EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv2 과정
[자료구조 프로그래밍]

제 1기

2021. 11. 05

박태인

EDDI
Electronic Design
Development Institute

목차

1) 비 재귀 insert_data (54 삽입)

```
1 nr_insert_tree_data(&root, 54);
  printf("55 삭제\n");
  //delete_tree_data(&root, 55);
  nr_delete_tree(&root, 55);
  print_tree(root);
```

```
// 재귀호출을 사용하지 않음(nr)

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

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

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

        return tmp;
}
```

STACK

main

0→0x2000 34,17 ⋯ 57 0

0x1000 root data tmp

1 nr_Insert_tree

0x1000 → 0x800C 54

root data

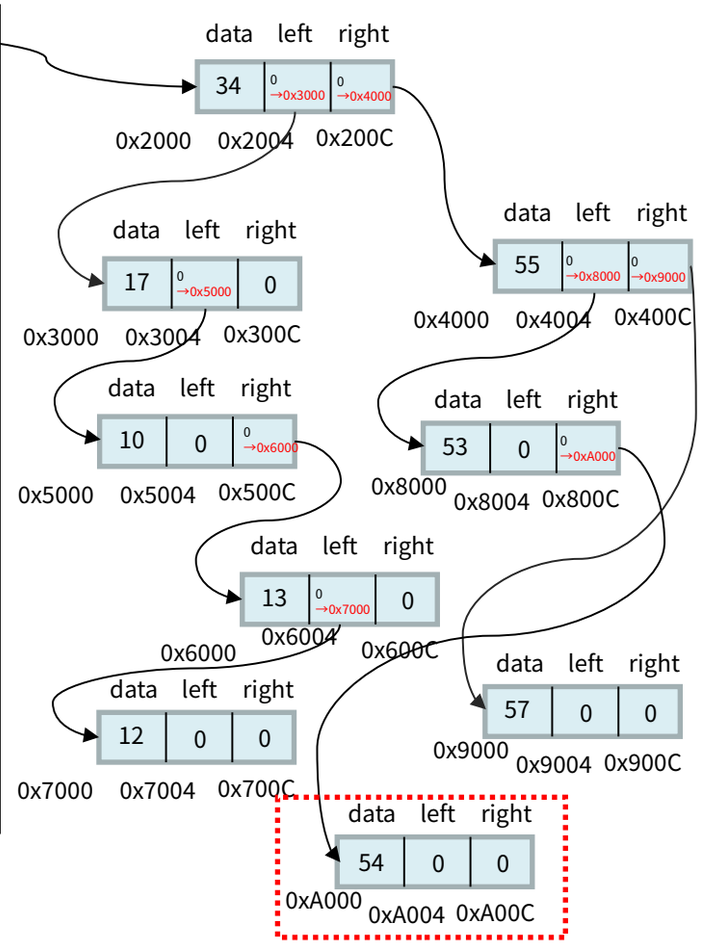
2 While 한번 : Else If (34 < data[54])
root = &(*root)->right;
root는 **0x200C** 가 된다.
While 두번 : if (55 > data[54])
root = &(*root)->left;
root = 0x4004 가 된다.
While 세번 : else if (53 < data[54])
root = &(*root)->right;
root = 0x800C 가 된다.

3 Create_node

0xA000 Malloc은
메모리의 주소 값
반환

tmp

HEAP



2-1) 이중 노드 삭제 (55 삭제)

ㄴ 삭제하려는 node의 왼쪽 max 값 찾고 node change 후 delete 하는 방식

```
// 21.10.30 추가
// 양쪽 모두의 node가 있는 tree 삭제 하기
// find로 max 값 찾고, node change 후 nr_delete
tree *chg_node(tree *root)
{
        tree *tmp = root;

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

        free(tmp);

        return root;
}

void find_max(tree **root, int *data)
{
        tree **tmp = root; // 이건 없어도 됨

        while(*tmp)
        {
                if((*tmp)->right)
                        tmp = &(*tmp)->right;

                else
                {
                        *data = (*tmp)->data;
                        *tmp = chg_node(*tmp);
                        break;
                }
        }
}
```

```
nr_insert_tree_data(&root, 54);
printf("55 삭제\n");
//delete_tree_data(&root, 55);
nr_delete_tree(&root, 55);
print_tree(root);
```

```
void nr_delete_tree(tree **root, int data)
{
        tree **tmp = root;
        int num;

        while(*tmp)
        {
                if((*tmp)->data > data)
                        tmp = &(*tmp)->left;
                else if((*tmp)->data < data)
                        tmp = &(*tmp)->right;
                else if((*tmp)->left && (*tmp)->right)
                {
                        find_max(&(*tmp)->left, &num);
                        (*tmp)->data = num;
                        return;
                }
                else
                {
                        (*tmp) = chg_node(*tmp);
                        return;
                }
        }

        printf("Not Found\n");
}
```

change_node

0x800C → 0 | 0x800C | (*tmp)는 0x800C

root | tmp

Left에 값이 없으므로
Else if(!root->left)
Root = root->right;
Root = 0

STACK

main

0→0x2000 | 34,17 ⋯ 57 | 0

0x1000 root | data | tmp

① nr_delete_tree

0x1000 root | 55 data | 54 num (0xF000) | 0x1000 → 0x200C tmp

While 한번 : else if (34 < data[55])
tmp = &(*tmp)->right;
tmp = 0x200C;
While 두번 : else if(*tmp)->left && (*tmp)->right
(55 node는 왼쪽 오른쪽이 다 존재)

② Find_max

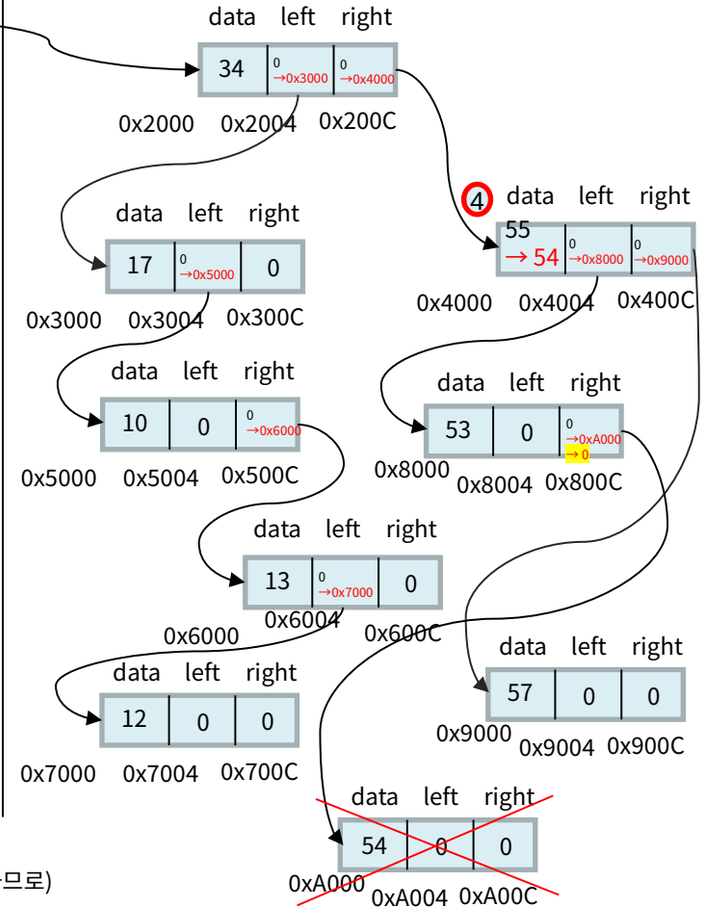
&(*tmp)->left 는 0x4004
Num 주소는 0xF000

0x4004 root | 0xF000 data | tmp 0x4004 → 0x800C

While 한번 : if (*tmp)->right가 존재 한다면
Tmp = &(*tmp)->right;
Tmp = 0x800C
While 두번 : tmp의 right가 존재하지 않으므로,
Else
(*tmp)->data 는 54인데, ***data는 num** 이죠.
체인지 노드 함수로 들어가는데
이때 *tmp를 넘겨줘
(*tmp) 자체는 0x800C 입니다. (*tmp는 tmp의 값을 의미 하므로)

③ Root 0 을 return 받았으므로 *tmp는 0 이 된다.

HEAP



이해가 어려우므로 한번 더 해보자.

