EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv2 과정

[자료구조 프로그래밍 – Binary tree]

제 1기

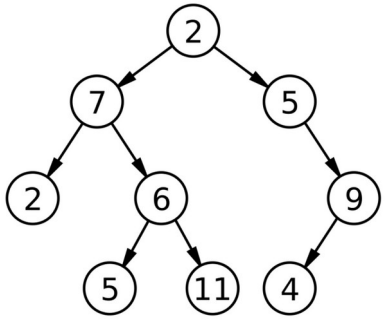
2021. 10. 29

박태인

목차

Binary tree 구조란

- 트리의 기본 개념
  - ㄴ 트리(Tree)란, 배열, 링크드리스트, 스택, 큐와 같이 일직선 개념의 자료구조가 아니라 부모-자식 개념을 가지는 자료구조이다.



기본 이진 트리

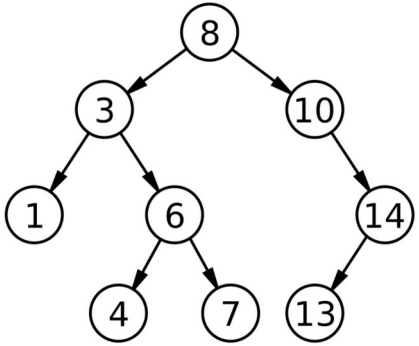
트리는 좌측 그림처럼 표현이 되는 자료구조인데, 여기서 원으로 표시되는 2, 7, 5 등을 노드라고 하고 2는 7과 5의 부모(Parent), 7과 5는 2의 자식(Child)이다.

트리는 그래프의 일종이며, 두 개 이상의 노드가 하나의 노드를 가리킬 수 없다.
즉, 7과 6이 동시에 2를 가리킬 수 없다.
또한, 그렇기에 사이클이 생길 수 없다.

트리에서 최상위 즉, 맨 위에 있는 노드를 루트 노드(root node)라고 하며, 그림에서는 2가 루트 노드가 된다.
루트 노드는 당연히 최상위에 위치하기 때문에 부모를 가지지 않는다.

또한, 트리에서는 더 이상 자식이 없는 노드를 잎 노드(leaf node)라고 표현하고 그 반대로 자식을 하나 이상 가지는 노드를 내부 노드(internal node)라고 한다.

위 그림에서는 2, 5, 11, 4 노드가 leaf node이며, 그 나머지 노드가 internal node가 된다.



이진 탐색 트리

이진 탐색 트리는 기본적인 특징은 이진 트리와 같지만 하나 다른 점은 자기 왼쪽에는 자신보다 값이 작은 노드가, 오른쪽에는 자신보다 값이 큰 노드가 와야한다는 것이다.

그래서 그림상 보면 루트 노드인 노드 8 왼쪽에는 8보다 작은 3, 1, 6, 4, 7이 있고 오른쪽에는 8보다 큰 10, 14, 13이 있다.

이 조건은 꼭 직속 부모-자식 관계에만 영향을 미치는 것이 아니다.

