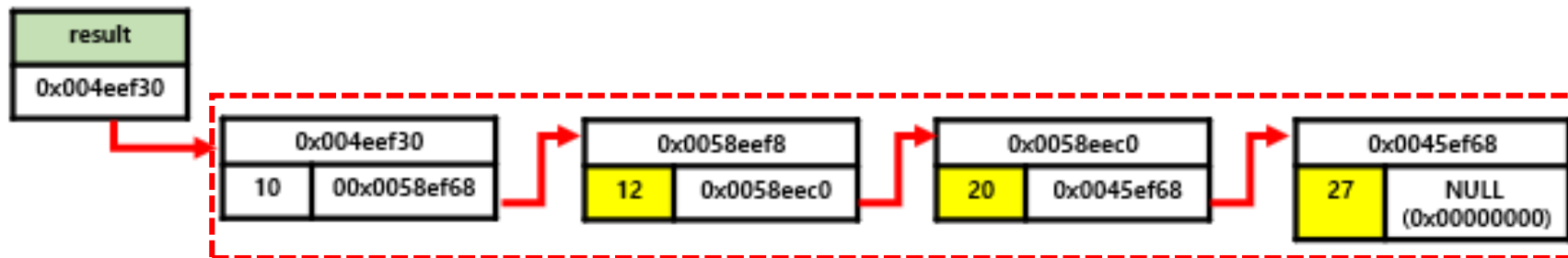
이름	값
SortedMerge이(가) 반환되었습니다.	0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
a	0x0045ef68 {data=27 next=0x00000000 <NULL>}
data	27
next	0x00000000 <NULL>
b	0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}}
data	12
next	0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	20
next	0x0045ef68 {data=27 next=0x00000000 <NULL>}
data	27
next	0x00000000 <NULL>
result	0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}}
data	12
next	0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	20
next	0x0045ef68 {data=27 next=0x00000000 <NULL>}
data	27
next	0x00000000 <NULL>



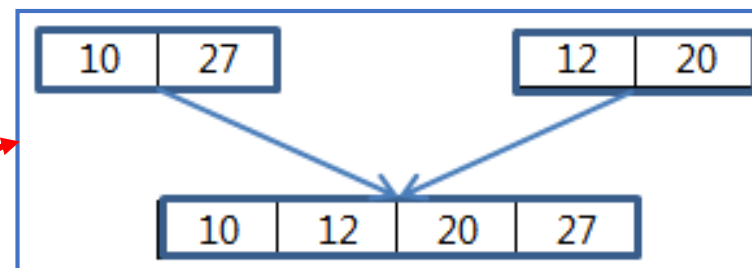
③ 다시 합치기 위해  
합병하는 함수 호출

# 4. 합병 정렬 - 구현 (selection)

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
```



이름	값
a	0x0045ef30 {data=10 next=0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	10
next	0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	12
next	0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
b	0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	12
next	0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	20
next	0x0045ef68 {data=27 next=0x00000000 <NULL>}
result	0x0045ef30 {data=10 next=0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	10
next	0x0045eef8 {data=12 next=0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	12
next	0x0045eec0 {data=20 next=0x0045ef68 {data=27 next=0x00000000 <NULL>}}
data	20
next	0x0045ef68 {data=27 next=0x00000000 <NULL>}
data	27
next	0x00000000 <NULL>



③ 다시 합치기 위해  
합병하는 함수 호출