2-2) 이중 노드 삭제 (34 삭제)
ㄴ 삭제하려는 node의 왼쪽 max 값 찾고 node change 후 delete 하는 방식

```
// 21.10.30 추가
// 양쪽 모두의 node가 있는 tree 삭제 하기
// find로 max 값 찾고, node change 후 nr_delete

tree *chg_node(tree *root)
{
        tree *tmp = root;

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

        free(tmp);

        return root;
}

void find_max(tree **root, int *data)
{
        tree **tmp = root; // 이건 없어도 됨

        while(*tmp)
        {
                if((*tmp)->right)
                        tmp = &(*tmp)->right;

                else
                {
                        *data = (*tmp)->data;
                        *tmp = chg_node(*tmp);
                        break;
                }
        }
}
```

```
nr_insert_tree_data(&root, 54);
printf("55 삭제\n");
//delete_tree_data(&root, 55);
nr_delete_tree(&root, 55);
print_tree(root);
```

```
void nr_delete_tree(tree **root, int data)
{
        tree **tmp = root;
        int num;

        while(*tmp)
        {
                if((*tmp)->data > data)
                        tmp = &(*tmp)->left;
                else if((*tmp)->data < data)
                        tmp = &(*tmp)->right;
                else if((*tmp)->left && (*tmp)->right)
                {
                        find_max(&(*tmp)->left, &num);
                        (*tmp)->data = num;
                        return;
                }
                else
                {
                        (*tmp) = chg_node(*tmp);
                        return;
                }
        }

        printf("Not Found\n");
}
```

change_node

0x2004 → 0x5000 (root) 0x2004 (tmp) (*tmp)는 0x800C

Right 에 값이 없으므로
if(!root->right)
Root = root->left;
Root = 0x5000;

STACK

main

0→0x2000 (root) 0x1000 | 34,17 ⋯ 57 (data) | 0 (tmp)

1 nr_delete_tree

0x1000 (root) | 34 (data) | 17 (num) 0xF000 | 0x1000 (tmp)

While 한번 : else if(*tmp)->left && (*tmp)->right
(34 node는 왼쪽 오른쪽이 다 존재)

2 Find_max

&(*tmp)->left 는 0x4004
Num 주소는 0xF000

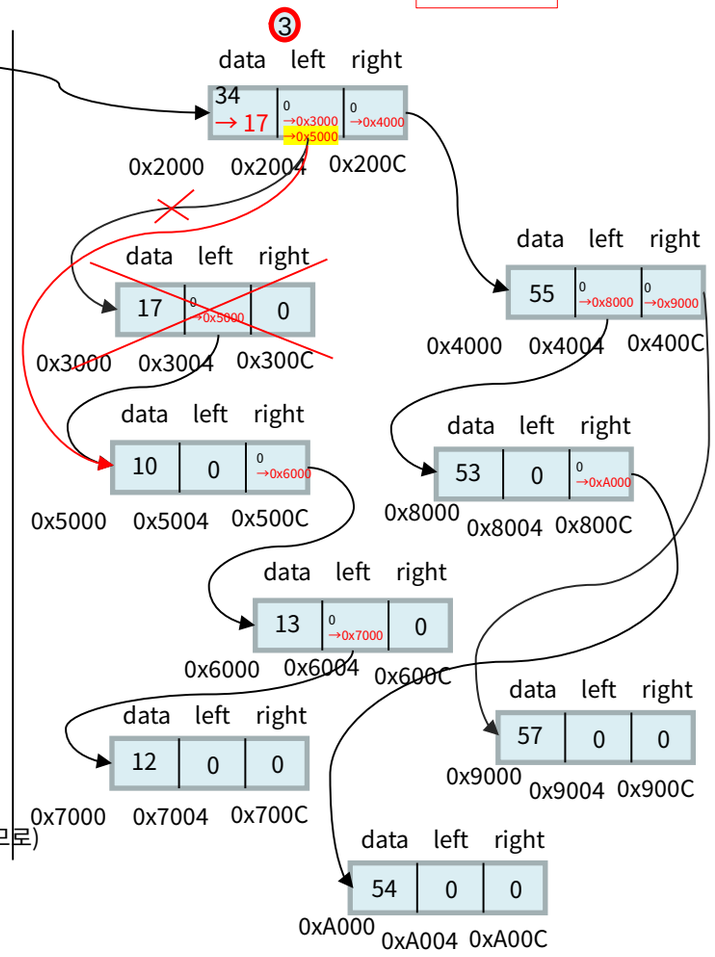
0x2004 (root) | 0xF000 (data) | 0x2004 (tmp)

While 한번 : tmp의 right가 존재하지 않으므로,
Else
(*tmp)->data 는 17인데, ***data는 num** 이죠.

그리고, 체인지 노드 함수로 들어가는데
이때 *tmp를 넘겨줘
(*tmp) 값은 0x2004 입니다. (*tmp는 tmp의 값을 의미 하므로)

3 Root 0x5000 을 return 받았으므로 *tmp는
0x5000 이 된다.

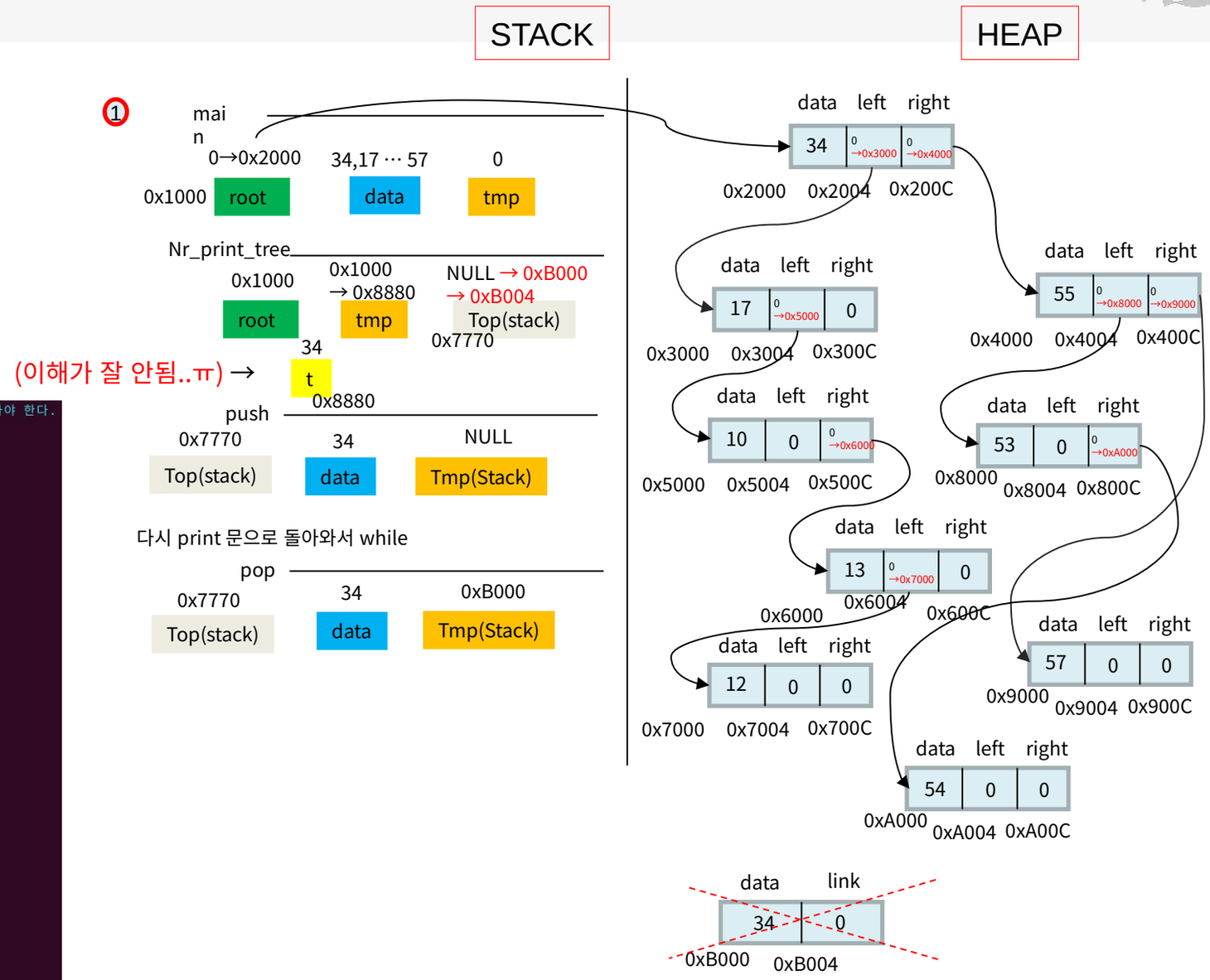
HEAP



3) 비 재귀 방식 print 구현해 보기
ㄴnr_print_tree(&root)

```c
void nr_print_tree(tree **root)
{
        tree **tmp = root;      // 실질적으로는 이것도 void* 타입으로 써야 의미가 통일 되긴 한다.
        stack *top = NULL;

        push(&top, *tmp);

        while(stack_is_not_empty(top))
        {
                tree *t = (tree *)pop(&top); // 여기서 (tree *) 형 변환을 해준 것.
                tmp = &t;

                printf("data = %d, ", (*tmp)->data);

                if((*tmp)->left)
                        printf("left = %d, ", (*tmp)->left->data);
                else
                        printf("left = NULL, ");

                if((*tmp)->right)
                        printf("right = %d\n", (*tmp)->right->data);
                else
                        printf("right = NULL\n");

                push(&top, (*tmp)->right);
                push(&top, (*tmp)->left);
        }
}
```

```c
// 재귀 호출 하지 않는 print 가이드 코드, stack의 push와 pop을  void* 형으로 반환 받아야 한다.
// stack 을 통해서 복귀 주소를 push 하고 pop 으로 다시 돌아간다.
// push, pop은 구현해 보도록 할 것

/*
stack *create_stack_node(void)
{
        stack *tmp;

        tmp = (stack *)malloc(sizeof(stack));
        tmp->link = NULL;

        return tmp;
}

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

        *top = create_stack_node();
        (*top)->data = data;
        (*top)->link = tmp;
}

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

        if (!(*top))
        {
                printf("Stack is empty\n");
                return -1;
        }

        tmp = *top;

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

        free(tmp);

        return data;
}

*/
```