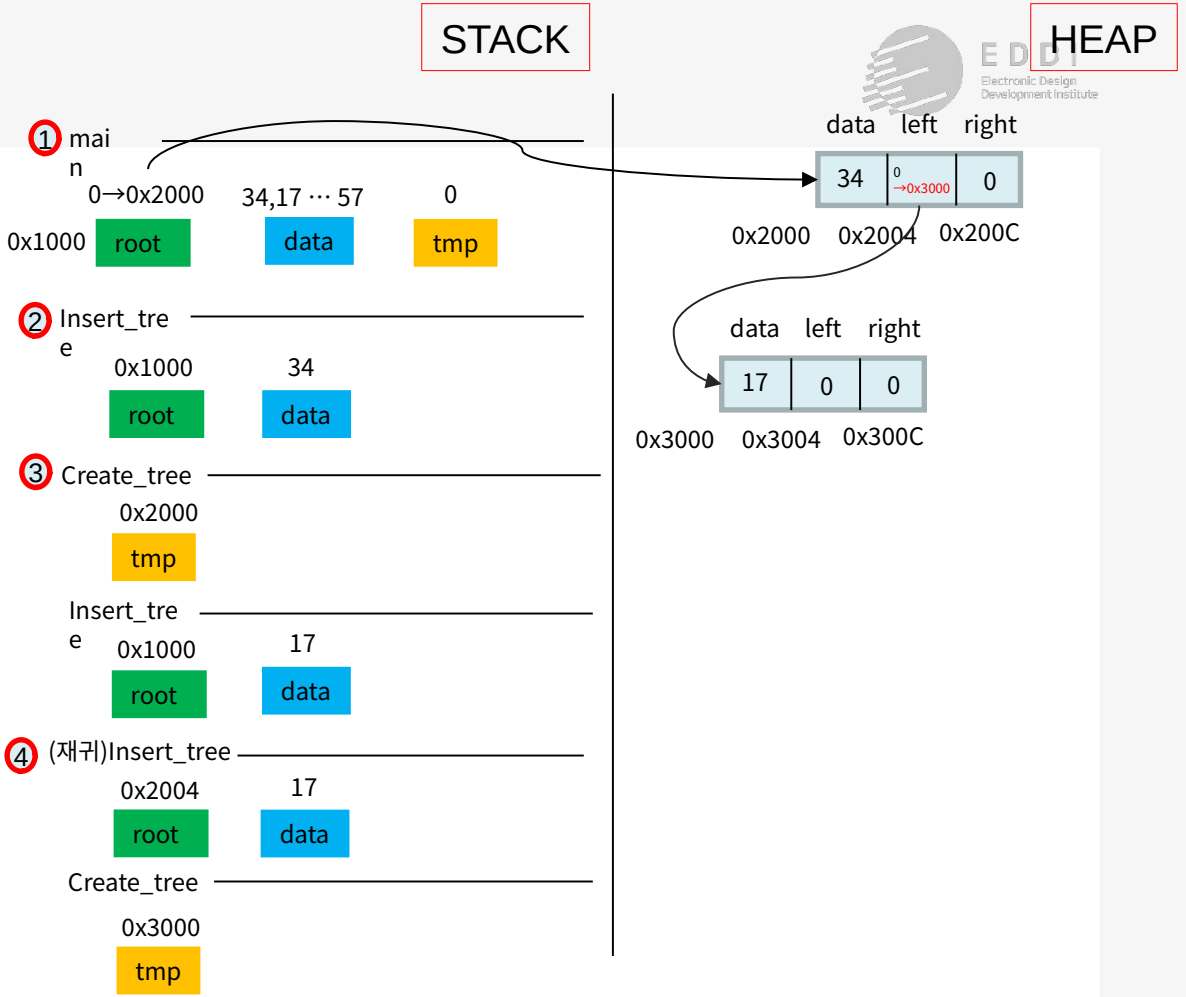
4를 기준으로 생각해보면, 3도 결국엔 자기보다 왼쪽에 있는 노드이므로 자신보다 작아야 하고, 1도 자기보다 왼쪽에 있는 노드이므로 당연히 4보다 작다.
반면, 13은 자기보다 오른쪽에 있으므로 당연히 크다.

이렇게 특정 노드를 기준으로 왼쪽에는 보다 작은 값이, 오른쪽에는 보다 큰 값을 가진 노드가 위치한 트리를 이진 탐색 트리라고 한다.

Insert_tree_data (1)

```c
int main(void)
{
    1  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++)
       {
    2      insert_tree_data(&root, data[i]);
       }
```

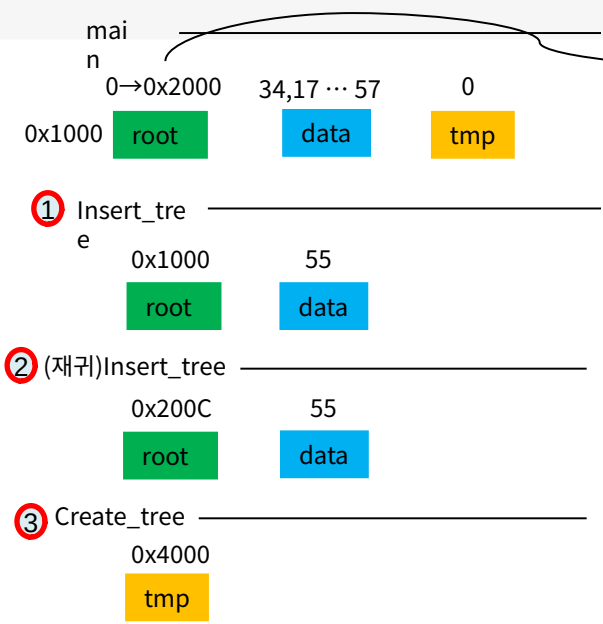
```c
#include <stdlib.h>
#include <stdio.h>

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

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

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



Insert_tree_data (2)

```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++)
        {
            1 insert_tree_data(&root, data[i]);
        }
```

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는
        {
            3 *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);
        }
    2   else if((*root)->data < data)
        {
                insert_tree_data(&(*root)->right, data);
        }
}
```



Insert_tree_data (3)

```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++)
        {
            1 insert_tree_data(&root, data[i]);
        }
```

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는
        {
            4 *root = create_tree_node();
              (*root)->data = data;
              return;
        }

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

STACK

main

0→0x2000 | 34,17 ⋯ 57 | 0

0x1000 | root | data | tmp

1 Insert_tree

0x1000 | 10

root | data

2 (재귀)Insert_tree — 34 와 비교

0x2004 | 10

root | data

3 (재귀)Insert_tree — 17 과 비교

0x3004 | 10

root | data

4 Create_tree

0x5000

tmp

HEAP

	data	left	right
0x2000	34	0 →0x3000	0 →0x4000
0x3000	17	0 →0x5000	0
0x4000	55	0	0
0x5000	10	0	0