```
Not Sorted Linked List is:
27 10 12 20 25 13 15
Sorted Linked List is:
10 12 13 15 20 25 27
```

## 4. 합병 정렬 (Merge Sort)

---

- 분할 정복 방식 (Divide and Conquer)

- ① 분할 (Divide): 배열을 같은 크기를 가진  
2개의 부분 배열로 분할 =>  $O(1)$
- ② 정복 (Conquer): 부분 배열을 정렬  
부분 배열의 크기가 충분히 작지  
않을 경우, 재귀 호출을 이용하여  
다시 분할 정복 방법 적용 =>  $O(n/2)$
- ③ 결합 (Combine): 정렬된 부분배열을  
하나의 배열에 통합 =>  $O(n)$

# 4. 합병 정렬 복잡도 분석

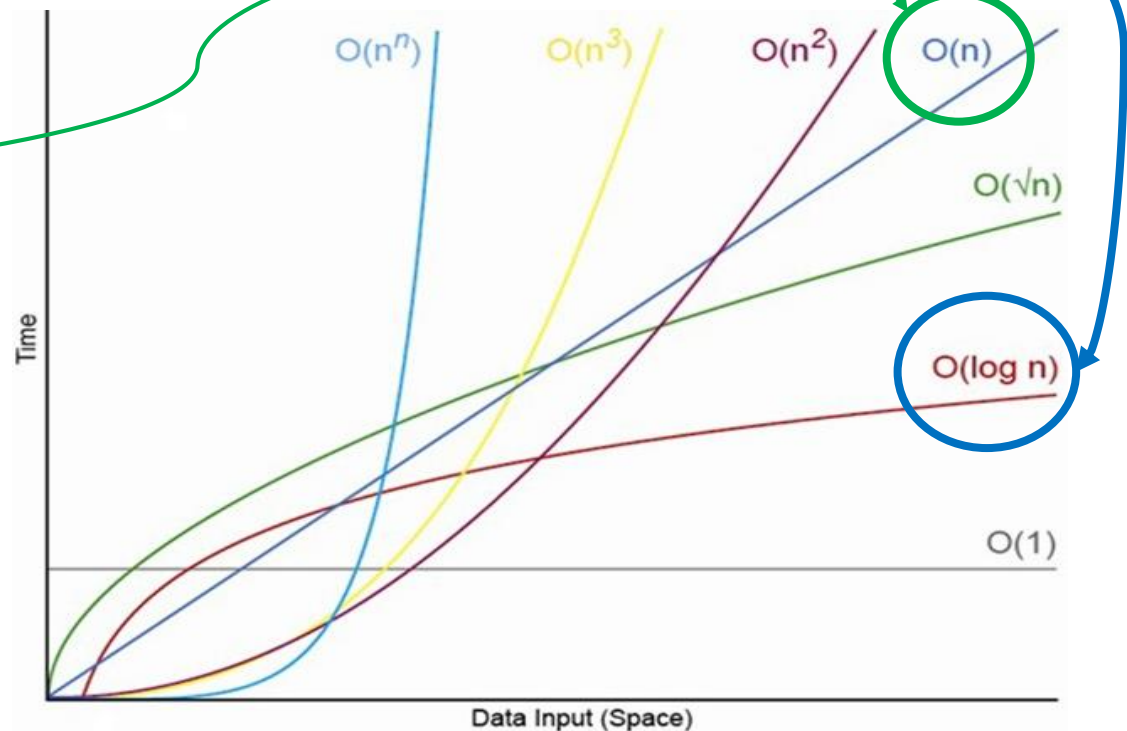
- 합병 정렬의 시간 복잡도

합병 정렬의 Divide 단계에서 분할되는 깊이가  $\log N$ 에 비례한다.

각 깊이 별 Divide가 수행되어 합병 해야되는 배열의 수는 많아지지만, 총 원소의 수는 똑같다. (N개)

따라서 각 깊이 별 수행되는 merge의 시간 복잡도는  $O(N)$ 이 된다.

( $\log N$ 개만큼) 수행하기 때문에  
시간 복잡도는  $O(N * \log N)$ 이 된다.



## 5. 퀵 정렬 (quick sort)

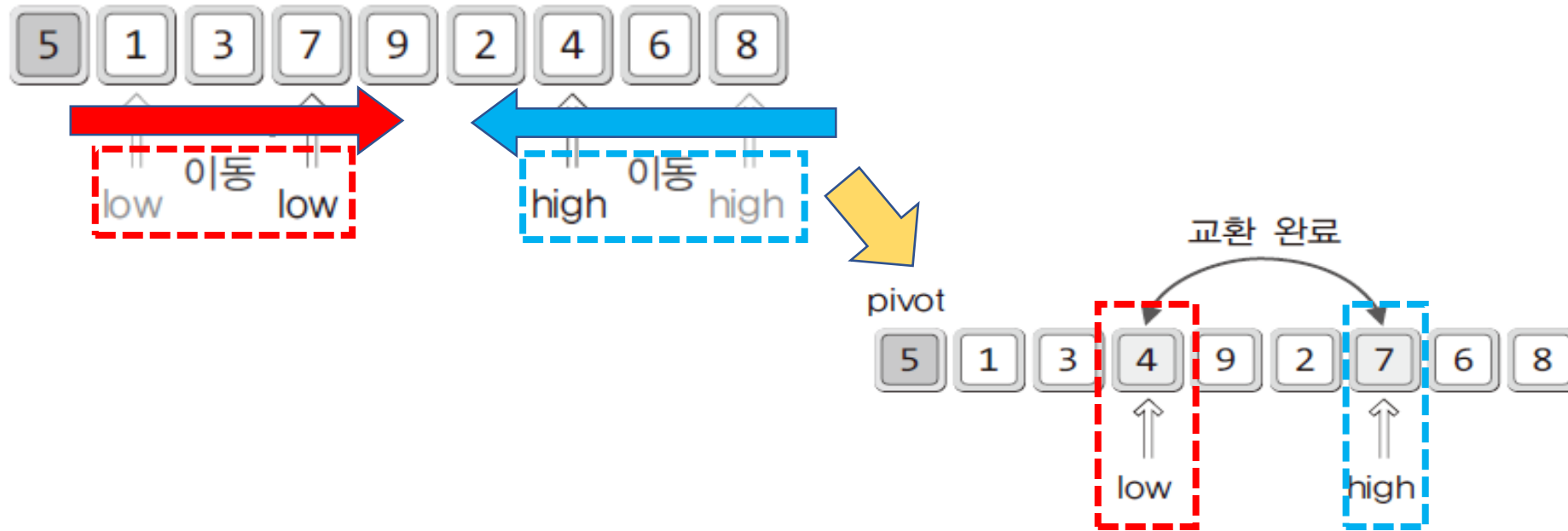
---



- 리스트에서 임의의 피벗(pivot)을 선택하고 이 피벗을 기준으로 작은 값은 왼쪽에 큰 값은 오른쪽에 이동시킨다.

이 과정을 더 이상 나눌 수 없을 때 까지 반복한다. (재귀)

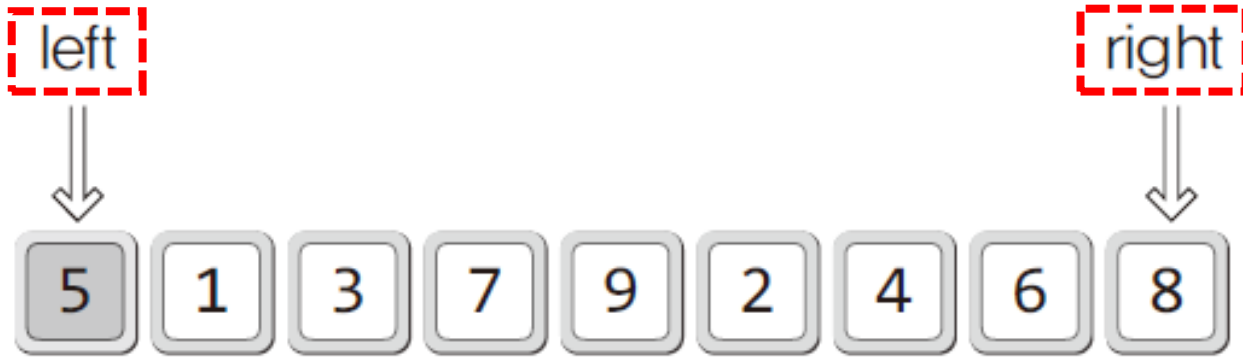
## 5. 퀵 정렬 (quick sort)



- 리스트에서 임의의 피벗(pivot)을 선택하고 이 피벗을 기준으로 작은 값은 왼쪽에, 큰 값은 오른쪽에 이동시킨다.

이 과정을 더 이상 나눌 수 없을 때 까지 반복한다. (재귀)

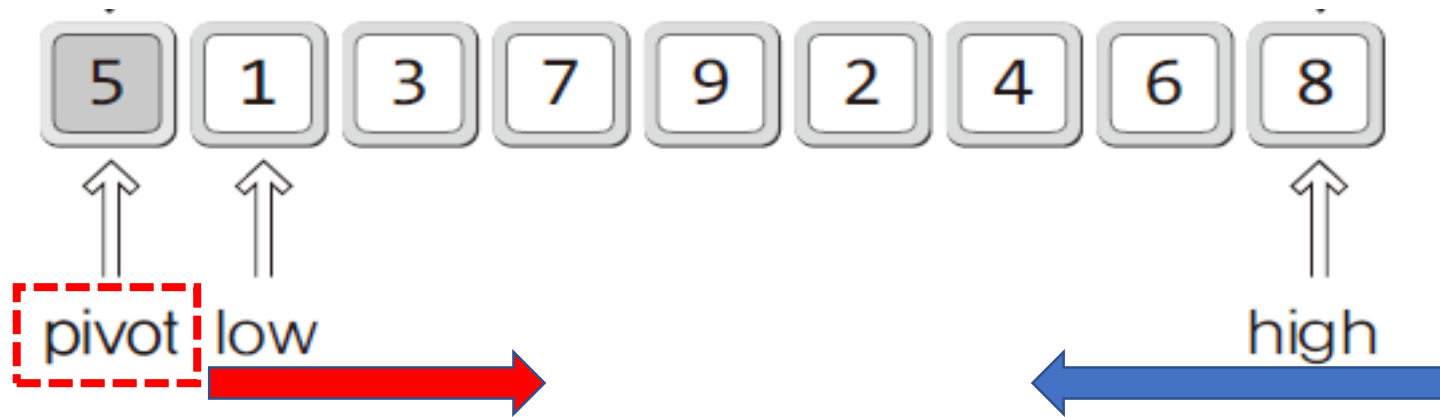
## 5. 퀵 정렬 (quick sort) - 용어



- 퀵 정렬에서 재귀호출을 수행하거나 call back 할 때 시작과 종료위치를 갖는다. (고정)
  - right : 정렬할 대상(배열)의 가장 오른쪽 위치
  - left : 정렬할 대상(배열)의 가장 왼쪽 위치

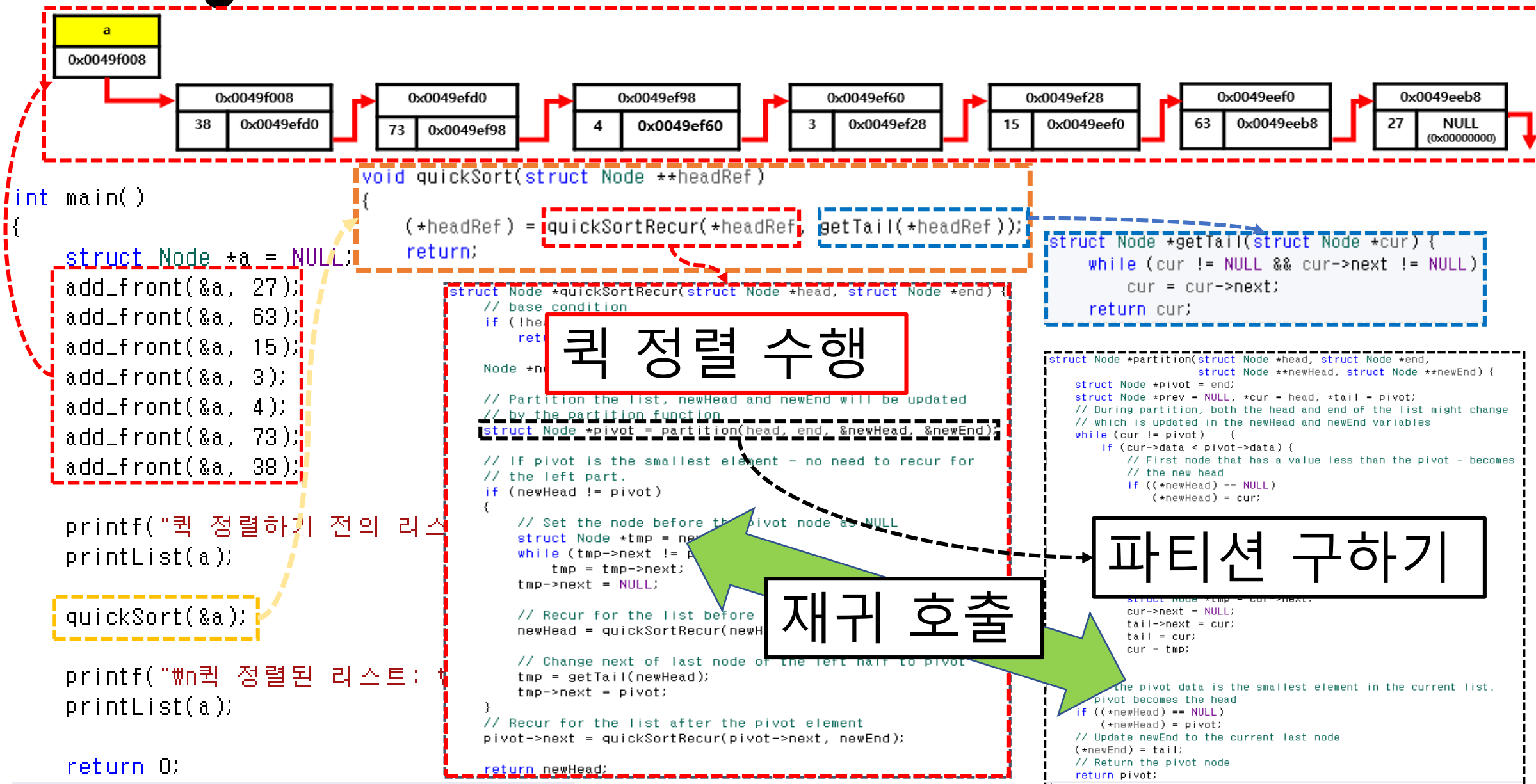


## 5. 퀵 정렬 (quick sort) - 용어



- 피벗(pivot) : 비교를 위해 기준(중심)이 되는 데이터
- low : 피벗을 제외한 왼쪽 위치  
(피벗과 비교 후, 오른쪽으로 이동)
- high : 피벗을 제외한 오른쪽 위치  
(피벗과 비교 후, 왼쪽으로 이동)

# 5. 퀵 정렬 - 구현 (selection)



## 5. 퀵 정렬 - 구현 (selection)