4) Find 더블 및 싱글 포인터 함수

```
        // 밑에 3줄은 find 코드, (if if else 문)
        if(tmp = find_tree_data(&root, 13))
                printf("tmp->data = %d\n", (*tmp)->data);
```

```
tree **find_tree_data(tree **root, int data)
{
    ① while(*root)
      {
              if((*root)->data > data)
                      root = &(*root)->left;
              else if((*root)->data < data)
                      root = &(*root)->right;
              else
                      return root;
      }

      return NULL;
}
```

```
//find 함수이며 delete를 만들 때 find를 사용하면 훨씬 구현이 용이해 진다. insert랑 비슷.
//싱글 호출은 재귀호출 된것의 리턴 값을 받게끔 해야 한다.
//만약 return 타입을 그냥 root로 하려면 반환형이 tree**이 되면 된다. 아래 구현.
/*
tree *find_tree_data(tree **root, int data)
{
        while(*root)
        {
                if((*root)->data > data)
                        root = &(*root)->left;
                else if((*root)->data < data)
                        root = &(*root)->right;
                else
                    ② return *root;
        }

        return NULL;
}*/
```

STACK

HEAP

main

0→0x2000 34,17 ⋯ 57 NULL

0x1000 root data tmp

① ** Find_tree

0x1000 13

root data

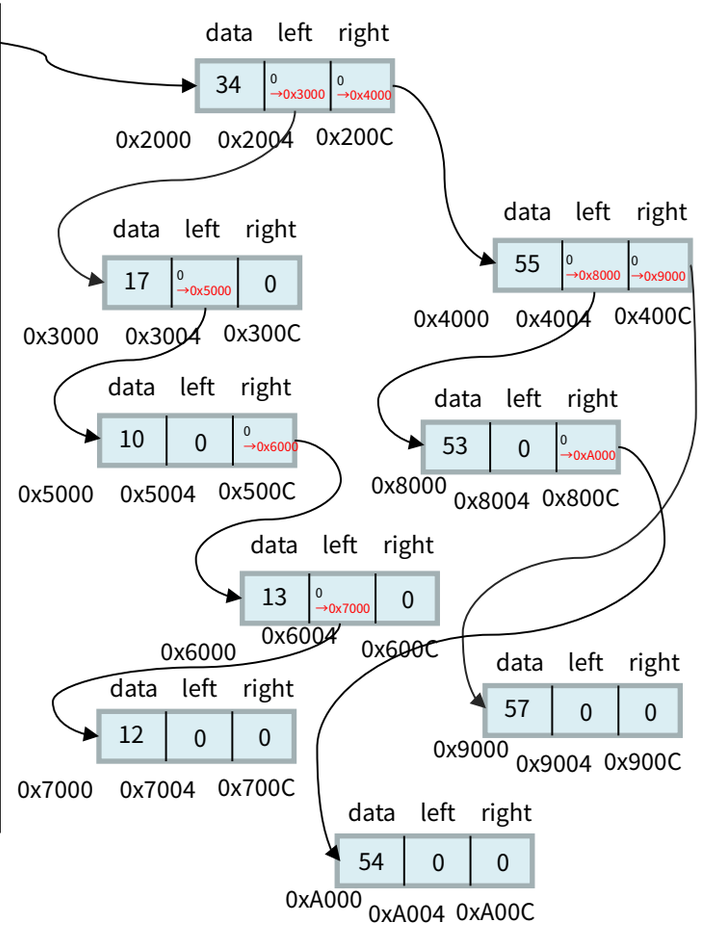
1) 34 > 13
root = &(*root)->left;
root = 0x2004
3) 10 < 13
root = &(*root)->right;
root = 0x500C

2) 17 > 13
root = &(*root)->left;
root = 0x3004

4) Return root(0x500C)

② * Find_tree

Root가 값의 값의 가리키는 값이 Heap 영역의
0x2000을 가르키게 되므로
더블 포인터를 이용해야
두 단계 넘어서의 Heap 영역의
데이터를 수정하기 용이하다.



Code 전체 (1)

```c
void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는 그 주소의 데이터 값을 의미 한다.
        {
                *root = create_tree_node();
                (*root)->data = data;
                return;
        }

        // 언제 왼쪽에 넣어야 하는지
        // 언제 오른쪽에 넣어야 하는지를 판정해야함
        //insert_tree_data(&(*root)->link, data);
        if((*root)->data > data)
        {
                insert_tree_data(&(*root)->left, data);
        }
        else if((*root)->data < data)
        {
                insert_tree_data(&(*root)->right, data);
        }
}

// 재귀호출을 사용하지 않음(nr)

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;
}
```

```c
void nr_print_tree(tree **root)
{
        tree **tmp = root;      // 실직적으로는 이것도 void* 타입으로 써야 의미가 통일 되긴 한다.
        stack *top = NULL;

        push(&top, *tmp);

        while(stack_is_not_empty(top))
        {
                tree *t = (tree *)pop(&top); // 여기서 (tree *) 형 변환을 해준 것.
                tmp = &t;

                printf("data = %d, ", (*tmp)->data);

                if((*tmp)->left)
                        printf("left = %d, ", (*tmp)->left->data);
                else
                        printf("left = NULL, ");

                if((*tmp)->right)
                        printf("right = %d\n", (*tmp)->right->data);
                else
                        printf("right = NULL\n");

                push(&top, (*tmp)->right);
                push(&top, (*tmp)->left);
        }
}
```

```c
// 재귀 호출 하지 않는 print 가이드 코드, stack의 push와 pop을  void* 형으로 반환 받아야 한다.
// stack 을 통해서 복귀 주소를 push 하고 pop 으로 다시 돌아간다.
// push, pop은 구현해 보도록 할 것

/*
stack *create_stack_node(void)
{
        stack *tmp;

        tmp = (stack *)malloc(sizeof(stack));
        tmp->link = NULL;

        return tmp;
}

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

        *top = create_stack_node();
        (*top)->data = data;
        (*top)->link = tmp;
}

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

        if (!(*top))
        {
                printf("Stack is empty\n");
                return -1;
        }

        tmp = *top;

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

        free(tmp);

        return data;
}

*/
```

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

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

typedef struct _stack stack;
struct _stack
{
        void *data;
        struct _stack *link;
};

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}
```

Code 전체 (2)

```c
//아래는 재귀 호출 스타일 print
void print_tree(tree *root)
{
        if(root)
        {
                printf("tree root = %d\n", root->data);
                print_tree(root->left);
                print_tree(root->right);
        }
}

