EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv2 과정
[자료구조 프로그래밍]

제 1기

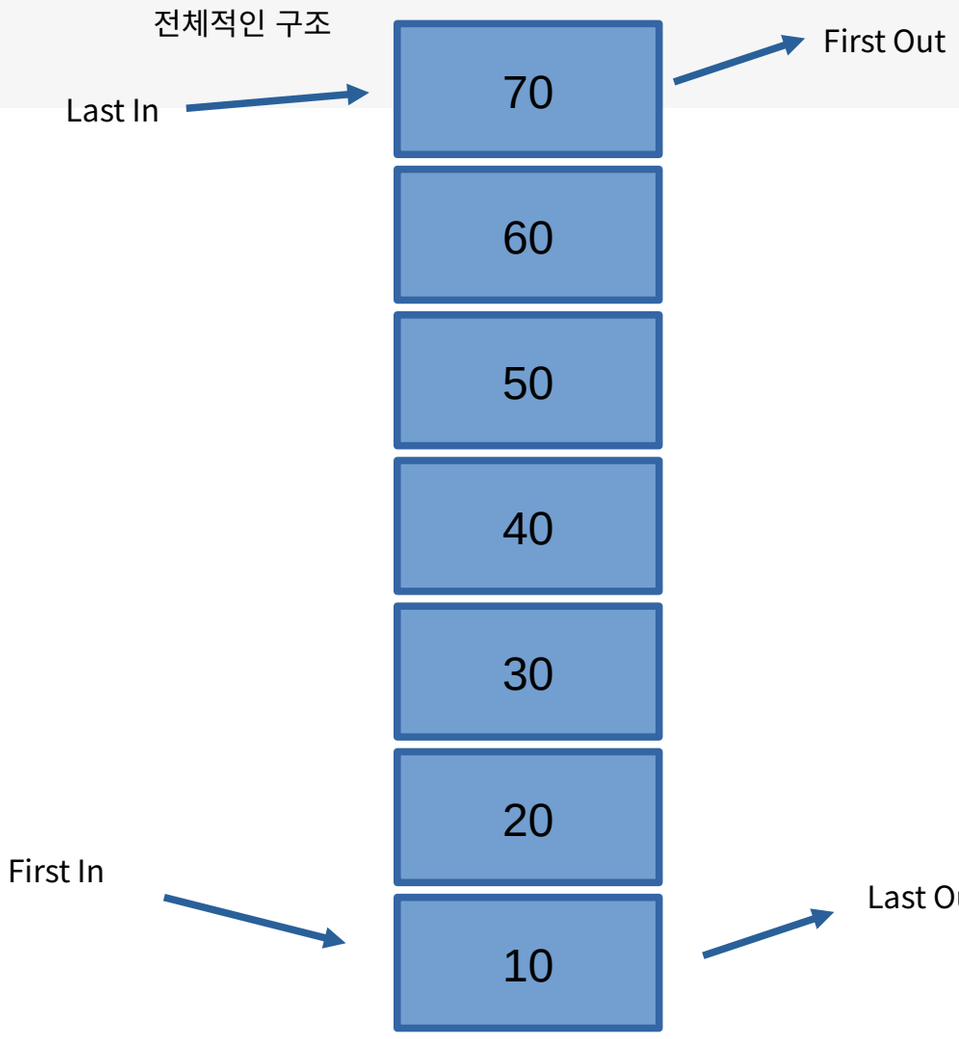
2021. 12. 10

박태인

EDDI
Electronic Design
Development Institute

목차

1) stack 구조 그림을 코드화



1 각각의 노드를
값의 data와
노드를 가르칠 link 두 분으로 나눈 구조체 정의를 하자.

```
Struct _stack
{
        int data;
        struct _stack *link;
}
```

이것을 typedef 을 이용해서 명칭을 정의 한다.

```
Typedef struct _stack stack;
Struct _stack
{
        int data;
        struct _stack *link;
}
```

```
typedef struct _stack stack;
struct _stack
{
        int data;
        struct _stack *link;
};
```

1) stack 구조 그림을 코드화

② 우선 main을 구성 해본다.

Stack 쌓을 값을 나열하고
for문을 통해
stack을 입력 및 출력 한다.

```
Int main(void)
{
        stack *top = NULL;
        int data[] ={10, 20, 30, 40, 50, 60, 70}
        int I;

        for(i=0; i<7; i++)
        {
        stack 값 in
        }

        for(t=0; i<7; i++)
        {
                stack 값 out
        }

        return 0;
}
```

push_data(&top, data[i])

```
Void push_data(stack **top, int data)
{
        stack *tmp = *top;

        *top = create_stack_node();
        (*top)→data = data;
        (*top)→link = tmp;
}
```

Push 함수로 오면서 top의 주소 값과 data 값을 가져온다.

그리고 main의 top의 값이 새로 생성 될 node를 가르키게 하기 위해 *top = 생성 node

이제 부터 top 은 heap 영역에 새로 생성될 node를 가르키고 있으므로
(*top)→data 에 data 값을 넣어 준다.

link에는 어떤 값을 넣을 까? link는 앞의 node를 가르켜야 할 것이다. Why?
여기서 tmp가 필요한 이유가 나오는 것.

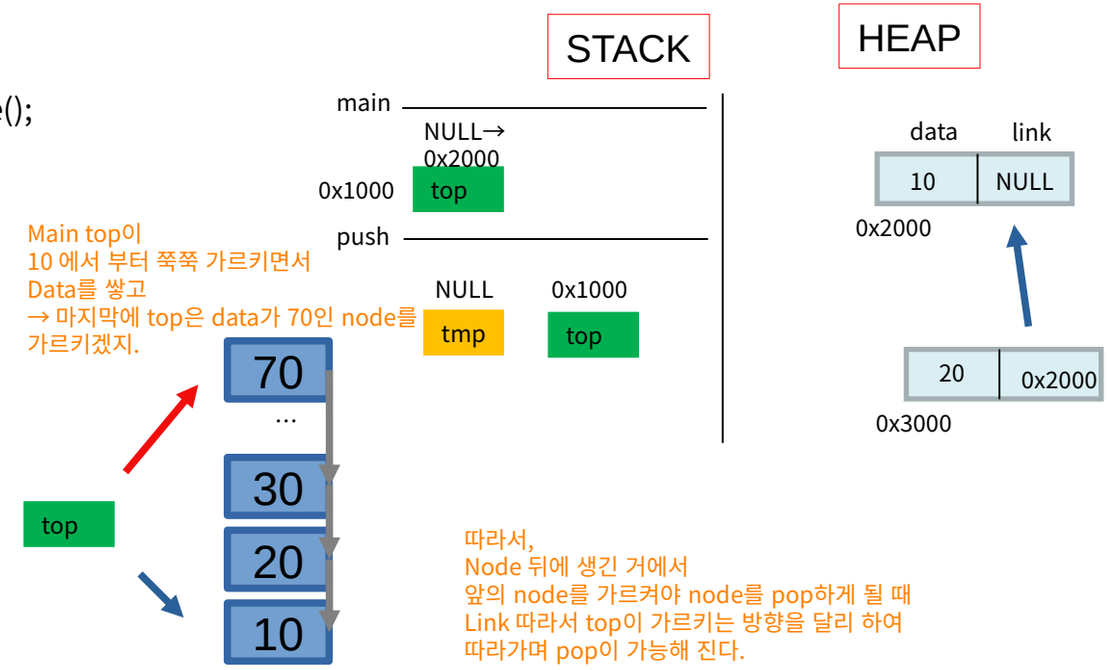
top은 create node가 됨에 따라 값이 달라지게 될 것이고,
link에 create node 전 top의 주소를 넣을 수 있도록 tmp에 주소 값을 저장 시켜 놓았다.