```
void quickSort(struct Node **headRef)
{
```

```
    (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
    return;
```

퀵 정렬 수행

맨 끝 노드를 구한다

이름	값
a	0x0049f008 {data=38 next=0x0049efd0 {
data	38
next	0x0049efd0 {data=73 next=0x0049ef98 {
data	73
next	0x0049ef98 {data=4 next=0x0049ef60 {d
data	4
next	0x0049ef60 {data=3 next=0x0049ef28 {d
data	3
next	0x0049ef28 {data=15 next=0x0049eef0 {c
data	15
next	0x0049eef0 {data=63 next=0x0049eeb8 {
data	63
next	0x0049eeb8 {data=27 next=0x00000000
data	27
next	0x00000000 <NULL>

## 5. 퀵 정렬 - 구현 (selection)

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {  
    // base condition  
    if (!head || head == end)  
        return head;  
  
    Node *newHead = NULL, *newEnd = NULL;  
  
    // Partition the list, newHead and newEnd will be updated  
    // by the partition function  
    ① struct Node *pivot = partition(head, end, &newHead, &newEnd);  
  
    // If pivot is the smallest element - no need to recur for  
    // the left part.  
    if (newHead != pivot)  
    {  
        // Set the node before the pivot node as NULL  
        struct Node *tmp = newHead;  
        while (tmp->next != pivot)  
            tmp = tmp->next;  
        tmp->next = NULL;  
  
        // Recur for the list before pivot  
        newHead = quickSortRecur(newHead, tmp);  
  
        // Change next of last node of the left half to pivot  
        tmp = getTail(newHead);  
        tmp->next = pivot;  
    }  
  
    // Recur for the list after the pivot element  
    pivot->next = quickSortRecur(pivot->next, newEnd);  
  
    return newHead;  
}
```

① 퀵 정렬 함수 수행  
피벗을 구하기 위한  
파티션 함수 호출

## 5. 퀵 정렬 - 구현 (selection)

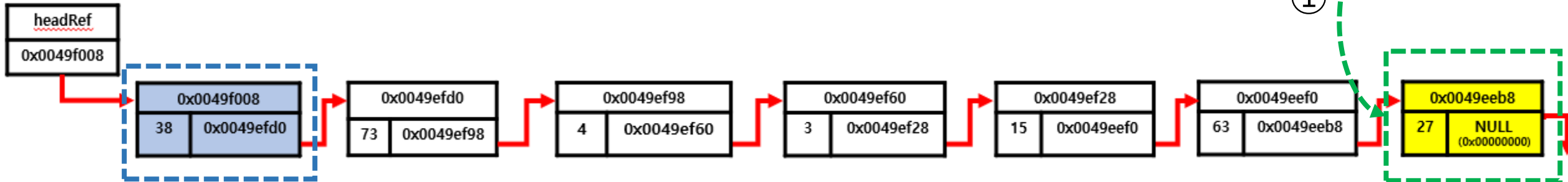
```
struct Node* partition(struct Node* head, struct Node* end, struct Node** newHead, struct Node** newEnd)
```

```
{  
    struct Node* pivot = end;  
    struct Node* prev = NULL;  
    struct Node* cur = head;  
    struct Node* tail = pivot;  
    while (cur != pivot) {  
        if (cur->val < pivot->val) {  
            prev = cur;  
            swap(&cur->val, &prev->val);  
        }  
        cur = cur->next;  
    }  
    swap(&cur->next, &tail->next);  
    swap(&cur, &tail);  
    *newHead = head;  
    *newEnd = tail;  
    return cur;  
}
```

파티션 구하기

피벗

①



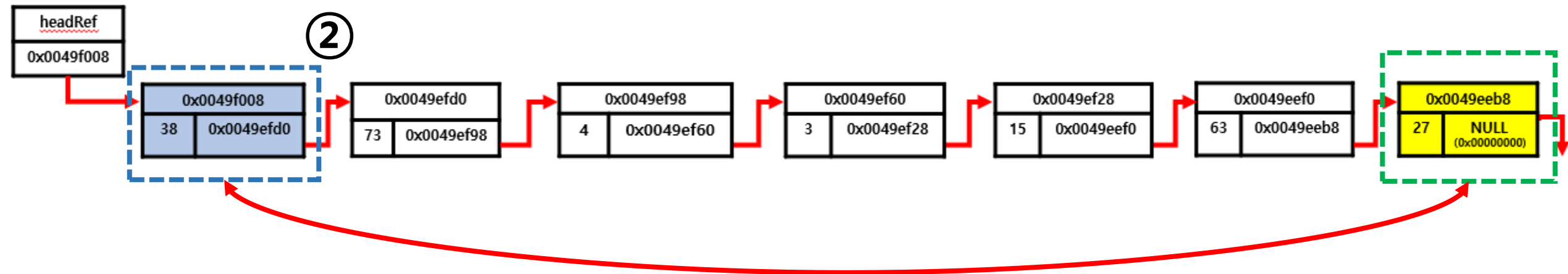
① 기준점이 되는 피벗을 맨 끝 노드를 기준으로 지정한다.

## 5. 퀵 정렬 - 구현 (selection)

```
struct Node* partition(struct Node* head, struct Node* end, struct Node** newHead, struct Node** newEnd)
```

```
{  
    struct Node* pivot = end;  
    struct Node* prev = NULL;  
    struct Node* cur = head;  
    struct Node* tail = pivot;  
    while (cur != pivot) {  
        if (cur->data < pivot->data) {  
            swap(&cur->data, &pivot->data);  
            swap(&cur->next, &prev->next);  
            prev = cur;  
            cur = cur->next;  
        } else {  
            cur = cur->next;  
        }  
    }  
    *newHead = head;  
    *newEnd = pivot;  
}
```

값 비교



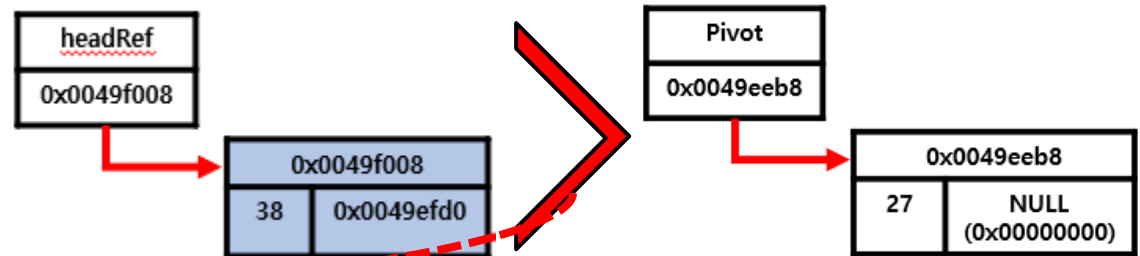
② 피벗과 맨 앞 노드의 값을 비교한다.

# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd to the current last node
    (*newEnd) = tail;
    // Return the pivot node
    return pivot;
}
```

값 비교 (38 vs 27)



② 피벗과 맨 앞 노드의 값을 비교한다  
38과 피벗(27) 비교 시,  
Else 수행

# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

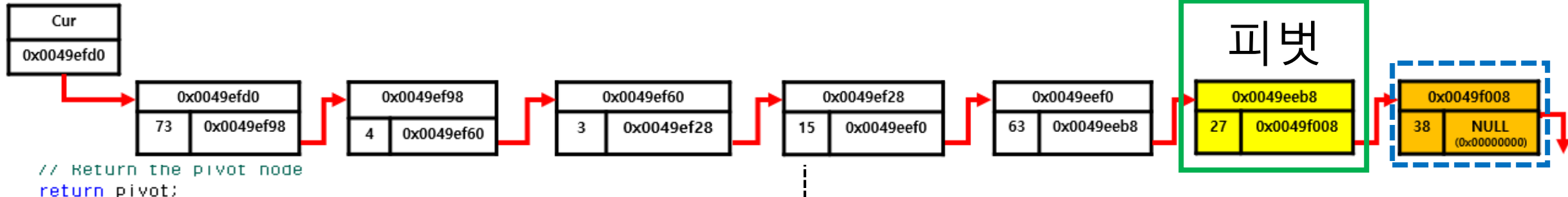
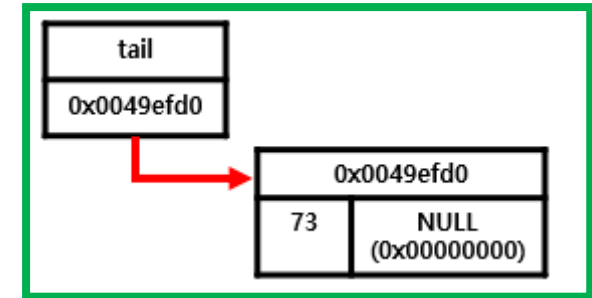
            prev = cur;
            cur = cur->next;
        }
        else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    return pivot;
}
```

④

```
else { // If cur node is greater than pivot
    // Move cur node to next of tail, and change tail
    if (prev)
        prev->next = cur->next;
    struct Node *tmp = cur->next;
    cur->next = NULL;
    tail->next = cur;
    tail = cur;
    cur = tmp;
}
```

값 비교 (38 vs 27)

④ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동



피벗



# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

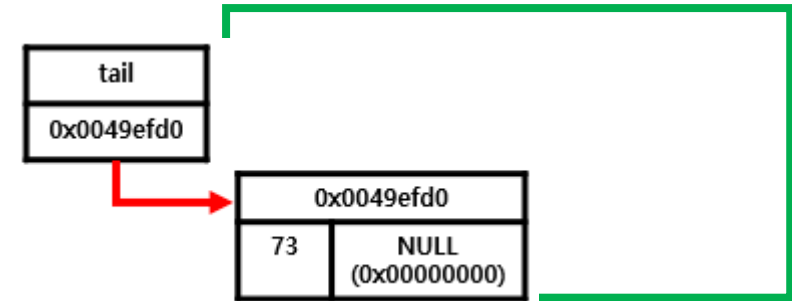
            prev = cur;
            cur = cur->next;
        }
        else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
}
```

④