//find 함수이며 delete를 만들 때 find를 사용하면 훨씬 구현이 용이해 진다. insert랑 비슷.
//싱글 호출은 재귀호출 된것의 리턴 값을 받게끔 해야 한다.
//만약 return 타입을 그냥 root로 하려면 반환형이 tree**이 되면 된다. 아래 구현.
/*
tree *find_tree_data(tree **root, int data)
{
        while(*root)
        {
                if((*root)->data > data)
                        root = &(*root)->left;
                else if((*root)->data < data)
                        root = &(*root)->right;
                else
                        return *root;
        }

        return NULL;
}*/
```

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

        return NULL;
}

/* 1차 delete 함수

void delete_tree_data(tree **root, int data)
{
        tree *tmp;
        root = find_tree_data(root, data);

        tmp = *root;


        // 아래 if, else if는 한쪽만 또는 하나도 자식이 없는 경우
        // 이것은 모두 자식이 있는 경우를 지울 때도 해야 하는 과정이기 때문에 함수화 하는게 좋다.
        if(!(*root)->left)
        {
                *root = (*root)->right;
        }
        else if(!(*root)->right)
        {
                *root = (*root)->left;
        }
        // 양쪽 자식이 모두 존재 할 때
        // 여기서 root는 삭제 할 대상을 찾은 것임.
        else
        {
                //int max = proc_left_max(root);
                //int min = proc_right_min(root);
        }

        free(tmp);
}
*/
```

```c
// 21.10.30 추가
// 양쪽 모두의 node가 있는 tree 삭제 하기
// find로 max 값 찾고, node change 후 nr_delete

tree *chg_node(tree *root)
{
        tree *tmp = root;

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

        free(tmp);

        return root;
}

void find_max(tree **root, int *data)
{
        tree **tmp = root; // 이건 없어도 됨

        while(*tmp)
        {
                if((*tmp)->right)
                        tmp = &(*tmp)->right;

                else
                {
                        *data = (*tmp)->data;
                        *tmp = chg_node(*tmp);
                        break;
                }
        }
}
```

```c
void nr_delete_tree(tree **root, int data)
{
        tree **tmp = root;
        int num;

        while(*tmp)
        {
                if((*tmp)->data > data)
                        tmp = &(*tmp)->left;
                else if((*tmp)->data < data)
                        tmp = &(*tmp)->right;
                else if((*tmp)->left && (*tmp)->right)
                {
                        find_max(&(*tmp)->left, &num);
                        (*tmp)->data = num;
                        return;
                }
                else
                {
                        (*tmp) = chg_node(*tmp);
                        return;
                }
        }

        printf("Not Found\n");
}
```

Code 전체 (3)

EDDI
Electronic Design Development Institute

```c
//아래는 결국 delete 3가지 경우의 수를 모두 포함하는 통합형 delete 함수
void delete_tree_data(tree **root, int data)
{
        int num;
        root = find_tree_data(root, data);

#if 0
아래 주석 부분은 위의 수정 과정 이었기에 남겨 둔 것임.

        if (!(*root)->left)
        {
                *root = (*root)->right;
                free(tmp);
        }
        else if (!(*root)->right)
        {
                *root = (*root)->left;
                free(tmp);
        }
        else
        {
                //int max = proc_left_min(root);
                //int min = proc_right_min(root);
        }
        free(tmp);
#endif

        if((*root)->left && (*root)->right)
        {
                find_max(&(*root)->left, &num);
                (*root)->data = num;
        }
        else
        {
                (*root) = chg_node(*root);
        }
}
```

```c
int main(void)
{
        int i;
        tree *root = NULL;
        tree **tmp = NULL; // find 함수 추가 하면서 추가됨
        int data[] = { 34, 17, 55, 10, 13, 12, 53, 57 };

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

        print_tree(root);

        // 밑에 3줄은 find 코드, (if if else 문)
        if(tmp = find_tree_data(&root, 13))
                printf("tmp->data = %d\n", (*tmp)->data);

        if(tmp = find_tree_data(&root, 77))
                printf("tmp->data = %d\n", (*tmp)->data);
        else
                printf("데이터를 찾을 수 없습니다!\n");

        // 밑은 삭제에 대한 구현
        printf("12삭제\n");
        nr_delete_tree(&root, 12);
        print_tree(root);

        printf("10삭제\n");
        nr_delete_tree(&root, 10);
        print_tree(root);

        nr_insert_tree_data(&root, 54);
        printf("55 삭제\n");
        //delete_tree_data(&root, 55);
        nr_delete_tree(&root, 55);
        print_tree(root);


        return 0;
}
```

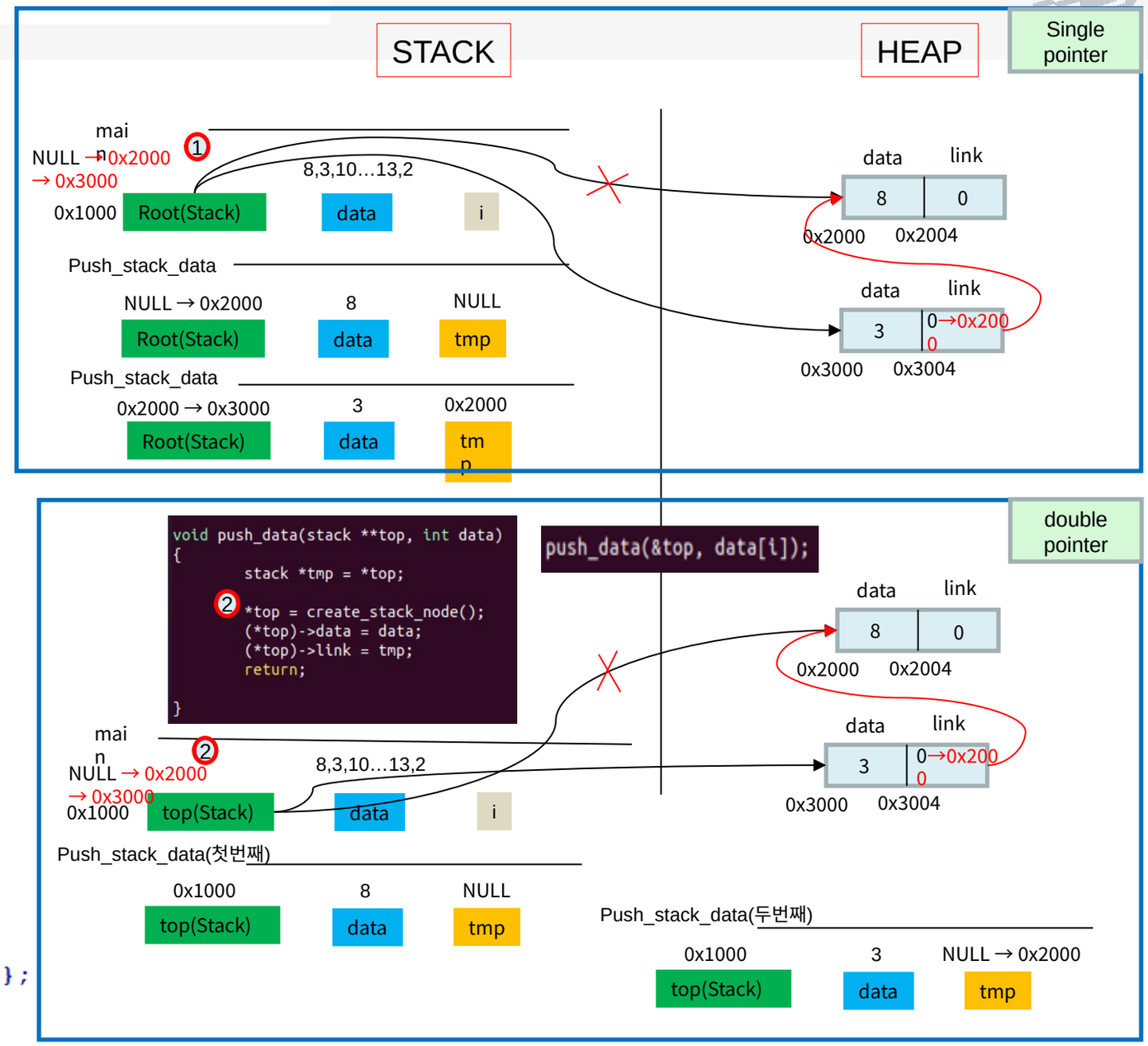
5-1) Single pointer stack 함수 구현
ㄴ push_stack

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

typedef struct _stack stack;
typedef struct _queue queue;
typedef struct _tree tree;

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

```c
stack* push_stack_data(stack* root, int data)
{
    stack* tmp = root;
    root = create_new_stack_node();
    root->data = data;
    root->link = tmp;
    return root;
}

stack* pop_stack_data(stack* root)
{
    stack* tmp = root;
    root = root->link;
    free(tmp);
    return root;
}

int main()
{
    stack* root_stack = NULL;
    queue* root_queue = NULL;
    tree* root_tree = NULL;

    int i;

    int data[10] = {8,3,10,1,6,4,7,14,13,2};
```

```c
stack* create_new_stack_node()
{
    stack* tmp = malloc(sizeof(stack));
    tmp->link = NULL;
    return tmp;
}
```

```c
for(i=0;i<10;i++)
    root_stack = push_stack_data(root_stack, data[i]);
```



결국 두 구현방법이 차이는

Single pointer의 경우
Push stack에서 얻게 된
Heap 영역의 root 주소 값을
Main으로 return 해주어야
그 값을 지칭 할 수 있게 되고,