③ push에 대한 개념을 잡아보자

조건 - push를 할려면 push를 할 node가 필요하다.
- 첫 시작 node를 top 이라 지칭 하고 시작 값을 NULL 로 한다.
- 가르키는 node에 data를 순서대로 집어 넣는다.

push_data(&top, data[i])

Push 함수를 만들어 보면,
main에서 가르키는 첫 node top의 주소를 가져가고 data를 가져가서
가르키는 node에 data를 넣도록 해보자.



1) stack 구조 그림을 코드화

```
Void push_data(stack **top, int data)
{
        stack *tmp = *top;

        *top = create_stack_node();
        (*top)→data = data;
        (*top)→link = tmp;
}
```

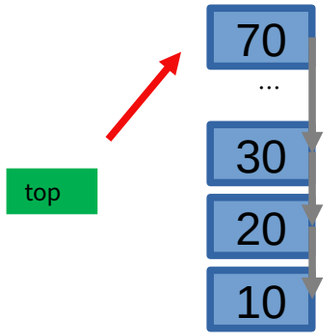
4 앞서서 push 함수를 만들었는데, data를 push 하려면 자연스럽게 node를 생성하는 함수가 필요 할 것이다.

```
create_stack_node();

Stack *create_stack_node(void)
{
        stack *tmp;

        tmp = (stack *)malloc(sizeof(stack));
        tmp→link = NULL;

        return tmp;
}
```

Heap 영역에 node를 생성하기 위해 malloc을 사용 할 것 이고, 그 값을 받기 위한 tmp를 생성.

새로운 node이기 때문에 우선 tmp→link는 NULL 처리.

5 이번에는 pop을 해봐야 겠죠?

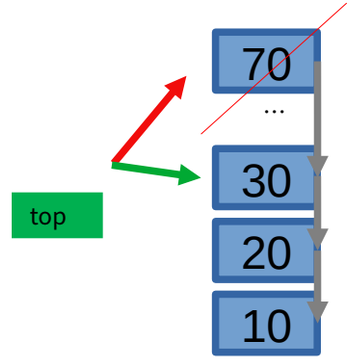


pop의 알고리즘을 생각해 보면
모든 data의 push가 완료 되면 왼쪽과 같은 그림이 되고.

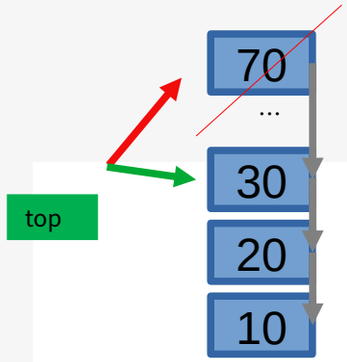
오른쪽 그림과 같이 가르키고 있는 node를 free 시켜 준뒤

link가 가르키는 다음 node로 이동하고,
차례로 가르키는 node를 free 시키도록 한다.

create_stack_node 함수가 stack * 형인 이유?
→ 반환되어 들어가는 top이 *top 이므로



1) stack 구조 그림을 코드화



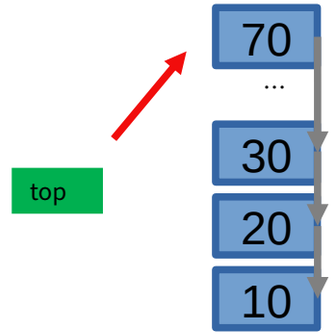

```
Void pop_data(stack **top)
{
        stack *tmp;

        if(!(*top))                     // 우선 pop이 할게 없는지 확인
        {
                printf("stack is empty\n");
                return -1;
        }

        tmp = *top;             // top의 값을 tmp에 백업

        *top = tmp→link;   // tmp의 link는 이전 node 값의 주소가 들어 있으므로 *top을 하면 top이 이전의 node를 가르키게 된다.

        free(tmp);              // free 시킨다.

        return 0;
}
```

1) stack 구조 그림을 코드화

- 이제 main이 좌측과 같은 형태를 가지게 되고, 이번에는 node의 data를 print 하는 함수를 만듦으로써 함수가 제대로 동작하는지 살펴 볼 것이다.

EDDI
Electronic Design Development Institute

```
Int main(void)
{
        stack *top = NULL;
        int data[] ={10, 20, 30, 40, 50, 60, 70};
        int I;

        for(i=0; i<7; i++)
        {
                push_data(&top, data[i]);
        }

        print_stack_data(top);

        for(t=0; i<8; i++)
        {
                pop_data(&top);
        }

        print_stack_data(top);

        return 0;
}
```




```
Void print_stack_data(stack *top)
{
        while(top)
        {
                printf("data = %d\n, top→data);  // top의 data 출력
                top = top→link;      //  top을 다음 link 가르키게
        }
        printf("\n");
}
```

```
data = 70
data = 60
data = 50
data = 40
data = 30
data = 20
data = 10

Stack is empty
```

Print 함수는 문제가 없는 듯 하나
Push 하듯이 print 하면 이미

모두 free가 된 상태기 때문에
Stack is empty가 인쇄 되어 버린다.

삭제 되는게 어떤건지 그리고 결과가 어떻게 됐는지 알고 싶으니 고쳐 보자.

```
for(t=0; i<8; i++)
        {
                printf("pop_data = %d\n", pop_data(&top);
                print_stack_data(top);
        }
```

이런식으로 하면 pop 되는게 무엇인지 알 수 있게 되고,
for문을 돌 때 마다 어떤식으로 삭제 됐는지 알 수 있게 된다.

아니 근데, 이렇게 될려면 pop_data가 int형 data를 반환 해야 한다.

1) stack 구조 그림을 코드화

```
for(t=0; i<8; i++)
        {
                printf("pop_data = %d\n", pop_data(&top);
                print_stack_data(top);
        }
```

이런식으로 하면 pop 되는게 무엇인지 알 수 있게 되고,
for문을 돌 때 마다 어떤식으로 삭제 됐는지 알 수 있게 된다.