```
else { // If cur node is greater than pivot
    // Move cur node to next of tail, and change tail
    if (prev)
        prev->next = cur->next;
    struct Node *tmp = cur->next;
    cur->next = NULL;
    tail->next = cur;
    tail = cur;
    cur = tmp;
}
```

값 비교 (73 vs 27)

④ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동



피벗



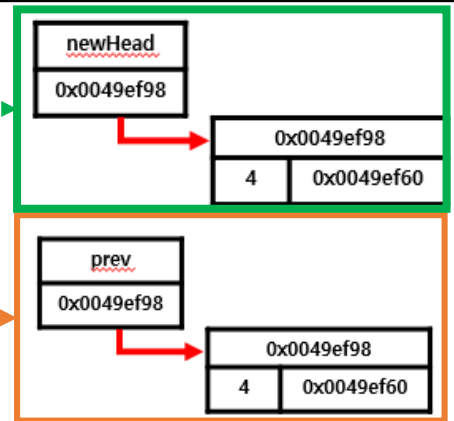
# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        ⑤ if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

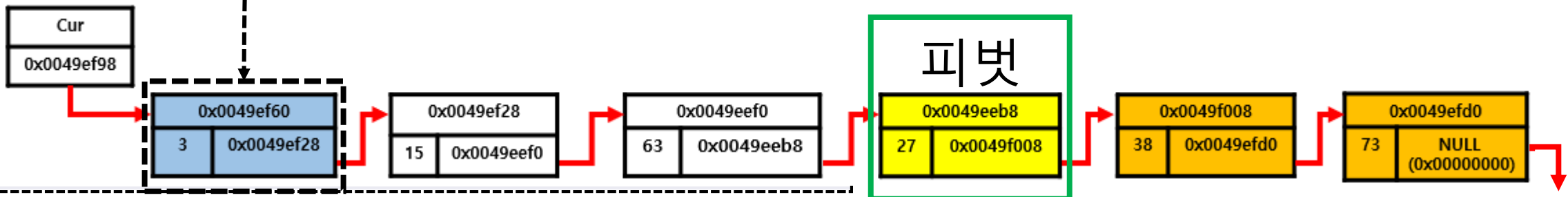
            prev = cur;
            cur = cur->next;
        } else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
}
```

값 비교 (4 vs 27)

⑤ 비교 후, 피벗보다 작은 값은 새로운 노드에 저장 후 이동



prev	0x0066ef98 {data=4 next=0x0066e
data	4
next	0x0066ef60 {data=3 next=0x0066e



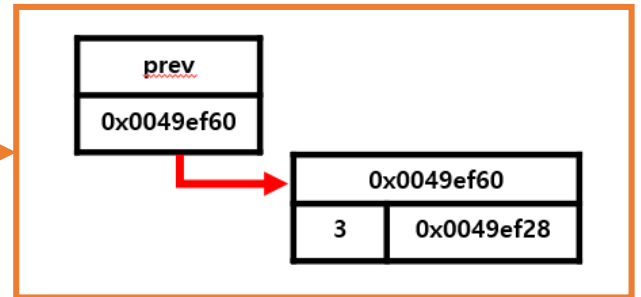
# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        ⑤ if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

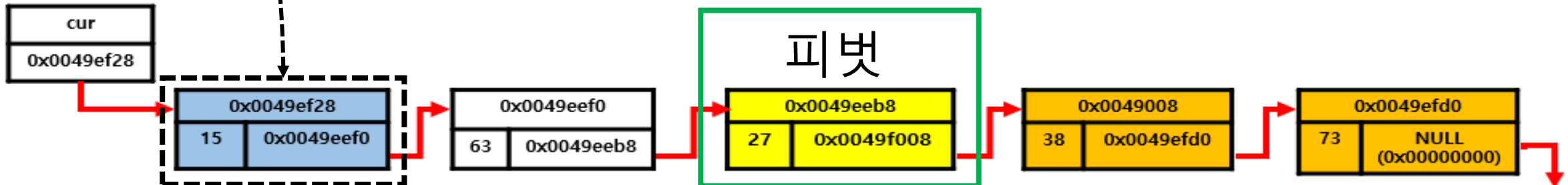
            prev = cur;
            cur = cur->next;
        } else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    return pivot;
}
```

값 비교 (3 vs 27)

⑤ 비교 후, 피벗보다 작은 값은 새로운 노드에 저장 후 이동



prev	0x0066ef60 {data=3 next=0x0066e
data	3
next	0x0066ef28 {data=15 next=0x0066



# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        ⑤ if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        } else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    // If the pivot becomes the new head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd
    (*newEnd) = tail;
    // Return the pivot
    return pivot;
}
```

값 비교 (15 vs 27)

⑤ 비교 후, 피벗보다 작은 값은 새로운 노드에 저장 후 이동



prev	0x0066ef28 {data=15 next=0x0066ef28}
data	15
next	0x0066eeef0 {data=63 next=0x0066eeef0}
data	63
next	0x0066eeb8 {data=27 next=0x0066eeb8}

# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        } else { // If cur
            // Move cur node to next or tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    // If
    // piv
    if ((*newE
    // Ret
    return pivot;
}
```

값 비교 (63 vs 27)

⑥ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동

prev	0x0066ef28 {data=15 next=0x0066eef0}
data	15
next	0x0066eef0 {data=63 next=0x0066eeb8}
data	63
next	0x0066eeb8 {data=27 next=0x0066f008}



prev	0x0066ef28 {data=15 next=0x0066eeb8}
data	15
next	0x0066eeb8 {data=27 next=0x0066f008}
data	27
next	0x0066f008 {data=38 next=0x0066f008}

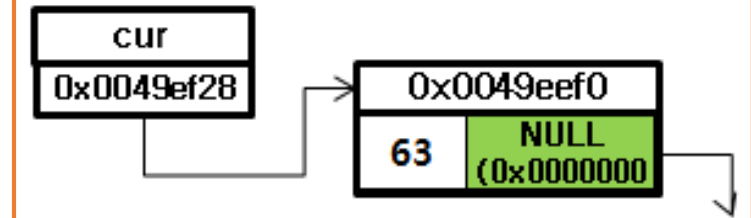
# 5. 퀵 정렬 - 구현 (selection)

값 비교 (63 vs 27)

⑥ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else { // If cur node is greater than pivot
            // Move cur node to next of tail,
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd to the current last node
    (*newEnd) = tail;
    // Return the pivot node
    return pivot;
}
```



cur	0x0066eef0 {data=63 next=0x0000
data	63
next	0x00000000 <NULL>

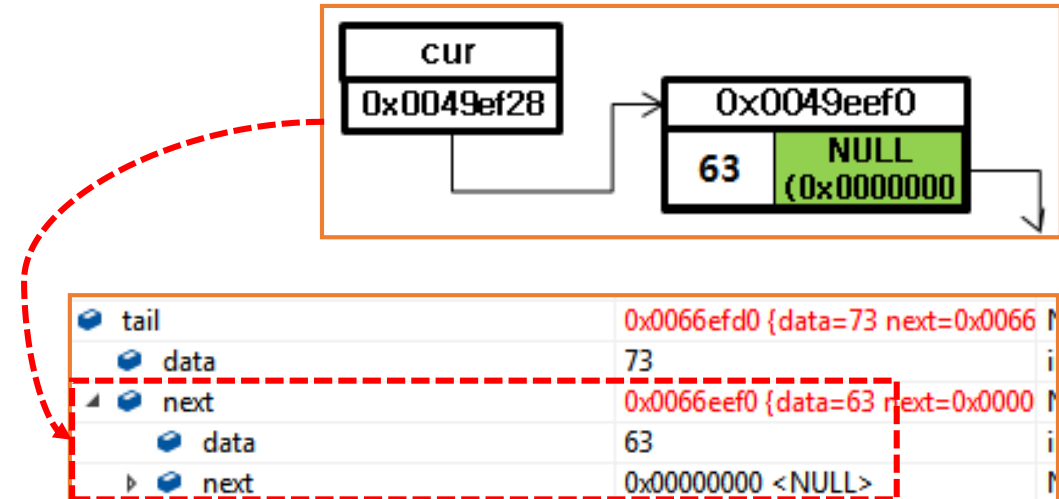
# 5. 퀵 정렬 - 구현 (selection)