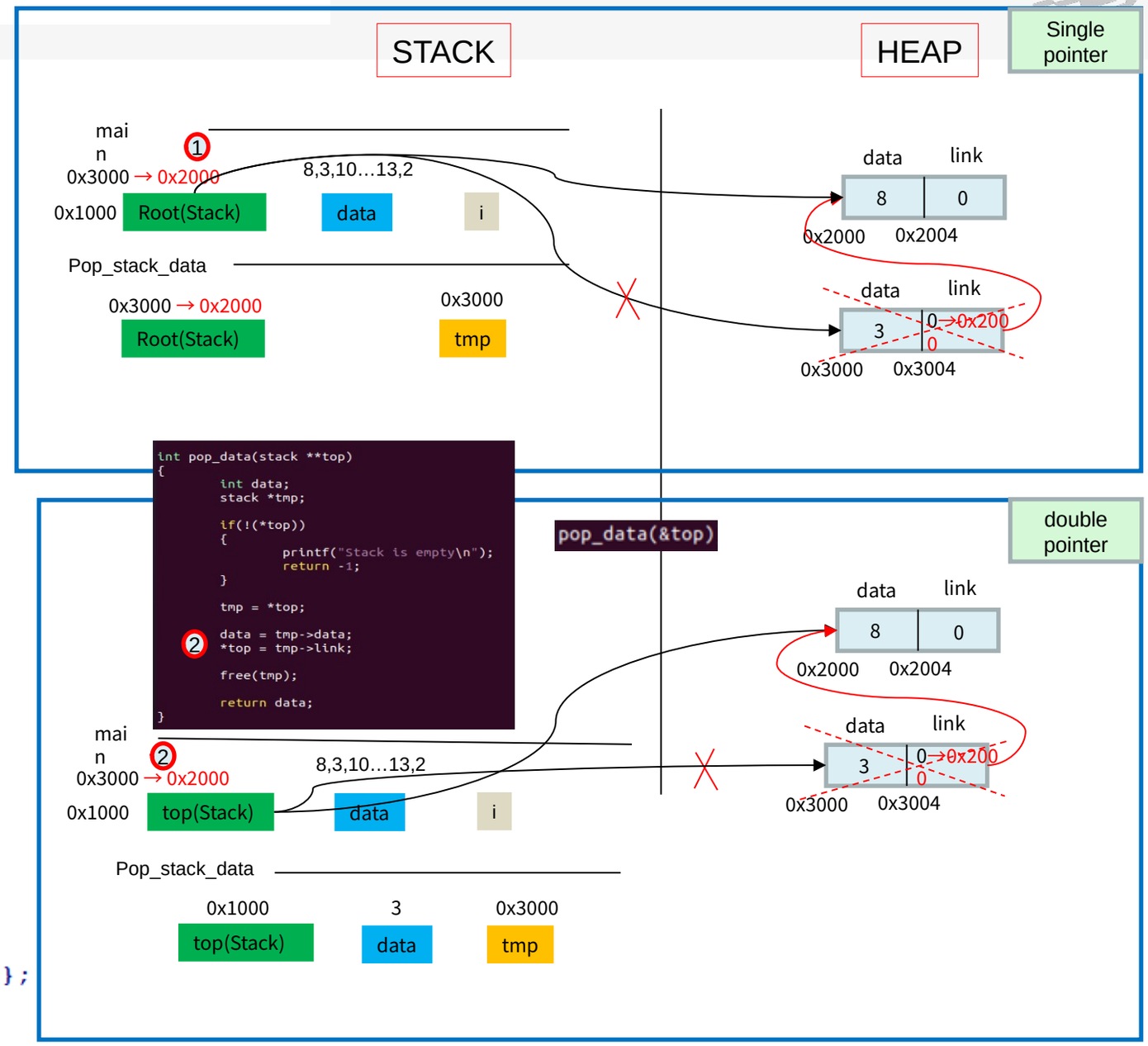
Double pointe의 경우
Push_data 함수 내에서도
Main 의 top을 지칭 할수 있어
top을 return 할 필요 없이
Main 에서의 Heap 영역의
메모리 지칭 변경이 가능해 진다.

5-2) Single pointer stack 함수 구현
ㄴ pop_stack

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

typedef struct _stack stack;
typedef struct _queue queue;
typedef struct _tree tree;

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

stack* push_stack_data(stack* root, int data)
{
    stack* tmp = root;
    root = create_new_stack_node();
    root->data = data;
    root->link = tmp;
    return root;
}

stack* pop_stack_data(stack* root)
{
    stack* tmp = root;
    root = root->link;
    free(tmp);
    return root;
}

int main()
{
    stack* root_stack = NULL;
    queue* root_queue = NULL;
    tree* root_tree = NULL;

    int i;

    int data[10] = {8,3,10,1,6,4,7,14,13,2};
```

```c
stack* create_new_stack_node()
{
    stack* tmp = malloc(sizeof(stack));
    tmp->link = NULL;
    return tmp;
}
```

```c
for(i=0;i<10;i++)
    root_stack = pop_stack_data(root_stack);
```



결국 두 구현방법이 차이는

Single pointer의 경우
Push stack에서 얻게 된
Heap 영역의 root 주소 값을
Main으로 return 해주어야
그 값을 지칭 할 수 있게 되고,

Double pointe의 경우
Push data 함수 내에서도
Main 의 top을 지칭 할수 있어
top을 return 할 필요 없이
Main 에서의 Heap 영역의
메모리 지칭 변경이 가능해 진다.

6-1) Single pointer Queue 함수 구현
ㄴ insert_queue

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

typedef struct _stack stack;
typedef struct _queue queue;
typedef struct _tree tree;

struct _queue
{
    int data;
    struct _queue* link;
};

queue* insert_queue_data(queue* root, int data)
{
    queue* tmp = root;

    if(!root)
    {
        root = create_new_queue_node();
        root->data = data;
        return root;
    }
    else
    {
        while(root->link)
        {
            root = root->link;
        }
        root->link = create_new_queue_node();
        root->link->data = data;
        return tmp;
    }
}
int main()
{
    stack* root_stack = NULL;
    queue* root_queue = NULL;
    tree* root_tree = NULL;

    int i;

    int data[10] = {8,3,10,1,6,4,7,14,13,2};
```

Tmp 를 return 해서 main의 root가 data '8'을 가르 킬 수 있도록 하였다.

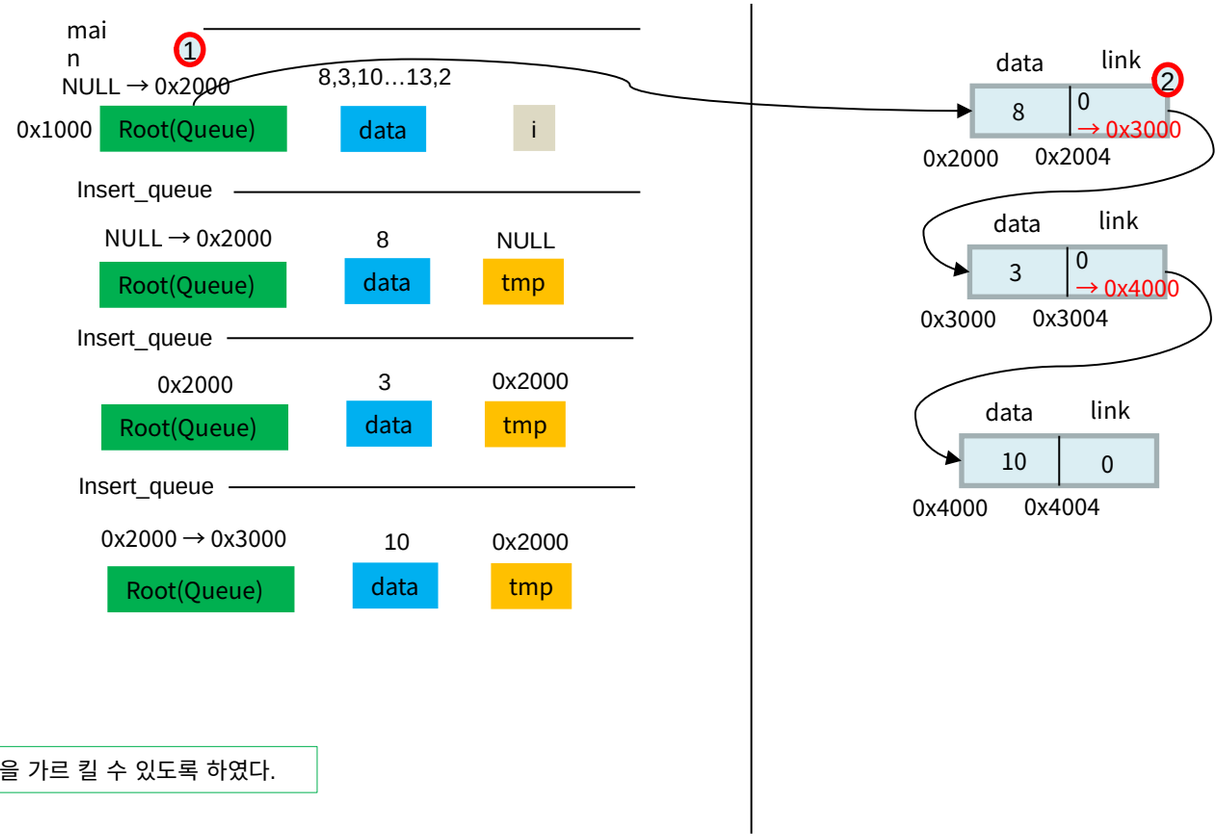
```c
queue* create_new_queue_node()
{
    queue* tmp = malloc(sizeof(queue));
    tmp->link = NULL;
    return tmp;
}
```