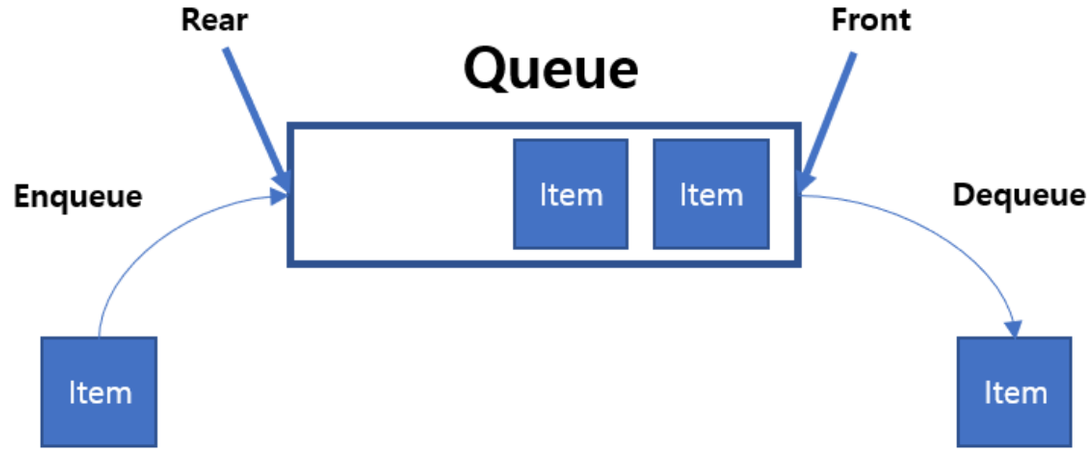
아니 근데, 이렇게 될려면 pop_data가 int형 data를 반환 해야 한다.

```
int pop_data(stack **top)
{
        int data;
        stack *tmp;

        if(!(*top))                     // 우선 pop이 할게 없는지 확인
        {
                printf("stack is empty\n");
                return -1;
        }

        tmp = *top;             // top의 값을 tmp에 백업

        data = tmp→data;  // 반환 할 data 값은 tmp의 data 값 이다.
        *top = tmp→link;   // tmp의 link는 이전 node 값의 주소가 들어 있으므로 *top을 하면 top이 이전의 node를 가르키게 된다.

        free(tmp);              // free 시킨다.

        return data;
}
```

```
data = 70
data = 60
data = 50
data = 40
data = 30
data = 20
data = 10

pop = 70
data = 60
data = 50
data = 40
data = 30
data = 20
data = 10

pop = 60
data = 50
data = 40
data = 30
data = 20
data = 10

pop = 50
data = 40
data = 30
data = 20
data = 10

pop = 40
data = 30
data = 20
data = 10

pop = 30
data = 20
data = 10

pop = 20
data = 10

pop = 10

Stack is empty
pop = -1
```

1) Queue 구조 그림을 코드화

◆ 큐는 한쪽 끝(rear)에서는 삽입 연산만 이루어지며, 다른 한 쪽 끝(front)에서는 삭제 연산만 이루어지는 유한 순서 리스트 이다.



◆ 특성 : 구조상 먼저 삽입된 item이 먼저 삭제가 이루어 진다. (FIFO)

일단 어쨌든 node형 struct를 만들어 봅시다

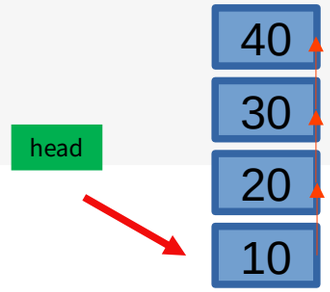
Typedef struct _queue queue;
Struct _queue
{
- int data;
- struct _queue *link;
- };

```
typedef struct _queue queue;
struct _queue
{
        int data;
        struct _queue *link;
};
```

1) Queue 구조 그림을 코드화

그리고 전략을 한번 짜보자.

일단 저 node를 가지고 main을 적어 보는거지




```
int main(void)
{
        int I;
        int data[ ] = { 10, 20, 30, 40};
        queue *head  = NULL;

        for(i=0; i<4; i++)
        {
                enqueue_data(&head, data[i]);  // queue를 삽입
        }

        for(i=0; i<5; i++)
        {
                dequeue_data(&head);  // queue를 빼기
        }

        return 0;
}
```

일단 이정도 방향성을 잡아 둔다.
우선 enqueue를 생각해 보자.

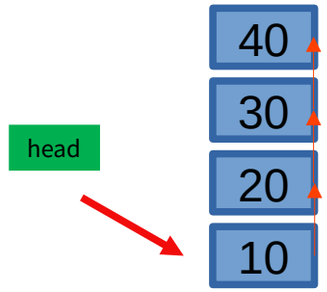
일단, 생각으로는 앞선 stack 처럼 node를 하나씩 쌓지만,
main의 head는 계속 첫 생성했 node를 가르키게 하고
link를 다음 node를 가르키게 해서

삭제 시에 head가 가르키는 방향을 조절하면서
처음 IN 했던 값이 처음 OUT 될 수 있는 구조로 잡아 보자.

```
enqueue_data(queue **head, int data)
{
        if(!(*head))          // head가 가르키는 node가 없는지? 없다면
        {
                *head = create_queue();  //  생성하고
                                         //  이게 나중에 재귀호출 하고 재생성 할 시에
                                         //  head가 link를 가르키고 있기에
                                         //  거기다가 값을 넣으므로
                                         //  생성하면서 다음 node를 가르키게 된다.
                (*head)→data = data;     //  생성한 node에 data 삽입
                return;
        }
        enqueue_data(&*(head)→link, data);  // 재귀 방식 구현
                                            // 재귀를 head→link 주소를 가져와서
                                            // 생성 하면서 그곳에 다음 생성 되는 node를
                                            // 가르키도록 한다.
                                            // 그러면 자연스럽게 위와 같은 구조가 형성됨.
}

// head가 가르키는 방향을 달리 하진 않았다.
// 삭제시에 head 가르키는 방향을 link 따라가면서 삭제하도록 하기 위함.
```

1) Queue 구조 그림을 코드화

```
int main(void)
{
        int I;
        int data[ ] = { 10, 20, 30, 40};
        queue *head  = NULL;

        for(i=0; i<4; i++)
        {
                enqueue_data(&head, data[i]);  // queue를 삽입
        }

        for(i=0; i<5; i++)
        {
                dequeue_data(&head);  // queue를 빼기
        }

        return 0;
}
```




```
enqueue_data(queue **head, int data)
{
        if(!(*head))            // head가 가르키는 값이 없으면
        {
                *head = create_queue();  // node 생성
                *head→ data = data;      // 생성한 node에 data 삽입
                return;
        }

        enqueue_data(&(*head)→link, data);
}
```

그럼 이번에는 create_queue를 만들어 봐야 겠죠..

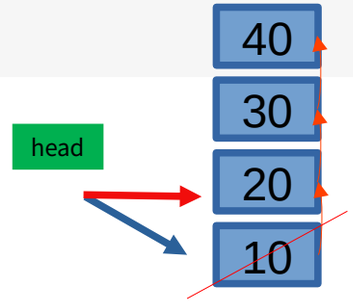
```
Queue *create_queue()
{
        queue *tmp;

        tmp = (queue *)malloc(sizeof(queue));
        tmp→link = 0;

        return tmp;
}
```

Create node를 할 때,
tmp를 만들어서 malloc의 값을 받고
Return 한다.

1) Queue 구조 그림을 코드화

```
int main(void)
{
        int I;
        int data[ ] = { 10, 20, 30, 40};
        queue *head  = NULL;

        for(i=0; i<4; i++)
        {
                enqueue_data(&head, data[i]);  // queue를 삽입
        }

        for(i=0; i<5; i++)
        {
                dequeue_data(&head);  // queue를 빼기
        }

        return 0;
}
```

이번에는 dequeue를 구현해 보고자 한다.



일단 그림적으로 구현해 보면
아래에서 부터 순서대로 삭제를 시켜야 한다고 가정 했을 때
Head 가 가르키는 node를 가르키고

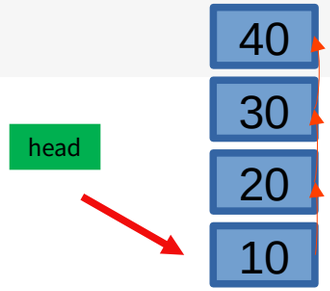
head는 다음 링크가 가르키는 곳을 가르키면 될 것이다.

```
Void dequeue_data(queue **head)
{
        if(*head)
        {
                queue *tmp = *head;       // head  가 가르키는 방향을 아래 코딩같이 표현해서 나타낸 다음
                                          // 그 다음 free 할 수 있도록 하기 위함.
                *head = (*head)→link;             // *head 가 가르키는 방향을 다음 link로 하기 위함
                free(tmp);
        }
        else
        {
                printf("Queue is empty!\n");
        }
}
```

1) Queue 구조 그림을 코드화

```
int main(void)
{
        int I;
        int data[ ] = { 10, 20, 30, 40};
        queue *head  = NULL;

        for(i=0; i<4; i++)
        {
                enqueue_data(&head, data[i]);  // queue를 삽입
        }
        print_queue(head);          // queue를 다 만들고 나서 쫙 출력

        for(i=0; i<5; i++)
        {
                dequeue_data(&head);  // queue를 빼기
        }

        return 0;
}
```

이번에는 앞 선 코드가 다 구현이 완료 되었지만
여기서 print를 함으로써 출력을 한번 해봅시다.