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        ⑥ else { // If cur node is greater than pivot
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct Node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd to the current last node
    (*newEnd) = tail;
    // Return the pivot node
    return pivot;
}
```

값 비교 (63 vs 27)

⑥ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동

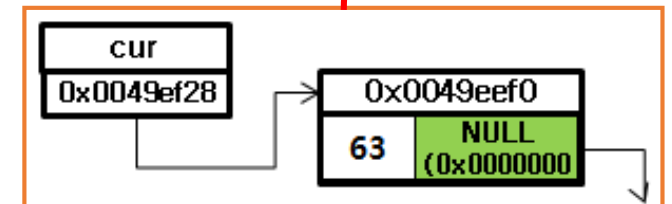
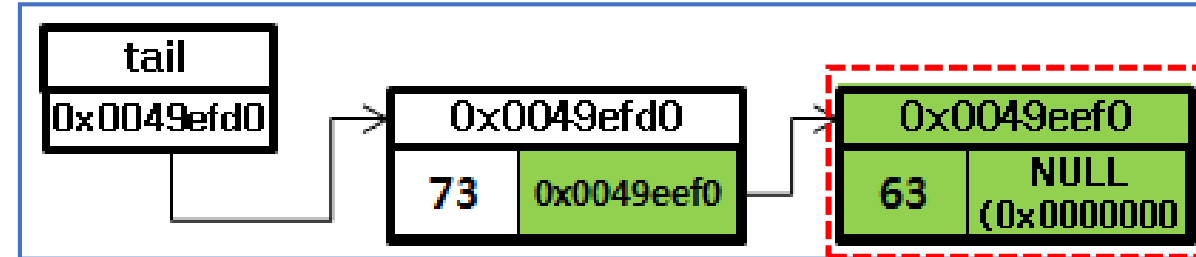
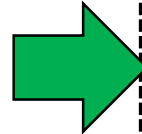


# 5. 퀵 정렬 - 구현 (selection)

값 비교 (63 vs 27)

⑥ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the pivot - becomes
            // tail
            if (prev)
                prev->next = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
        else {
            // If the pivot data is the smallest element in the current list,
            // pivot becomes the head
            if ((*newHead) == NULL)
                (*newHead) = pivot;
            // Update newEnd to the current last node
            (*newEnd) = tail;
            // Return the pivot node
            return pivot;
        }
    }
}
```



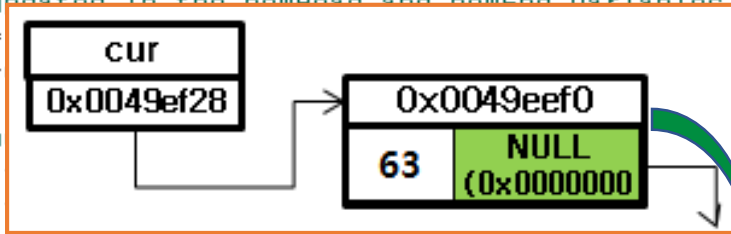


# 5. 퀵 정렬 - 구현 (selection)

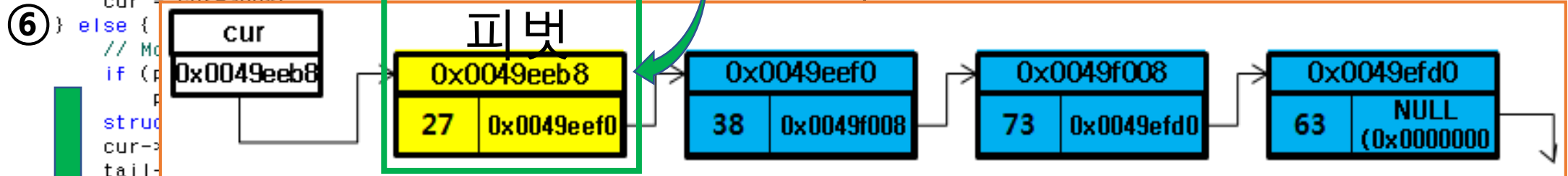
값 비교 (63 vs 27)

⑥ 비교 후, 피벗보다 큰 값은  
피벗 다음 노드 위치로 이동

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // Find the node to be swapped with the pivot
            // Swap the data of the node with the pivot
            if ((cur->data < pivot->data) && (prev == NULL || prev->data < cur->data)) {
                // Swap the data of the node with the pivot
                swap(&cur->data, &pivot->data);
            }
            prev = cur;
        } else {
            // Move the node to the right of the pivot
            cur->next = tail->next;
            tail->next = cur;
            cur = prev;
        }
    }
    // If the pivot data is the smallest element in the current list
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd to the current last node
    (*newEnd) = tail;
    // Return the pivot node
    return pivot;
}
```



not - becomes



cur	0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL> } } } }
data	27
next	0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL> } } }
data	38
next	0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL> } }
data	73
next	0x0051eef0 {data=63 next=0x00000000 <NULL> }
data	63
next	0x00000000 <NULL>

# 5. 퀵 정렬 - 구현 (selection)

값 비교 (조건 종료)

⑦ 피벗과 현재 노드와 같은 값이므로 반복 종료

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;
    // During partition,
    // which is updated
    ⑦ while (cur != pivot)
        if (cur->data < pivot->data)
            // First node
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        } else { // If cur
            // Move cur node
            if (prev)
                prev->next = cur->next;
            struct Node *tm;
            cur->next = pivot->next;
            tail->next = cur;

            pivot->next = cur;
            cur = pivot;
        }
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd to the current last node
    (*newEnd) = tail;
    // Return the pivot node
    return pivot;
}
```

Node	Address	data	next
pivot	0x0049eeb8	27	0x0049eef0
cur	0x0049eeb8	27	0x0049eef0
	0x0049eef0	38	0x0049f008
	0x0049f008	73	0x0049efd0
	0x0049efd0	63	NULL (0x00000000)

Variable	Value
cur	0x0049eeb8 {data=27 next=0x0049f008 {data=38 next=0x0049efd0 {data=63 next=0x00000000 <NULL>}}}

Variable	Value
cur	0x0049eeb8 {data=27 next=0x0049f008 {data=38 next=0x0049efd0 {data=63 next=0x00000000 <NULL>}}}

# 5. 퀵 정렬 - 구현 (selection)

파티션 함수 비교 후

⑧ 새로운 노드를 마지막 노드로 지정하여 저장

```
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd) {
    struct Node *pivot = end;

    tail = pivot;
    // If the list might change
    // newEnd variables

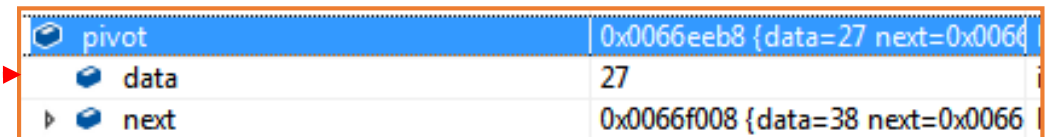
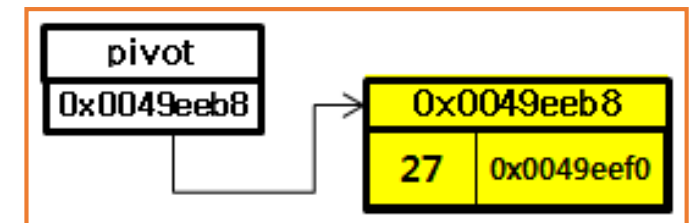
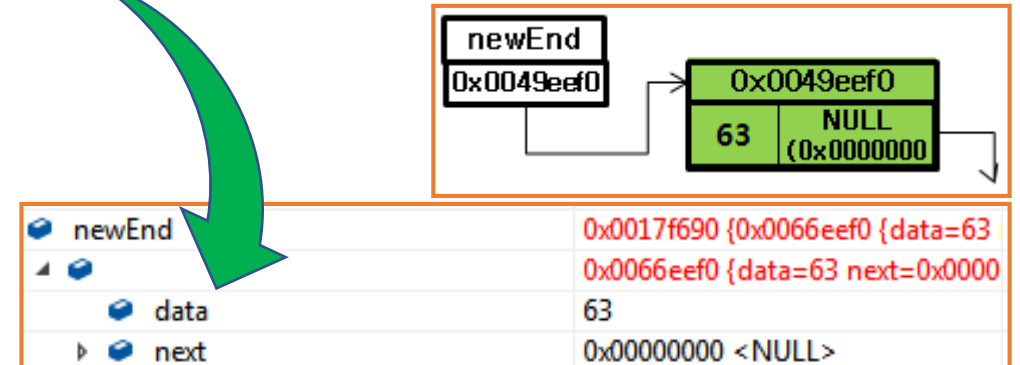
    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;
    // Update newEnd to the current last node
    (*newEnd) = tail;
    // Return the pivot node
    return pivot;
}
```

Diagram illustrating the state of the list during the partitioning process. The **tail** pointer (0x0049eef0) points to a node with data 63 and next pointer NULL (0x00000000). The **newEnd** pointer (0x0049eef0) also points to this node. The **newHead** pointer (0x0017f690) points to a node with data 63 and next pointer 0x0066eef0. The **tail** pointer is updated to point to the current node (0x0049eef0).

Diagram illustrating the state of the list during the partitioning process. The **newEnd** pointer (0x0049eef0) points to a node with data 63 and next pointer NULL (0x00000000). The **newHead** pointer (0x0017f690) points to a node with data 63 and next pointer 0x0066eef0. The **tail** pointer is updated to point to the current node (0x0049eef0).

Diagram illustrating the state of the list during the partitioning process. The **newEnd** pointer (0x0049eef0) points to a node with data 63 and next pointer NULL (0x00000000). The **newHead** pointer (0x0017f690) points to a node with data 63 and next pointer 0x0066eef0. The **tail** pointer is updated to point to the current node (0x0049eef0).

Diagram illustrating the state of the list during the partitioning process. The **newEnd** pointer (0x0049eef0) points to a node with data 63 and next pointer NULL (0x00000000). The **newHead** pointer (0x0017f690) points to a node with data 63 and next pointer 0x0066eef0. The **tail** pointer is updated to point to the current node (0x0049eef0).