Main 문

```c
for(i=0;i<10;i++)
    root_queue = insert_queue_data(root_queue, data[i]);
```

STACK

HEAP



6-2) Single pointer Queue 함수 구현
ㄴ delete_queue

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

typedef struct _stack stack;
typedef struct _queue queue;
typedef struct _tree tree;

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

```c
queue* delete_queue_data(queue* root)
{
    printf("deletequeue\n");
    queue* tmp = root;
    root = root->link;
    free(tmp);
    return root;
}
```

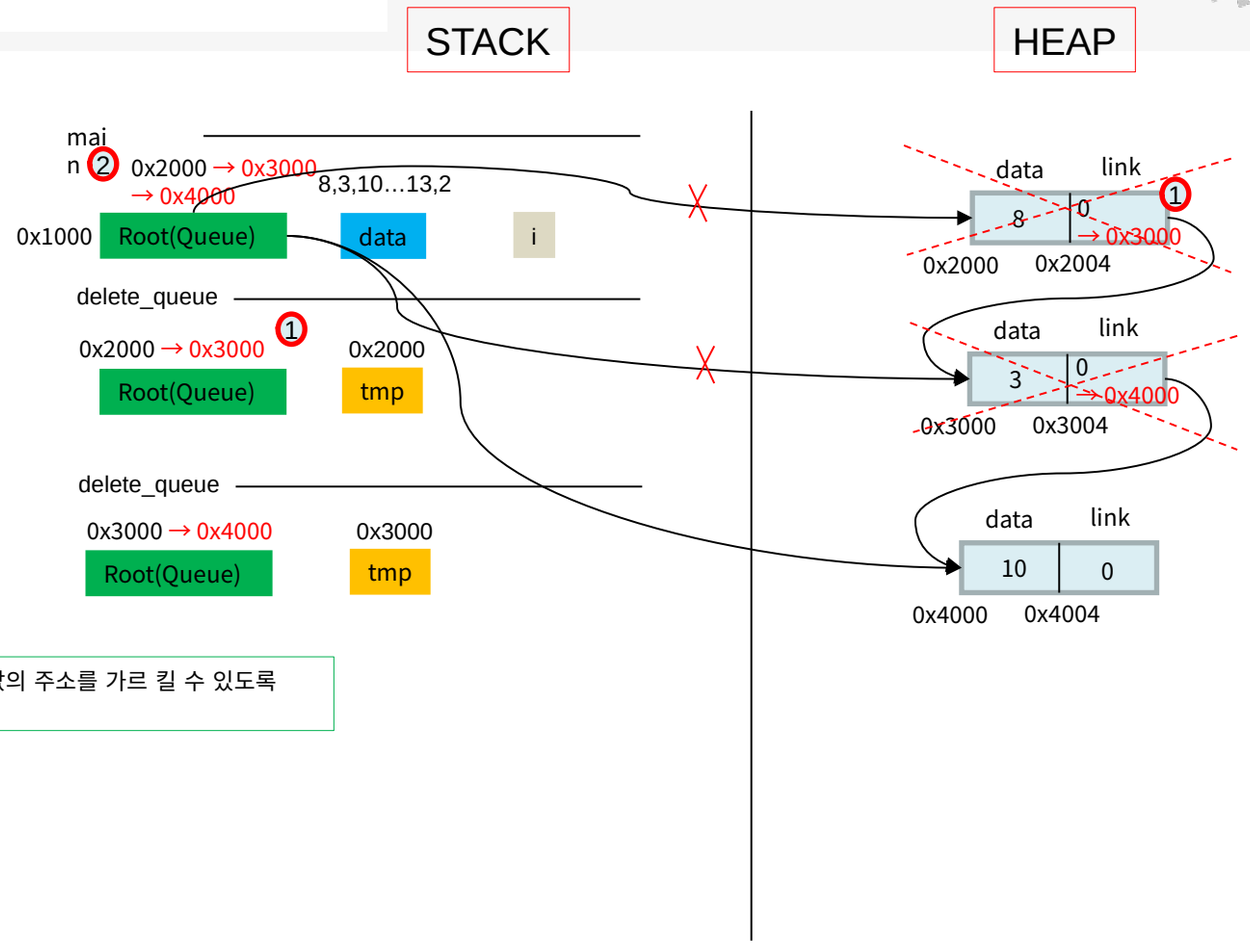
root 를 return 해서 main의 root가 다음 값의 주소를 가르 킬 수 있도록 하였다.

```c
int main()
{
    stack* root_stack = NULL;
    queue* root_queue = NULL;
    tree* root_tree = NULL;

    int i;

    int data[10] = {8,3,10,1,6,4,7,14,13,2};
```

```c
queue* create_new_queue_node()
{
    queue* tmp = malloc(sizeof(queue));
    tmp->link = NULL;
    return tmp;
}
```

Main 문

```c
for(i=0;i<10;i++)
    root_queue = delete_queue_data(root_queue);
```

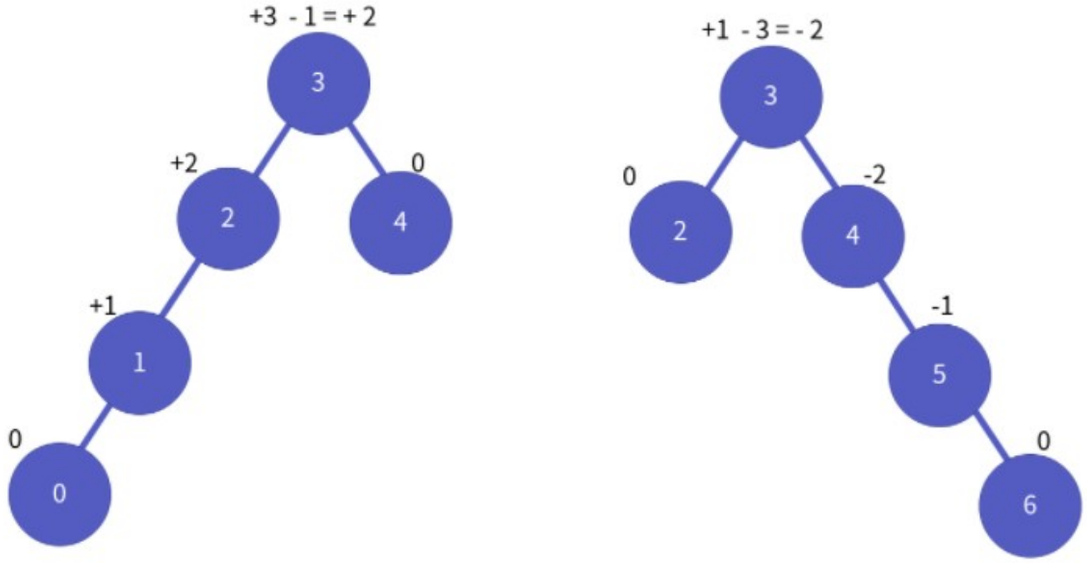


AVL 트리 개념(1)

AVL 트리는 **스스로 균형을 잡는 이진 탐색 트리** 입니다.

트리의 높이가 h 일 때, 복잡도는 0(h) 입니다.
한쪽으로 치우친 편향 이진 트리가 되면 트리의 높이가 높아지기 때문에 이를 방지하고자 높이 균형을 유지하는 AVL 트리를 사용하게 됩니다.

- AVL 트리의 특징
  - ㄴ AVL 트리는 이진 탐색 트리의 속성을 가진다.
  - ㄴ 왼쪽, 오른쪽 서브 트리의 높이 차이가 최대 1 이다.
  - ㄴ 어떤 시점에서 높이 차이가 1보다 커지면 회전(rotation)을 통해 균형을 잡아 높이 차이를 줄입니다.



좌측의 둥근 숫자 위 숫자들은 높이를 나타내는 것인데,
예를 들어 좌측은 아래에서 부터
좌측 또는 우측 높이를 0 1 2 3, 0 1 이런 식으로 나타내고 결과치는 좌측-우측 값 이다.

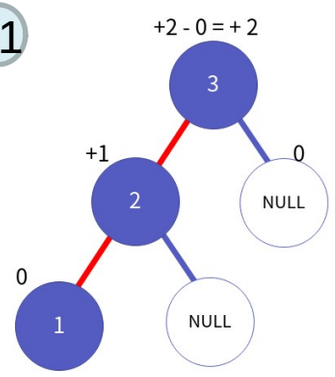
그러나 이러한 그림은 트리의 균형이 깨졌다고 볼 수 있으며,
4가지 회전을 통해서 스스로 균형을 잡게 됩니다.