우선 위와 같은 모습에서 print를 한다고 생각하면

```
Print_queue(queue *head)  // queue 구조체를 가리키는 head를 그대로 받은 뒤
{
        while(head)  // head가 존재 하는가? 존재 한다면
        {
                printf("queue head = %d\n", head→data);
                head = head→link;          // head는 다음 head의 link를 가르키게 한다.
        }
}
```

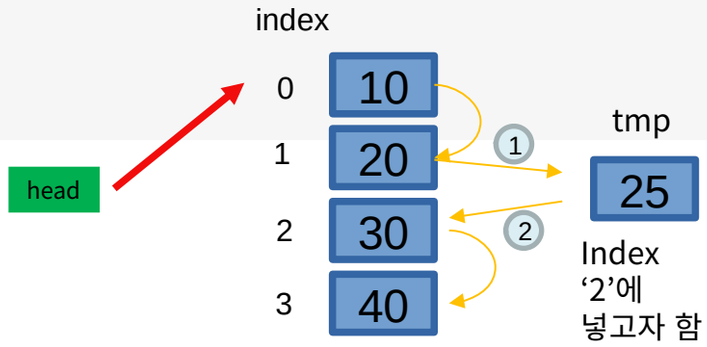
```
queue head = 10
queue head = 20
queue head = 30
queue head = 40
```

```
void dequeue_data(queue **head)
{
        if(*head)
        {
                queue *tmp = *head;
                printf("dequeue_data = %d\n",(*head)->data);
                *head = (*head)->link;
                free(tmp);
        }
        else
        {
                printf("Queue is empty!\n");
        }
}
```

그런데 dequeue는 어떻게 출력해 볼까?
삭제 하고 나서 출력은 무의미 하므로 삭제 하기 전에 어떤걸 삭제 하게 되는지 출력해 본다.

```
queue head = 10
queue head = 20
queue head = 30
queue head = 40
dequeue_data = 10
dequeue_data = 20
dequeue_data = 30
dequeue_data = 40
Queue is empty!
```

1) Queue 구조 그림을 코드화



한가지 더 생각 할 부분이 있다.
queue에서 이렇게 차례대로 data를 삽입 할 수도 있지만 data들
가운데 중간에 값을 내가 넣고자 하는 위치에 넣고 싶을 수 있을 것이다.

넣고자 하는 위치 → 이것을 index 라는 개념을 활용해 보도록 한다.

enqueue_data(&head, data, index)

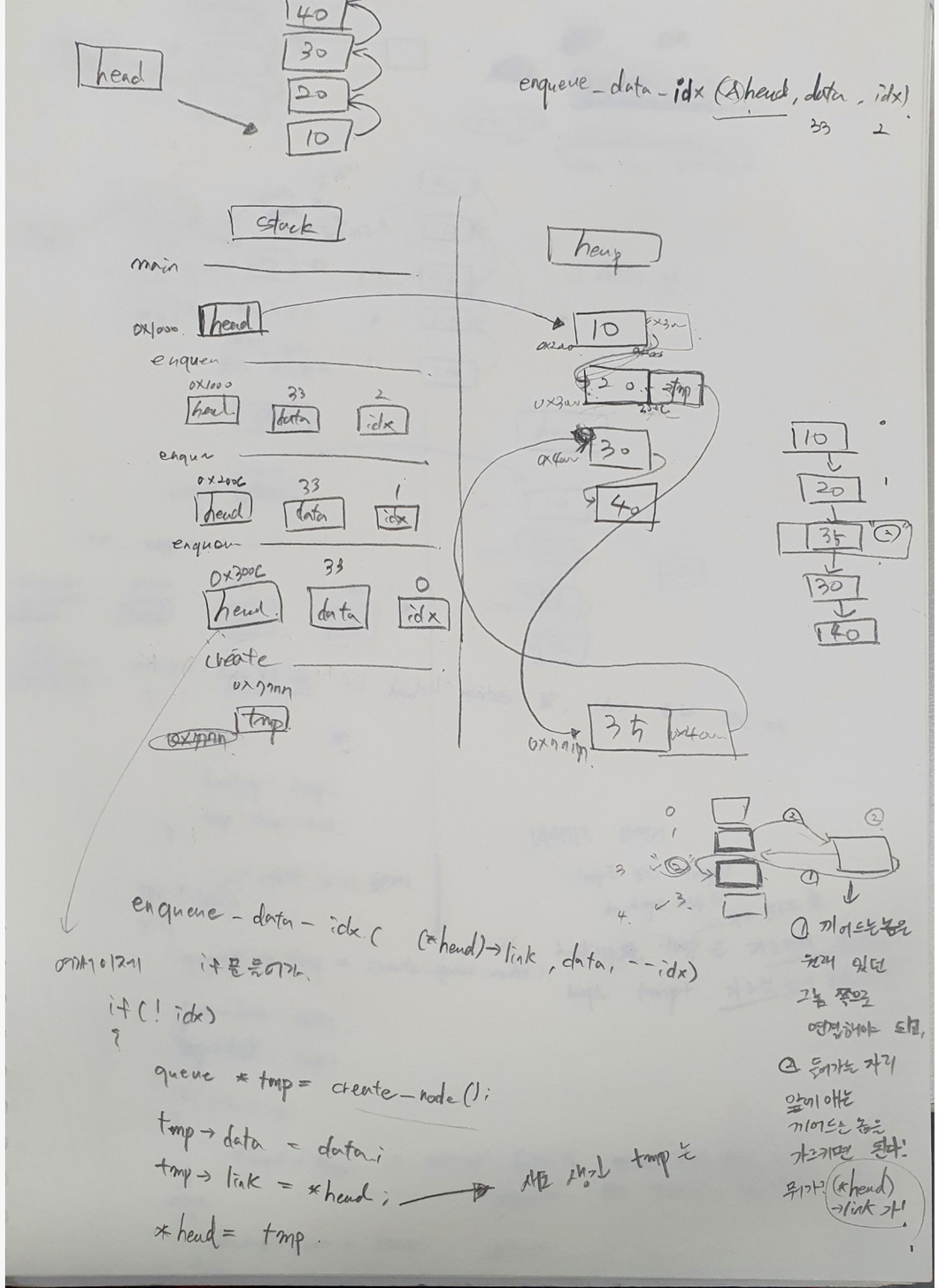
Index를 활용하여 node들 마다 번호를 매기는 것이다.

대충 모양을 생각해 보면

```
Void enqueue_data(queue **head, int data, int idx)
{
        if(!idx)        // index 가 존재하지 않을 경우 새롭게 생성
        {
                queue *tmp = create_queue_node();

                tmp→data = data;
                tmp→link=*head;

                *head = tmp;
                return;
        }
        enqueue_data(&(*head)→link, data, idx);
}
```

① 전략 1) 추가하려는 index 넘버의 앞 node는 tmp(추가하려는 node)를 가르켜야 한다.