## 5. 퀵 정렬 - 구현 (selection)

### 퀵 정렬 수행

- ① 피벗까지 위치를 조정  
배열로 고려 시,  
피벗의 이전 데이터를  
다시 파티션 함수 호출

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {  
    // base condition  
    if (!head || head == end)  
        return head;  
  
    Node *newHead = NULL, *newEnd = NULL;  
  
    // Partition the list, newHead and newEnd will be updated  
    // by the partition function  
    struct Node *pivot = partition(head, end, &newHead, &newEnd);  
  
    // If pivot is the smallest element - no need to recur for  
    // the left part.  
    if (newHead != pivot)  
    {  
        // Set the node before the pivot node as NULL  
        struct Node *tmp = newHead;  
        while (tmp->next != pivot)  
            tmp = tmp->next;  
        tmp->next = NULL;  
  
        // Recur for the list before pivot  
        newHead = quickSortRecur(newHead, tmp);  
  
        // Change next of last node of the left half to pivot  
        tmp = getTail(newHead);  
        tmp->next = pivot;  
    }  
    // Recur for the list after the pivot element  
    pivot->next = quickSortRecur(pivot->next, newEnd);  
  
    return newHead;  
}
```

# 5. 퀵 정렬 - 구현 (selection)

퀵 정렬 수행

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {  
    if (newHead != pivot)  
    {  
        // Set the node before the pivot node as NULL  
        struct Node *tmp = newHead;  
        while (tmp->next != pivot)  
            tmp = tmp->next;  
        tmp->next = NULL;  
    }  
}
```

② 피벗 이전 노드까지  
노드의 위치를 탐색

newHead	0x0051ef98 {data=4 next=0x0051ef60 {data=3 next=0x0051ef28 {data=15 next=0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	4	int
next	0x0051ef60 {data=3 next=0x0051ef28 {data=15 next=0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	3	int
next	0x0051ef28 {data=15 next=0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	15	int
next	0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	27	int
next	0x0051f008 {data=38 next=0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	38	int
next	0x0051efd0 {data=73 next=0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	73	int
next	0x0051eef0 {data=63 next=0x00000000 <NULL>}}}}}}}	Node *
data	63	int
next	0x00000000 <NULL>	Node *

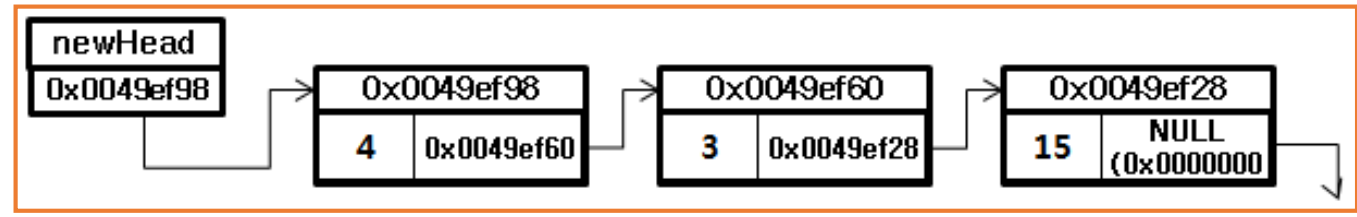
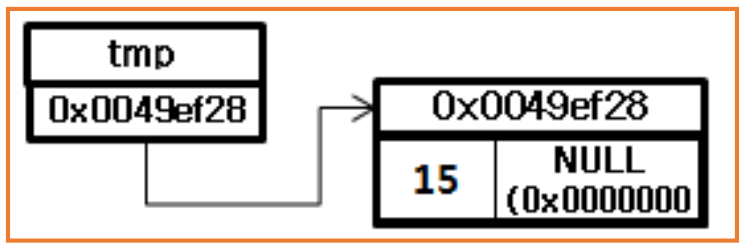
# 5. 퀵 정렬 - 구현 (selection)

## 퀵 정렬 수행

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {  
    if (newHead != pivot)  
    {  
        // Set the node before the pivot node as NULL  
        struct Node *tmp = newHead;  
        while (tmp->next != pivot)  
            tmp = tmp->next;  
        ③ tmp->next = NULL;  
    }  
}
```

③ 피벗 이전 노드의  
다음 노드 주소를  
NULL 처리로 정렬을 분리

newHead	0x0051ef98 {data=4 next=0x0051ef60 {data=3 next=0x0051ef28 {data=15 next=0x00000000 <NULL>}}	Node *
data	4	int
next	0x0051ef60 {data=3 next=0x0051ef28 {data=15 next=0x00000000 <NULL>}}	Node *
data	3	int
next	0x0051ef28 {data=15 next=0x00000000 <NULL>}	Node *
data	15	int
next	0x00000000 <NULL>	Node *



# 5. 퀵 정렬 - 구현 (selection)

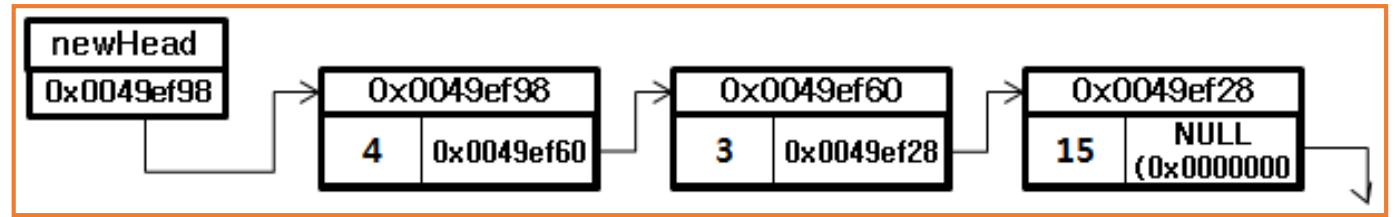
퀵 정렬 수행

④ 새로운 범위로 퀵 정렬을 다시 수행한다

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {  
    if (newHead != pivot)  
    {  
        // Set the node before the pivot node as NULL  
        struct Node *tmp = newHead;  
        while (tmp->next != pivot)  
            tmp = tmp->next;  
        tmp->next = NULL;
```

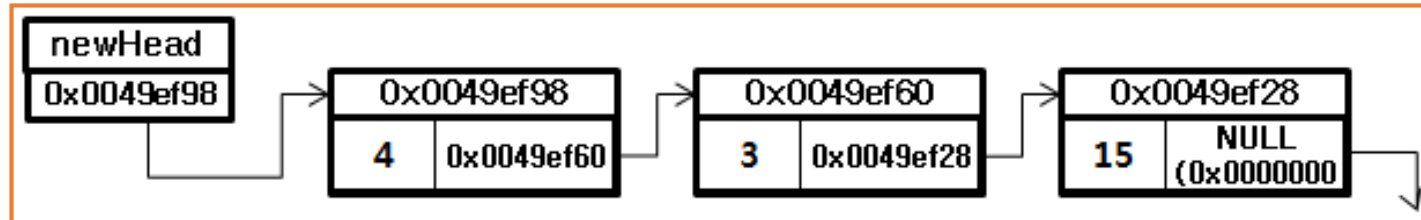
④ // Recur for the list before pivot  
newHead = quickSortRecur(newHead, tmp);

newHead	0x0051ef98 {data=4 next=0x0051ef60 {data=	Node *
data	4	int
next	0x0051ef60 {data=3 next=0x0051ef28 {data=	Node *
data	3	int
next	0x0051ef28 {data=15 next=0x00000000 <NL	Node *
data	15	int
next	0x00000000 <NULL>	Node *



# 5. 퀵 정렬 - 구현 (selection)

newHead = quickSortRecur(newHead, tmp); ⑤



퀵 정렬 수행

⑤ 피벗은 맨 우측 15를 기준으로 정렬 조건 불 충족 종료

피벗

15

if (newHead != pivot)

newHead	0x0051ef60	pivot	0x0051ef60
data	3	data	3
next	0x0051ef98	next	0x0051ef98

피벗

3

3

4



## 5. 퀵 정렬 - 구현 (selection)

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {
```

```
    if (newHead != pivot)
```

```
    {
```

```
        // Set the node before the pivot node as NULL
```

```
        struct Node *tmp = newHead;
```

```
        while (tmp->next != pivot)
```

```
            tmp = tmp->next;
```

```
        tmp->next = NULL;
```

```
        // Recur for the list before pivot
```

```
        newHead = quickSortRecur(newHead, tmp);
```

```
        // Change next of last node of the left half to pivot
```

```
        tmp = getTail(newHead);
```

```
        tmp->next = pivot;
```

```
    }
```

퀵 정렬 수행

⑥ 파티션 함수를 통해  
구한 노드의 마지막을  
정한 뒤 피벗의 끝 지정

⑥

## 5. 퀵 정렬 - 구현 (selection)

```
struct Node *quickSortRecur(struct Node *head, struct Node *end) {  
    // ...  
    if (newHead != pivot)  
    {  
        // Set the node before the pivot node as NULL  
        struct Node *tmp = newHead;  
        while (tmp->next != pivot)  
            tmp = tmp->next;  
        tmp->next = NULL;  
  
        // Recur for the list before pivot  
        newHead = quickSortRecur(newHead, tmp);  
  