다음 페이지에 이 4가지 방법에 대해 각각 설명 합니다.

AVL 트리 개념(2)

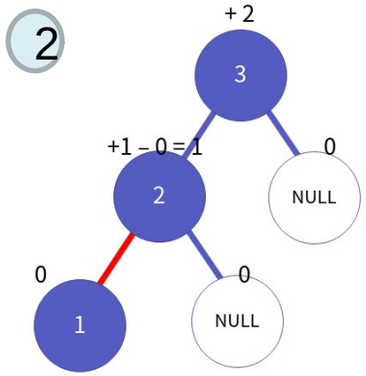
◆ LL 회전

1



위와 같은 상태의 트리가 있다고 가정 할 때,
루트 노드 기준으로, 왼쪽의 높이는 2
오른쪽의 높이는 0

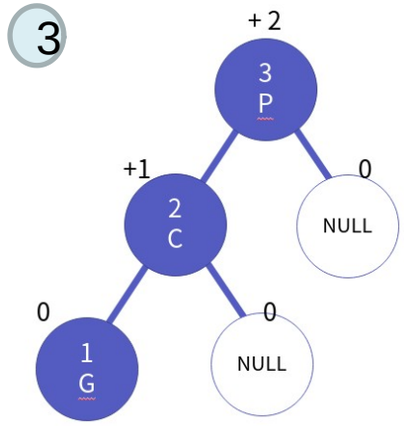
2



또한, 루트 노드의 왼쪽 자식 노드 기준으로는
왼쪽 높이는 1, 오른쪽 높이는 0

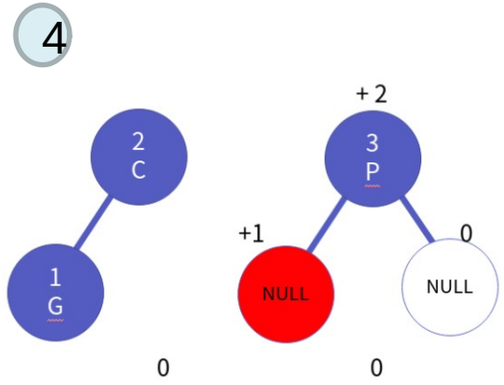
이런 상태,
루트 노드 기준으로 **왼쪽 높이 – 오른쪽 높이 > 1**
왼쪽 자식 노드 기준에서도 **또, 왼쪽 높이 – 오른쪽 높이 > 1**
“LL 상태” 라고 합니다.

3



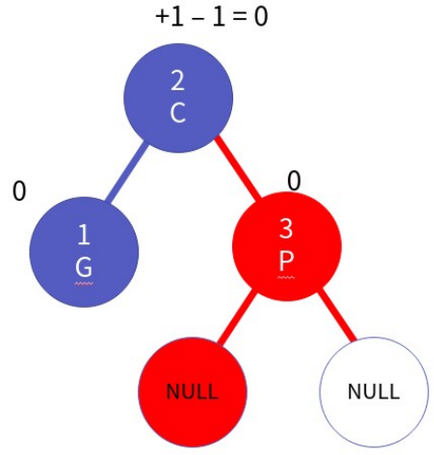
LL 회전은 다음과 같이 일어 납니다.
먼저 루트 노드를 p, 왼쪽 자식 노드를 c,
그 노드의 왼쪽 자식 노드를 g 라고 합시다.

4



이 경우 LL 회전은 먼저 부모 노드 p의 왼쪽 자식 노드를,
자식노드 c의 오른쪽 자식 노드로 바꿉니다.

5



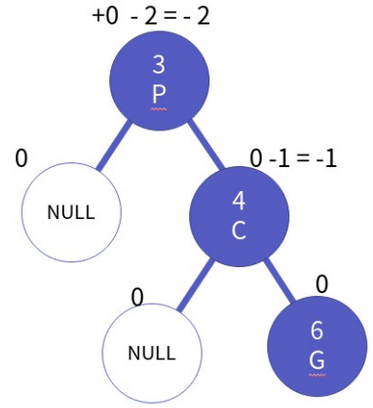
그 후, 자식 노드였던 c의
오른쪽 자식 노드를 부모 노드 p로 합니다.

AVL 트리 개념(3)

EDDI
Electronic Design Development Institute

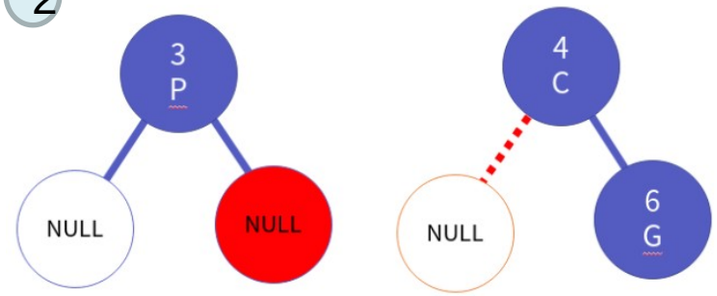
◆ RR회전

1



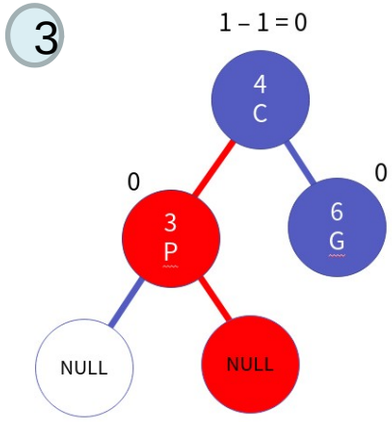
이런 상태,
루트 노드 기준으로 **오른쪽 높이 – 왼쪽 높이 > 1**
왼쪽 자식 노드 기준에서도 **또, 오른쪽 높이 – 왼쪽 높이 > 1**
“RR 상태” 라고 합니다.

2



RR 회전은 먼저 부모 노드 p의 오른쪽 자식 노드를,
자식 노드 c의 왼쪽 자식 노드로 바꿉니다.

3



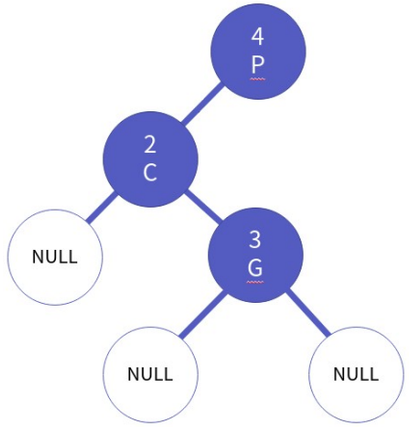
그 후, 자식 노드였던
C의 왼쪽 자식 노드를 부모 노드 p로 합니다.

AVL 트리 개념(4)

EDDI
Electronic Design Development Institute

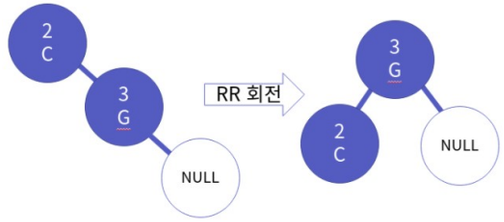
◆ LR회전

1



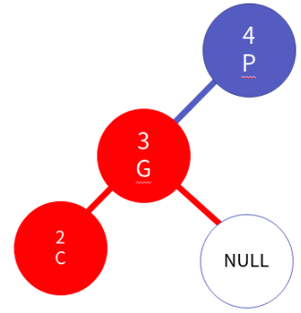
이런 상태,
루트 노드 기준으로 **왼쪽 높이 – 오른쪽 높이 > 1**
왼쪽 자식 노드 기준에서도 **또, 오른쪽 높이 – 왼쪽 높이 > 1**
“LR 상태” 라고 합니다.

2



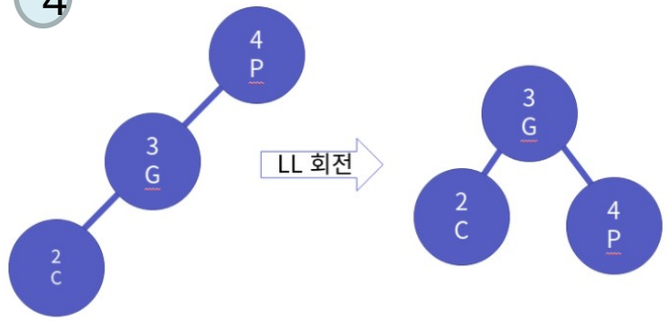
LR 회전은 자식 노드 c에 대해서
RR 회전을 진행 합니다.