② 전략 2) tmp는 추가 하려고 했던 index 자리의 node를 가르켜야 한다.



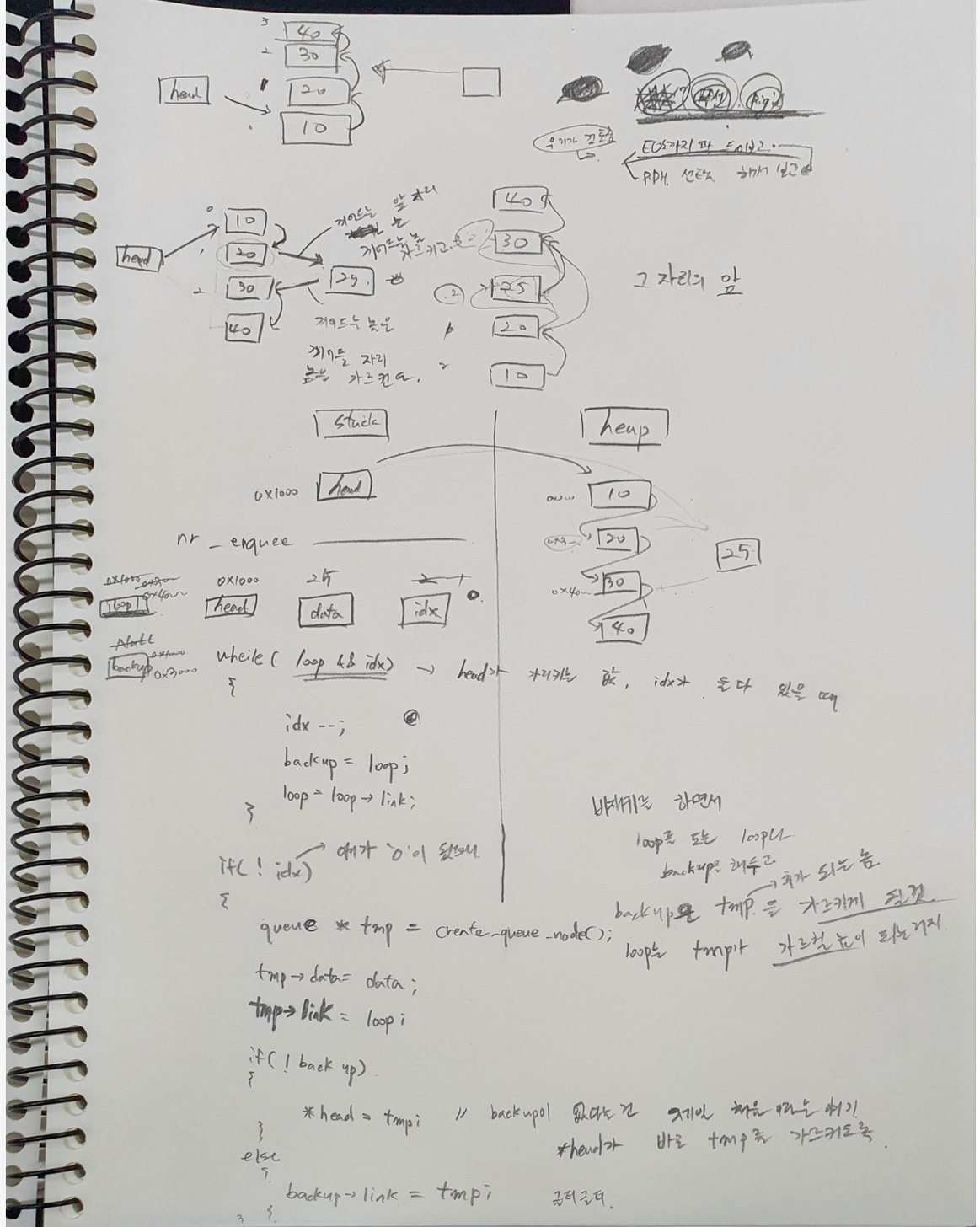
전략 1) 추가하려는 index 넘버의 앞 node는 tmp(추가하려는 node)를 가르켜야 한다.

전략 2) tmp는 추가 하려고 했던 index 자리의 node를 가르켜야 한다.

```c
void enqueue_data_idx(queue **head, int data, int idx)
{
        // 중간에 넣는 경우
        if (!idx)
        {
                queue *tmp = create_queue_node();

                tmp->data = data;
                tmp->link = *head;

                *head = tmp;
                return;
        }

        // 예외 핸들링
        if (!(*head))
        {
                printf("작업이 불가합니다!\n");
                return;
        }

        enqueue_data_idx(&(*head)->link, data, --idx);
}
```

이것을 다시 코딩화 하면

```c
Void enqueue_data_idx(queue **head, int data, int idx)
{
        if(!idx) // inx가 0이 되는 차례가 되면
        {
                queue *tmp = create_queue_node(); // 삽입 할 node 생성

                tmp->data = data; // 넣을 데이터 삽입
                tmp->link = *head; // tmp의 link가 들어 갈 idx의 node를 가르키도록

                *head = tmp; // *head 즉 들어갈 자리 앞 idx의 node가 tmp를
                             가르키도록
                return;
        }

        if(!(*head))    // *head가 가르키는 것이 없을 때
        {
                pritnf(“작업 불가\n”);
                return;
        }

        enqueue_data_idx(&(*head)->link, data, --idx);  // idx 값을 빼가면서 자리를
}                                                       찾아가는 과정
```



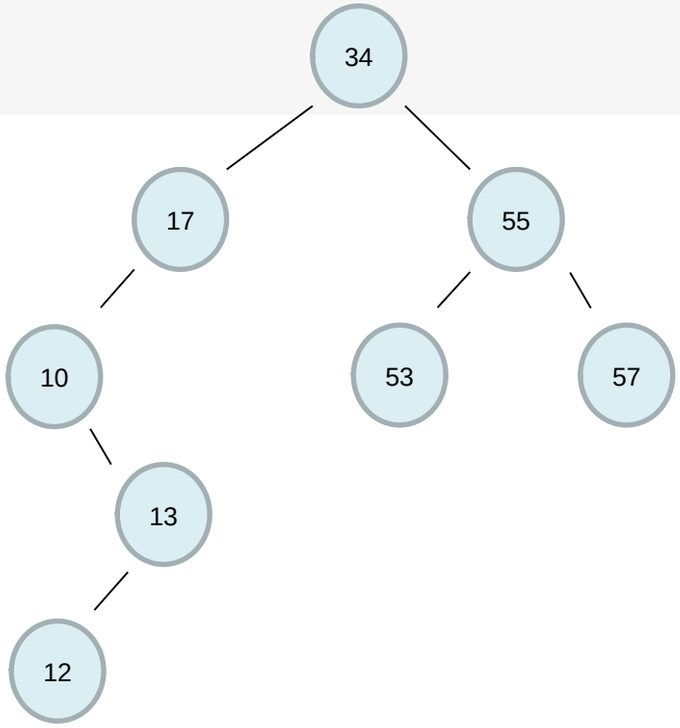
※ 이것을 비재귀로 구현 한다.