        // Change next of last node of the left half to pivot  
        tmp = getTail(newHead);  
        tmp->next = pivot;  
    }  
    // Recur for the list after the pivot element  
    pivot->next = quickSortRecur(pivot->next, newEnd);  
  
    return newHead;  
}
```

퀵 정렬 수행

⑦ 재귀 호출을 통해  
나누어진 노드를  
다시 퀵 정렬을 했지만  
비교 대상이 같아서 종료.

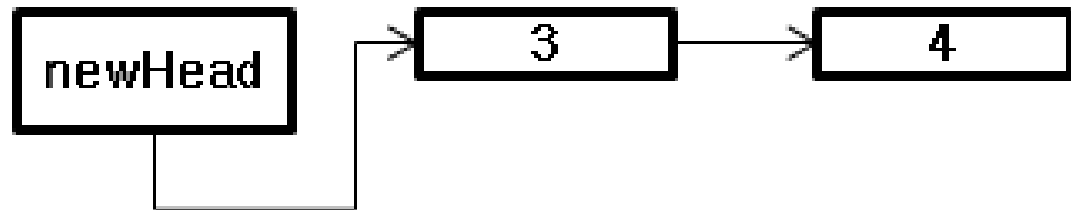
pivot->next

4

newEnd

4

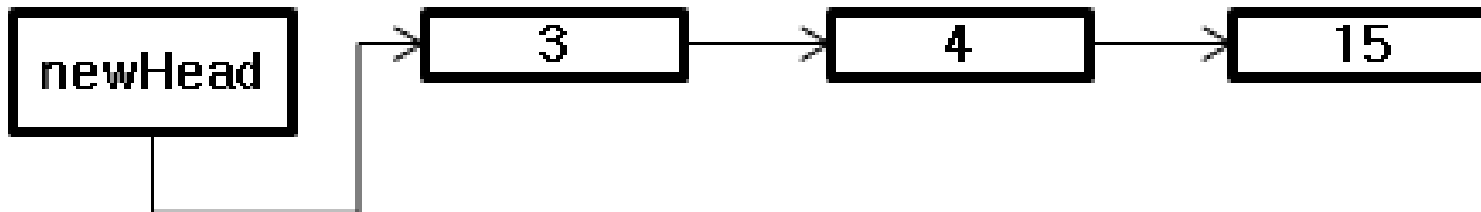
## 5. 퀵 정렬 - 구현 (selection)



newHead	0x0051ef60 {data=3 next=0x0051ef98 {data=
data	3
next	0x0051ef98 {data=4 next=0x00000000 <NULL>
data	4
next	0x00000000 <NULL>

퀵 정렬 수행

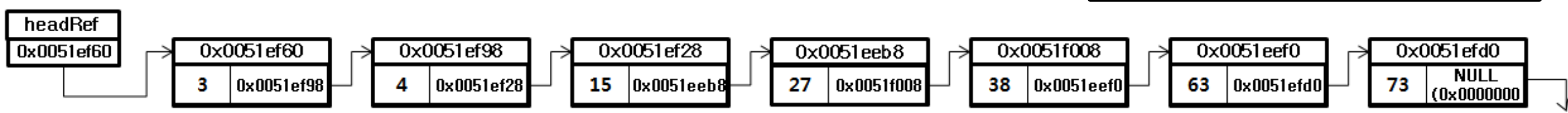
⑧ 재귀 호출을 하며  
퀵 정렬을 하면서  
노드의 정렬을 수행



newHead	0x0051ef60 {data=3 next=0x0051ef98 {data=
data	3
next	0x0051ef98 {data=4 next=0x0051ef28 {data=
data	4
next	0x0051ef28 {data=15 next=0x00000000 <NULL>
data	15
next	0x00000000 <NULL>

# 5. 퀵 정렬 - 구현 (selection)

퀵 정렬 완료



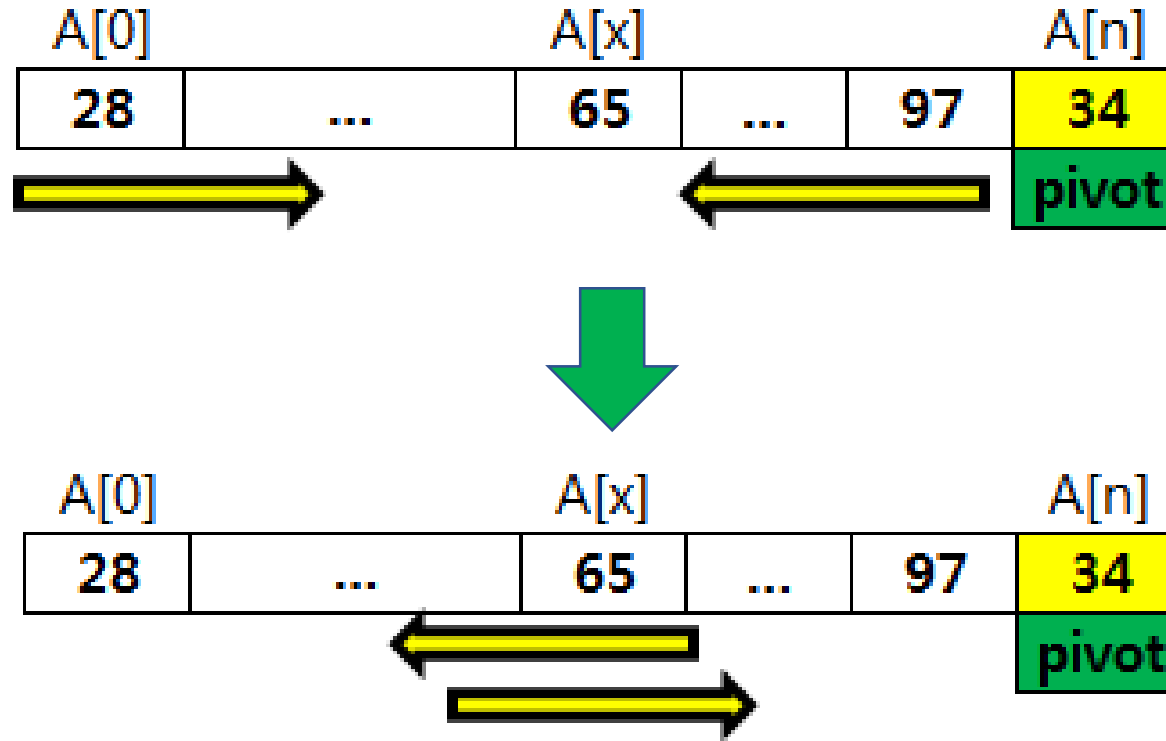
이름	값
headRef	0x0021fb6c {0x0051ef60 {data=3 next=0x0051ef98 {data=4 next=0x0051ef28 {data=15 next=0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051eef0 {data=63 next=0x0051efd0 {data=73 next=0x00000000 <NULL>}}
data	3
next	0x0051ef98 {data=4 next=0x0051ef28 {data=15 next=0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051eef0 {data=63 next=0x0051efd0 {data=73 next=0x00000000 <NULL>}}
data	4
next	0x0051ef28 {data=15 next=0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051eef0 {data=63 next=0x0051efd0 {data=73 next=0x00000000 <NULL>}}
data	15
next	0x0051eeb8 {data=27 next=0x0051f008 {data=38 next=0x0051eef0 {data=63 next=0x0051efd0 {data=73 next=0x00000000 <NULL>}}
data	27
next	0x0051f008 {data=38 next=0x0051eef0 {data=63 next=0x0051efd0 {data=73 next=0x00000000 <NULL>}}
data	38
next	0x0051eef0 {data=63 next=0x0051efd0 {data=73 next=0x00000000 <NULL>}}
data	63
next	0x0051efd0 {data=73 next=0x00000000 <NULL>}
data	73
next	0x00000000 <NULL>

퀵 정렬하기 전의 리스트:  
38 73 4 3 15 63 27

퀵 정렬된 리스트:  
3 4 15 27 38 63 73

## 5. 퀵 정렬 복잡도 분석

### ■ 원리



- 퀵 정렬의 알고리즘은 정리하면
  - 피벗(pivot)이 적당한 위치를 찾는 과정
  - 기준(피벗)을 통해 왼쪽/오른쪽을 정렬하는 과정

## 5. 퀵 정렬 복잡도 분석

### ■ 분석

A[0]		A[x]		A[n-1]	A[n]
28	...	34	...		65
					pivot

- 전체 n개의 원소를 탐색하고 partition을 한 뒤,  
34를 기준으로 왼쪽을 quick sort 수행  
오른쪽을 quick sort 수행 이를 수식으로 표현하면

$$T(n) = T(x) + T(n - (1 + x)) + n$$

- x+1번째 원소를 기준으로 2개로  
분할된다고 했을 때 배열에서는 A[x]가 기준

## 5. 퀵 정렬 복잡도 분석

- 최악의 경우

pivot								
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
1	2	3	4	5	6	7	8	9

								pivot