3



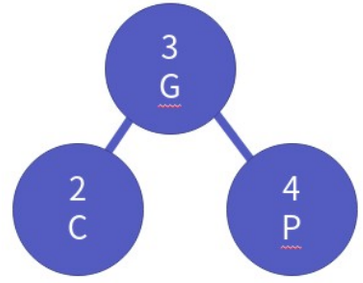
그러면 위와 같이 지게 된다.

4



그 후, 부모 노드 p 에 대해서 LL 회전을 수행하면 됩니다.

5



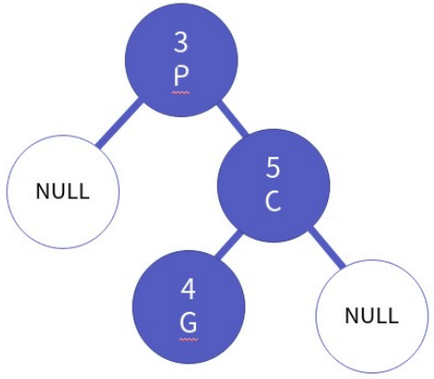
결국, LR 회전의 결과는 위와 같다.

AVL 트리 개념(5)

EDDI
Electronic Design Development Institute

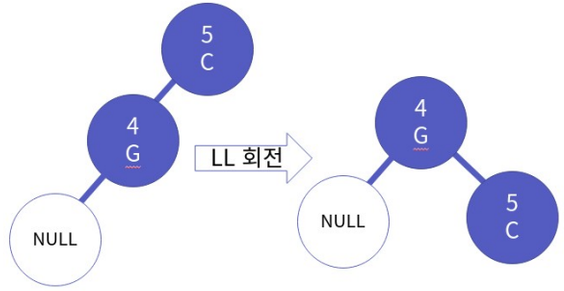
◆ RL회전

1



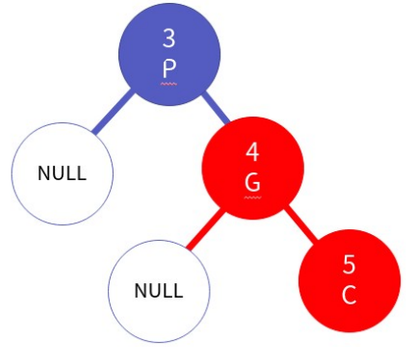
이런 상태,
루트 노드 기준으로 **오른쪽 높이 – 왼쪽 높이 > 1**
왼쪽 자식 노드 기준에서도 **또, 왼쪽 높이 – 오른쪽 높이 > 1**
"RL 상태" 라고 합니다.

2



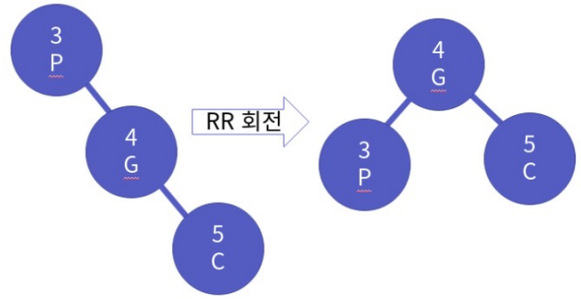
RL 회전은 자식 노드 c 에 대하여
LL 회전을 진행 합니다.

3



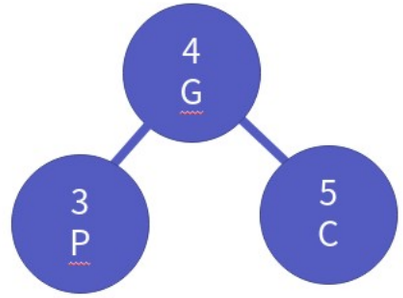
그러면 위와 같이 지게 된다.

4



그 후, 부모 노드 p 에 대해서 RR 회전을 수행하면 됩니다.

5

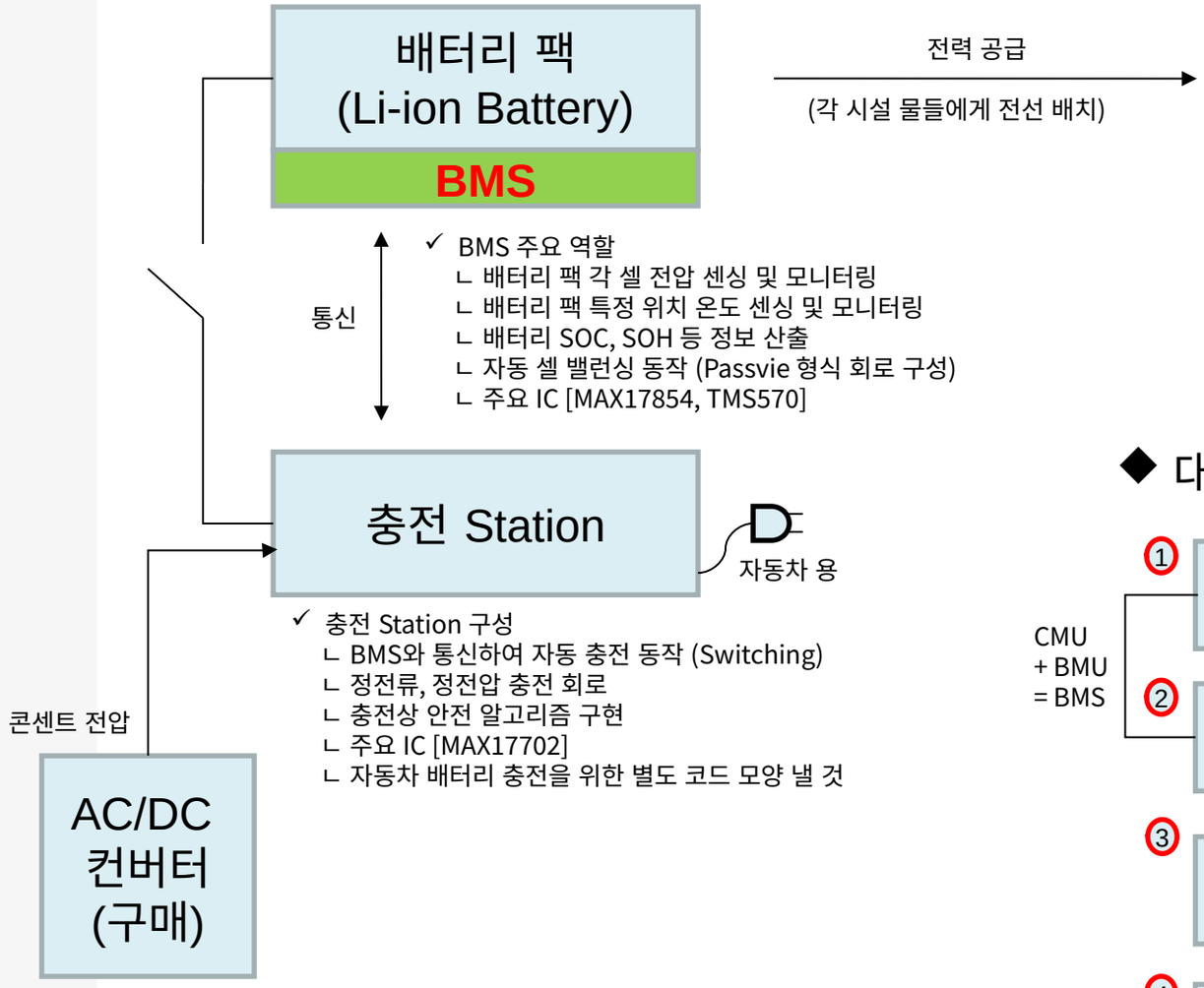


결국, RR 회전의 결과는 위와 같다.

※ 결국 AVL 트리는 이진 탐색 트리에서 삽입/삭제 연산에, 이런 균형을 잡는 회전을 전 노드에 대해서 재귀적으로 수행하는 과정을 추가하는 트리 입니다.

BMS 개발 milestone

◆ 전체 그림



도시

◆ 대략 개발 순서

CMU
+ BMU
= BMS

① **CMU 개발**
- Cell Monitoring Unit 으로 BMS의 전압, 온도 센싱 역할
- MAX17854 활용한 Kit 회로 참고하여 동작 알고리즘 파악
- SPI 통신으로 머리 IC [TMS570]과 통신

② **BMU 개발**
- Battery Management Unit 으로 BMS의 머리 역할
- CMU에서 받은 정보를 바탕으로 SOC, SOH 알고리즘 추출
- CMU의 Cell Balancing 회로 동작
- 충전 Station 과 통신하며 충전 동작 구현

③ **충전 회로 구현**
- 정전류, 정전압 방식 충전 회로 구현 및 시험
- 안전 동작 IC 구성
- 통신 및 동작 구현을 하기 위한 MCU 구성(아직 선택 못함)

④ **모니터링 시스템**
- 전압, 전류, 온도 등의 정보를 볼 수 있도록 구현
  ㄴ 방법 선택 : TFT LCD, 앱, 웹 등