- 비재귀로 하려면 일단 쉽게 생각해서 재귀로 하던 행동을 비재귀로 구현한다고 생각해 보자.
  - L idx 자리 찾아가는 과정을 반복문을 통해 찾아 간다.
    (들어가야 할 node(tmp)를 가르켜야 할 node[loop],
    들어가야 할 node(tmp)가 가르켜야 할 node[backup] 를 만들어서 활용해 볼 것)
  - L 제일 처음 위치에 들어 갔을 때는 head 가 tmp를 가르켜야 한다는 점
  - L 이외의 상황은 작업 불가 처리


```
void nr_enqueue_data_idx(queue **head, int data, int idx)
{
        queue *loop = *head;
        queue *backup = NULL;

        // 숏컷으로 예외처리 대신함
        while (loop && idx)
        {
                idx--;
                backup = loop;
                loop = loop->link;
        }

        if (!idx)
        {
                queue *tmp = create_queue_node();

                tmp->data = data;
                tmp->link = loop;

                if (!backup)
                {
                        *head = tmp;
                }
                else
                {
                        backup->link = tmp;
                }
                return;
        }

        printf("처리가 불가능한 작업입니다!!!\n");
}
```

```
Nr_enqueue_idx(queue **head, int data, int idx)
{
    queue *loop = *head;   // tmp를 가르키게 될
    queue *backup = NULL; // tmp가 가르키게 될

   while(loop && idx)   // 자리 찾아 가기
   {
       idx--; // idx 하나씩 줄이 면서
       backup = loop;  // loop 값을 backup에 넣어둬
       loop = loop->link; // loop에 loop의 link를 넣어, 여기에 tmp가 가르키도록 해야지 나중에
   }
   if(!idx)   // 자리를 찾으면
   {
       queue *tmp = create_queue_node();     // 새로운 node 생성 후 tmp에 담고
       tmp->data = data;     // 넣어야 할 data를 넣고
       tmp->link = loop;    // loop는 tmp가 가르켜야 할 node 이므로 tmp->link에 삽입

       if(!backup)   // 제일 앞에 삽입 하는 경우의 예외 처리
       {
           *head = tmp;    // *head가 tmp를 가르켜 버리면 된다.
       }
       else
       {
            backup->link = tmp; // tmp를 가르킬 놈은 backup->link 이므로
        }
    }
```

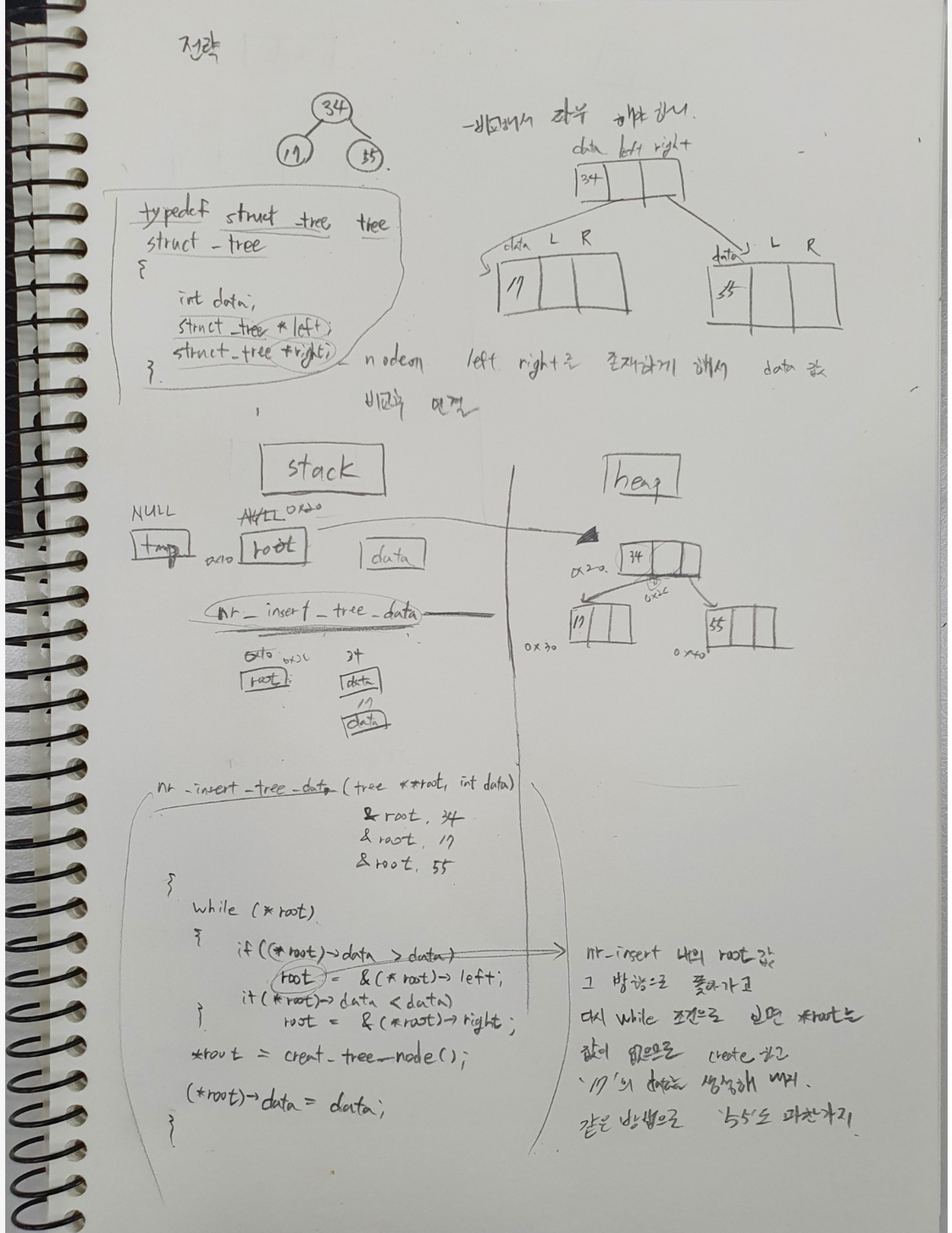
2) 이진 트리 구조 그림을 코드화



- 이진 트리 {34, 17, 55, 10, 13, 12, 53, 57 }; 로 입력 했을 때 좌측과 같은 구조
- 앞 서 만들어진 data 에 다음 data의 값 크기에 따라 작은 값일 때 왼쪽, 큰 값을 때 오른쪽

```
data = 34, left = 17, right = 55
data = 17, left = 10, right = NULL
data = 10, left = NULL, right = 13
data = 13, left = 12, right = NULL
data = 12, left = NULL, right = NULL
data = 55, left = 53, right = 57
data = 53, left = NULL, right = NULL
data = 57, left = NULL, right = NULL
```

```
Int main(void)
{
        int I;
        tree *root = NULL;
        int data[ ] = { 34, 17, 55, 10, 13, 12, 53, 57 }

        for(i=0; i<8; i++)
        {
              nr_insert_tree_node(&root, data[ I ]
        }
}
```



Node 를 만들고
Enqueue 하는 방법을 연구해 보자.

1) 노드 만들기
L 이진 트리로 연결 할 수 있도록 왼쪽 오른쪽을 만들자.

```
typedef struct _tree tree;
struct _tree
{
        int data;
        struct _tree *left;
        struct _tree *right;
};
```

```
Typedef struct _tree tree
Struct _tree
{
    int data;
    struct _tree *left;
    struct_tree *right;
}
```

2) nr_insert_tree 비재귀 삽입 함수 만들어 보자.