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
1	2	3	4	5	6	7	8	9

- 이미 배열이 정렬되어 피벗(pivot)이 배열의 한쪽 끝에 치우친 경우 분할/정복해서 작업하지 못하므로 최악의 경우를 보인다.

# 5. 퀵 정렬 복잡도 분석

## ■ 최악의 경우

pivot									
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	
1	2	3	4	5	6	7	8	9	

- 배열 0번 위치를 피벗으로 정한 경우 아래 수식에

$$T(n) = T(x) + T(n - (1 + x)) + n$$

x에 0을 대입하면  $T(n) = T(n - 1) + n$

$$T(n - 1) + n$$

$$= T(n - 2) + (n - 1) + n$$

= ...

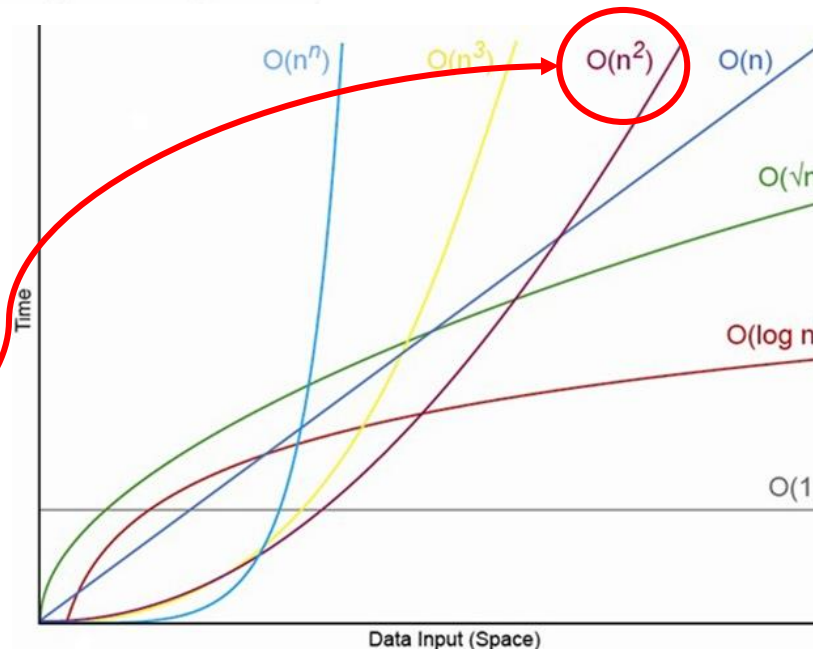
$$= T(n - a) + (n - a + 1) + \dots + n$$

= ...

$$= T(0) + 1 + \dots + n$$

$$= \sum_{k=1}^n k$$

$$= \frac{n(n+1)}{2}$$





## 5. 퀵 정렬 복잡도 분석

- 최선의 경우

pivot								
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	9	6	7	3	2	8	1	4

- 데이터가 분할할 수 있게 불규칙적으로 나열되어 있고 피벗이 중간에 해당하는 값에 가깝게 선택이 된 경우  
전체가  $n$ 이면, 절반인  $n/2$ 로 나누어 분할 수행한다

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

## 5. 퀵 정렬 복잡도 분석

### ■ 최선의 경우

pivot								
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	9	6	7	3	2	8	1	4

### ■ 학교에서 배운 수학에서 지수/로그를 참고하면

$$2T\left(\frac{n}{2}\right) + n$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

$$= 4T\left(\frac{n}{4}\right) + 2n$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$= \dots$$

$$= 2^x T\left(\frac{n}{2^x}\right) + xn$$

$$x = \log_2 n \Leftrightarrow 2^x = n$$

$$2^x T\left(\frac{n}{2^x}\right) + xn$$

$$= nT(1) + n\log_2 n$$

$$= kn + n\log_2 n$$

여기서 k는 상수를 의미

# 5. 퀵 정렬 복잡도 분석

## ■ 최선의 경우

pivot								
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	9	6	7	3	2	8	1	4

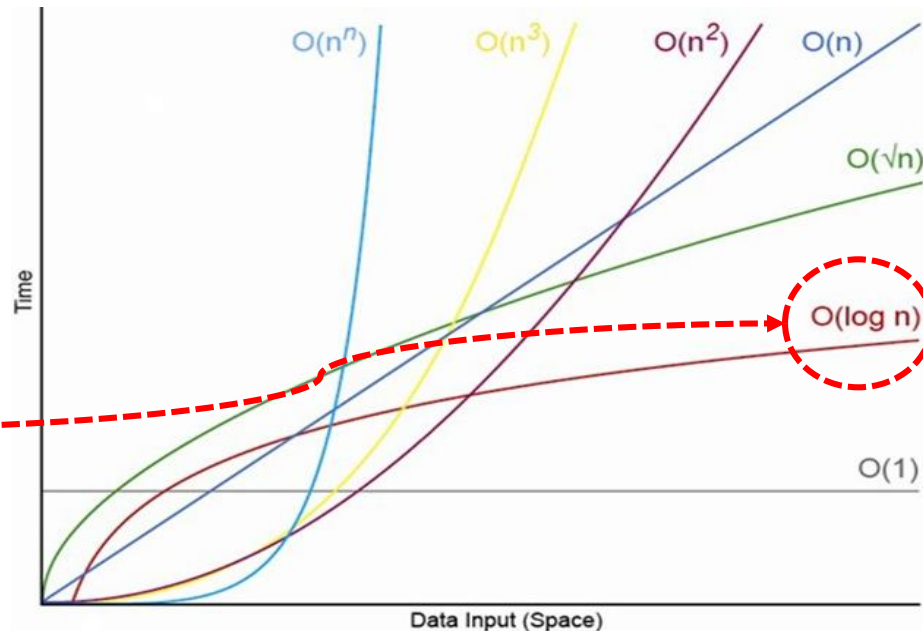
- 무한 급수를 참조해서  $\log_2$ 의  $n$  승 분의 상수  $k$ 는  $= 0$  인 경우 참고하면

$$x = \log_2 n \Leftrightarrow 2^x = n$$

$$\lim_{n \rightarrow \infty} \frac{k}{\log_2 n} = 0$$

- 여기서  $k$ 는 상수를 의미  
아래와 같이 비례

$$\log_2 n = \frac{\log n}{\log 2} = k' \log n$$



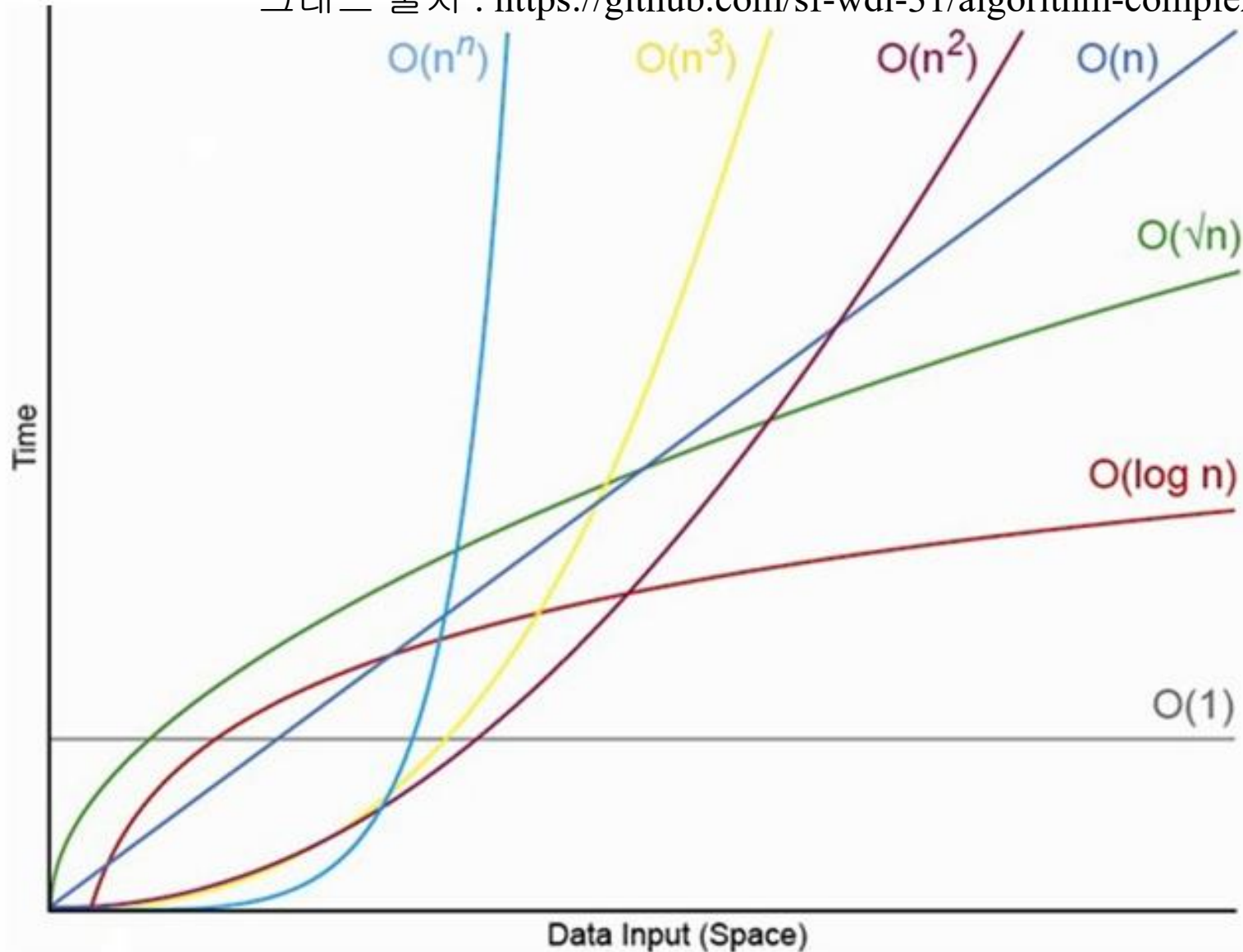
## 5. 정렬 알고리즘의 비교

---

알고리즘	최선	평균	최악
삽입 정렬	$O(n)$	$O(n^2)$	$O(n^2)$
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
버블 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
셸 정렬	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
퀵 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
힙 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
합병 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
기수 정렬	$O(dn)$	$O(dn)$	$O(dn)$

# 5. 정렬 알고리즘의 비교 (Graph)

그래프 출처 : <https://github.com/sf-wdi-31/algorithm-complexity-and-big-o>



## 5. 정렬 알고리즘의 실험 예 (정수 60,000개)

알고리즘	실행 시간(단위:sec)
삽입 정렬	7.438
선택 정렬	10.842
버블 정렬	22.894
셸 정렬	0.056
히프 정렬	0.034
합병 정렬	0.026
퀵 정렬	0.014