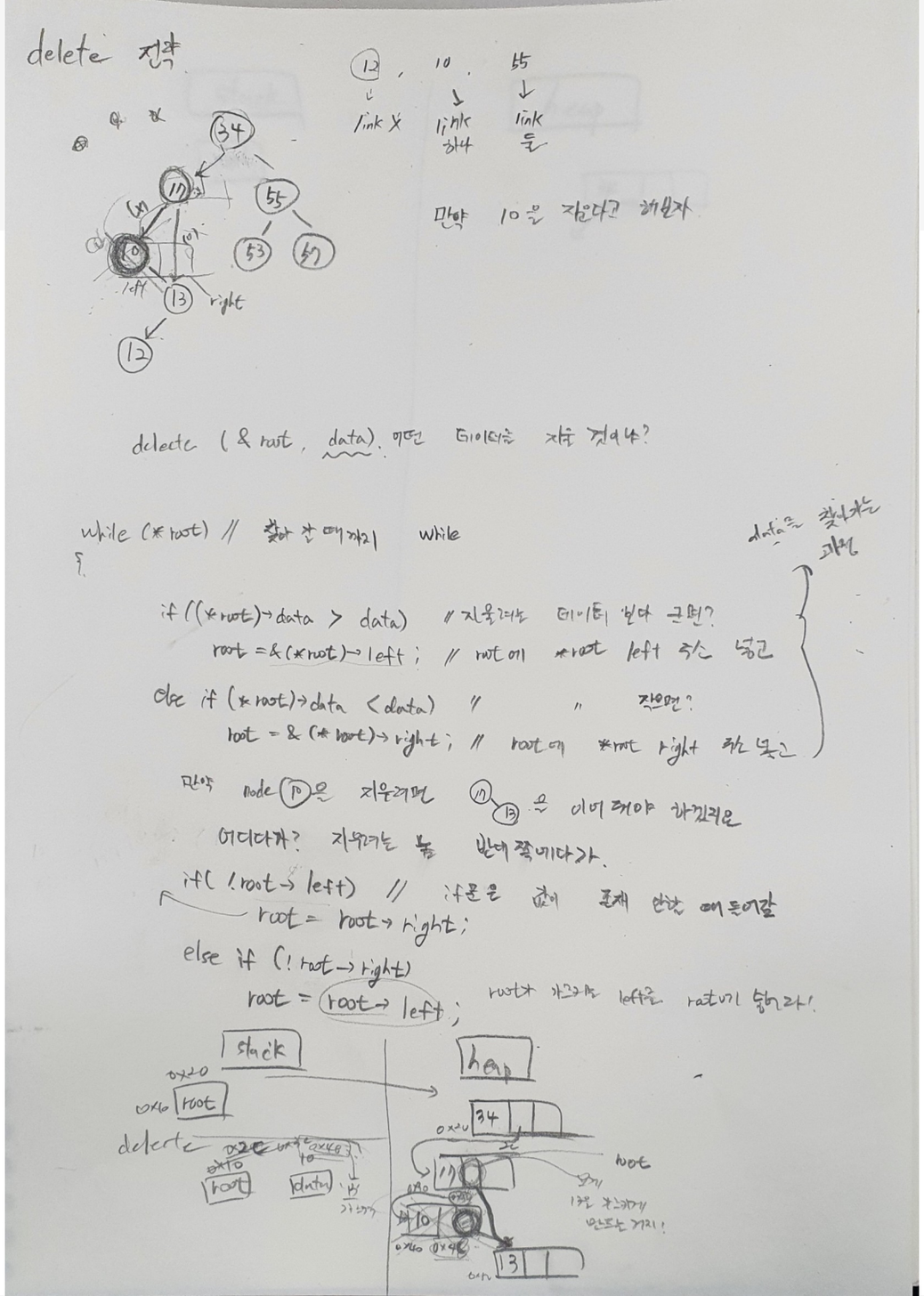
```
void nr_insert_tree_data(tree **root, int data)
{
        while(*root)
        {
                if((*root)->data > data)
                        root = &(*root)->left;
                else if((*root)->data < data)
                        root = &(*root)->right;
        }

        *root = create_tree_node();
        (*root)->data = data;
}
```

실행 : nr_insert_tree_data(&root, data)

```
Nr_Insert_tree_data(tree **root, int data)
{
    while(*root)   // root가 가르키는게 있다면
    {
        if(*root)->data > data)        // 들어가는 값과 비교해서 작으면 왼쪽
                root = &(*root)->left;
        else if((*root)->data < data)   // 들어가는 값과 비교해서 크면 오른쪽
                root = &(*root)->right;
    }
    (*root) = creat_tree_node();  // root가 가르키는 값이 없다면 생성
    (*root)->data = data;   // 생성하고 data 삽입
}
```

```
Tree *Create_tree_node(void)
{
    tree *tmp;
    tmp = (tree *)malloc(sizeof(tree));
    tmp->left = 0;
    tmp->right = 0;

    return tmp;
}
```

```
tree *create_tree_node(void)
{
        tree *tmp;

        tmp = (tree *)malloc(sizeof(tree));
        tmp->left = 0;
        tmp->right = 0;

        return tmp;
}
```



Node 를 삭제하는 방법에 대한
Dequeue를 연구해 보자.

Delete_tree(&root, data);
L 이와 같은 형태로 구현 한다고 했을 때,

EDDI
Electronic Design Development Institute

```
Delete_tree(tree **root, int data);
{
  if((*root)->data > data)    // 지울려는 데이터 보다 크면?
      root = &(*root)->left;   // root에 *root -> left 주소 넣고
  else if(*root)->data < data)   // 지울려는 데이터보다 작으면?
      root = &(*root)->right; // root에 *root right 주소 넣고
```

찾아가는 과정

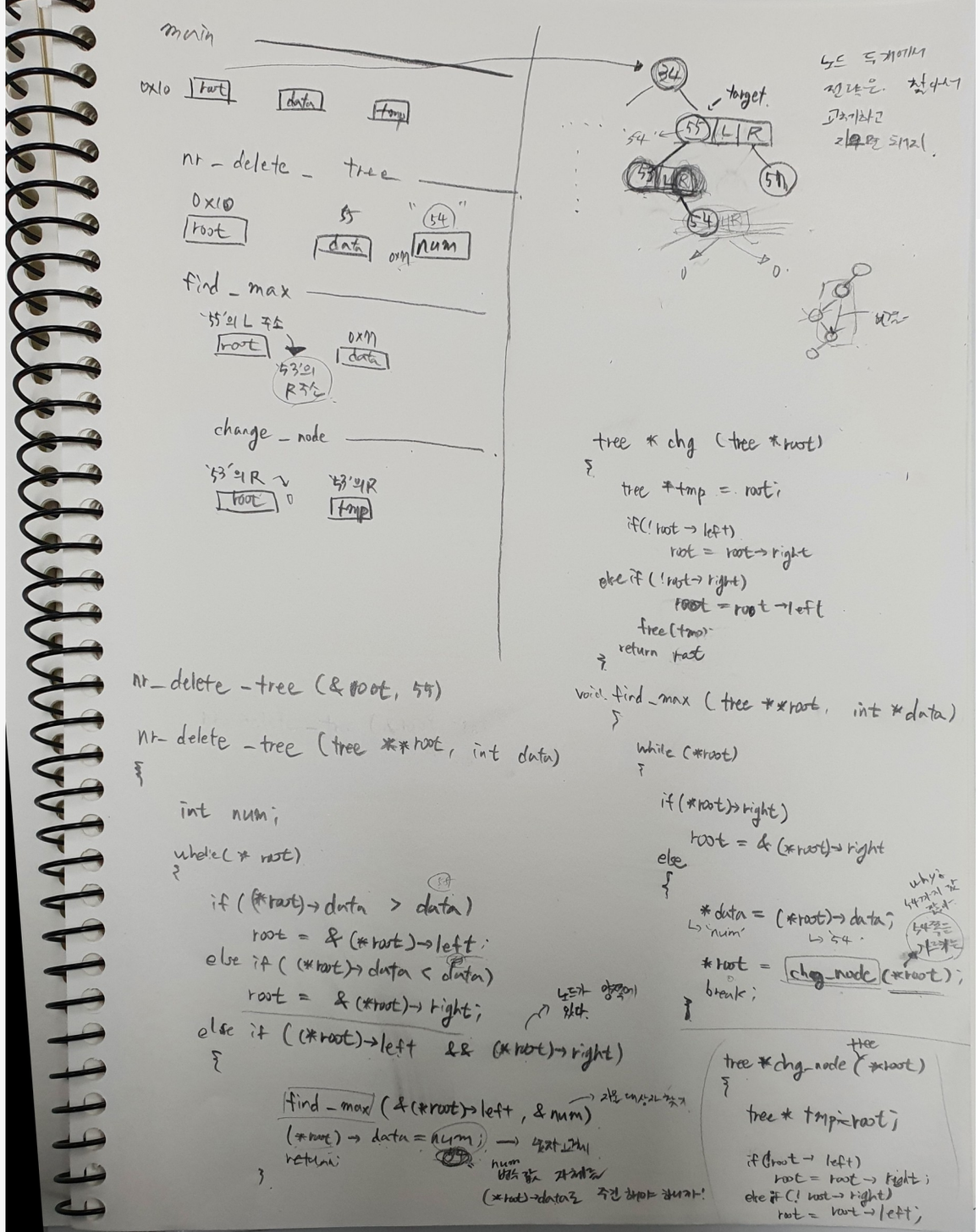
```
  Tree *tmp = root;    // 삭제하게 될 node를 담아 두고

  If(!root->left)  // if문은 값이 존재 안 할 때 들어감
      root = root->right;
  Else if(!root->right)
      root = root->left;

  Free(tmp);

  Return;
}
```

↓

```
(*root) = change_node(*root);
```

여기서 삭제하고 가르키는 node를
달리하는 것을
함수화 해본다면

```
Tree *change_node(tree *root)
{
    tree *tmp = root;

    if(!root->left)
       root = root->right;
    else if(!root->right)
       root = root->left;

    free(tmp);

     return root;
}
```



Tree_delete 구현(2 node_)

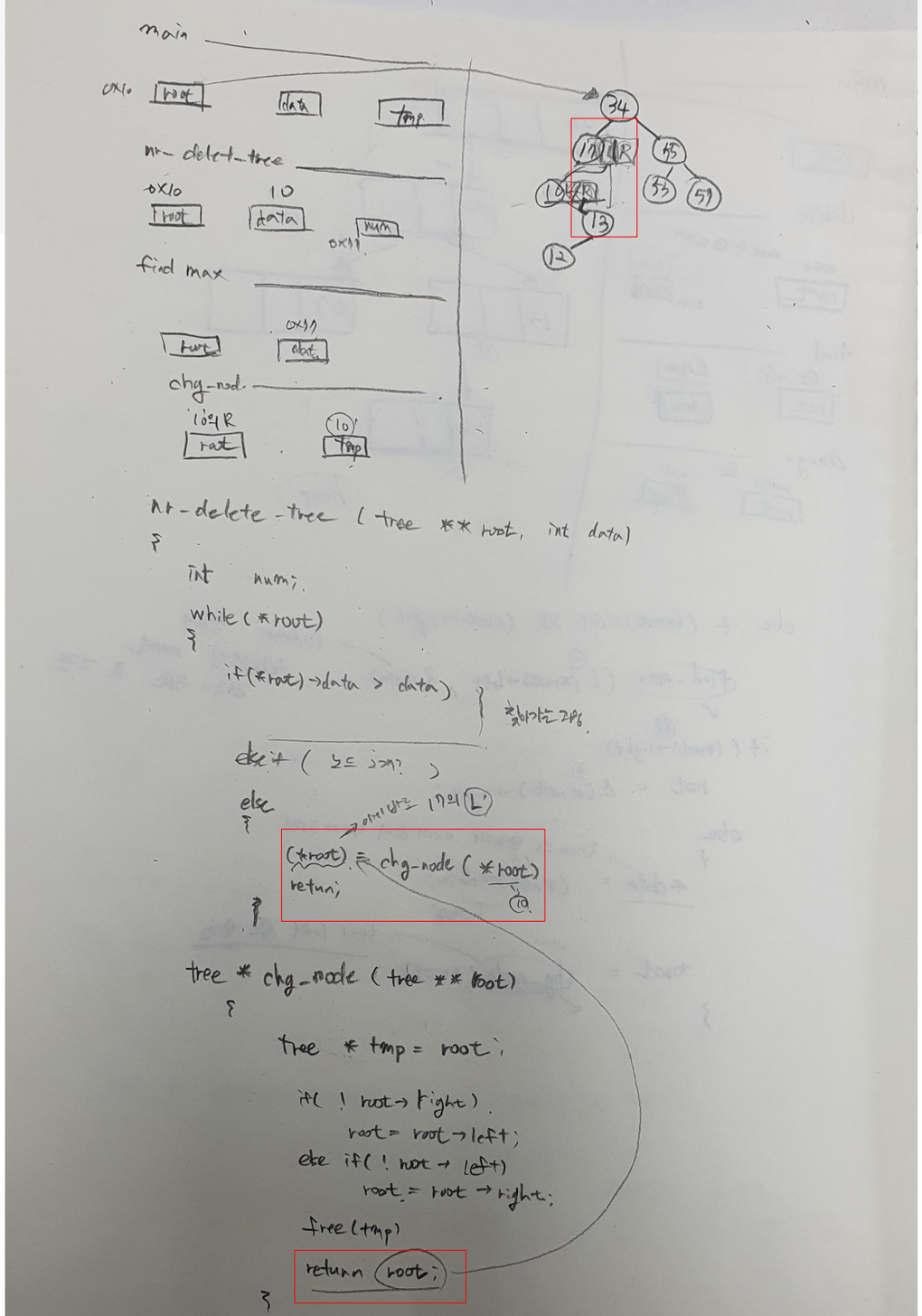
nr_delete_tree(&root, 55)

```
nr_delete_tree(tree **root, int data)
{
   Int num;

   while(*root)
   {
     if((*root)→data > data)
         Root = &(*root)→left;
    Else if((*root)→data <data)
        Root = &(*root)→right;
    Else if((*root)→left && (*root)→right)
    {
           find_max(&(*root)→&num)
          (*root)→data = num;
          return;
    }
```

```
Void find_max(tree **root, int *data)
{
   while(*root)
   {
      if((*root)→right)
         Root = &(*root)→right;
     Else
     {
        *data = (*root)→data;
       root = chg_node(*root);
     }
```

```
Tree *chg_node(*root)
{
  Tree * tmp = root;

 If(!root → left)
    Root = root → right;
 Else if(!root → right)
   Root = root → left;

  free(tmp);

  Return root;
}
```

Tree_delete 구현(1 node_ & connection)



nr_delete_tree 함수에서

지우려는 node를 찾아간 뒤에

change_node 함수로 들어 가서

왼쪽 오른쪽 있고 없고 비교하고,
없는 곳에

없애는 노드를 연결 하게끔

Root 값을 리턴하여 연결 한다.