0x2004 0x200C | 0x3004 0x300C | 0x4004 0x400C | 0x5004 0x500C

Insert_tree_data (4)

STACK

HEAP

```
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++)
        {
            1 insert_tree_data(&root, data[i]);
        }
```

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는
        {
            5 *root = create_tree_node();
              (*root)->data = data;
              return;
        }

        // 언제 왼쪽에 넣어야 하는지
        // 언제 오른쪽에 넣어야 하는지를 판정해야함
        //insert_tree_data(&(*root)->link, data);
3 2     if((*root)->data > data)
        {
                insert_tree_data(&(*root)->left, data);
        }
    4   else if((*root)->data < data)
        {
                insert_tree_data(&(*root)->right, data);
        }
}
```

main

0→0x2000 | 34,17 … 57 | 0

0x1000 root | data | tmp

1 Insert_tree

0x1000 | 13

root | data

2 (재귀)Insert_tree — 34 와 비교

0x2004 | 13

root | data

3 (재귀)Insert_tree — 17 과 비교

0x3004 | 13

root | data

4 (재귀)Insert_tree — 10 과 비교

0x500C | 13

root | data

5 Create_tree

0x6000

tmp

	data	left	right
0x2000	34	0→0x3000	0→0x4000
0x3000	17	0→0x5000	0
0x4000	55	0	0
0x5000	10	0	0→0x6000
0x6000	13	0	0

Insert_tree_data (5)

```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++)
        {
            1   insert_tree_data(&root, data[i]);
        }
```

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는
        {
            6   *root = create_tree_node();
                (*root)->data = data;
                return;
        }

    5   // 언제 왼쪽에 넣어야 하는지
        // 언제 오른쪽에 넣어야 하는지를 판정해야함
    3 2 //insert_tree_data(&(*root)->link, data);
        if((*root)->data > data)
        {
                insert_tree_data(&(*root)->left, data);
        }
    4   else if((*root)->data < data)
        {
                insert_tree_data(&(*root)->right, data);
        }
}
```

STACK

main

0→0x2000 | 34,17 ⋯ 57 | 0

0x1000 root | data | tmp

1 Insert_tree

0x1000 root | 12 data

2 (재귀)Insert_tree — 34 와 비교

0x2004 root | 12 data

3 (재귀)Insert_tree — 17 과 비교

0x3004 root | 12 data

4 (재귀)Insert_tree — 10 과 비교

0x500C root | 12 data

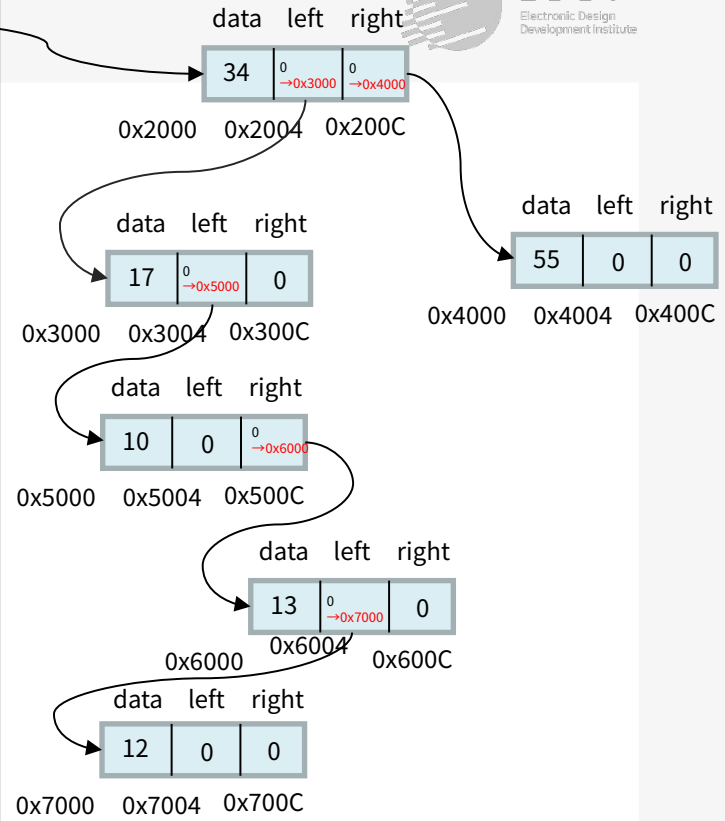
5 (재귀)Insert_tree — 13 과 비교

0x6004 root | 12 data

6 Create_tree

0x7000 tmp

HEAP



Insert_tree_data (6)

```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++)
        {
            1   insert_tree_data(&root, data[i]);
        }
```

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는
        {
            4   *root = create_tree_node();
                (*root)->data = data;
                return;
        }

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

STACK

main

0→0x2000 | 34,17 ⋯ 57 | 0

0x1000 root | data | tmp

1 Insert_tree

0x1000 root | 53 data

2 (재귀)Insert_tree — 34 와 비교

0x200C root | 53 data

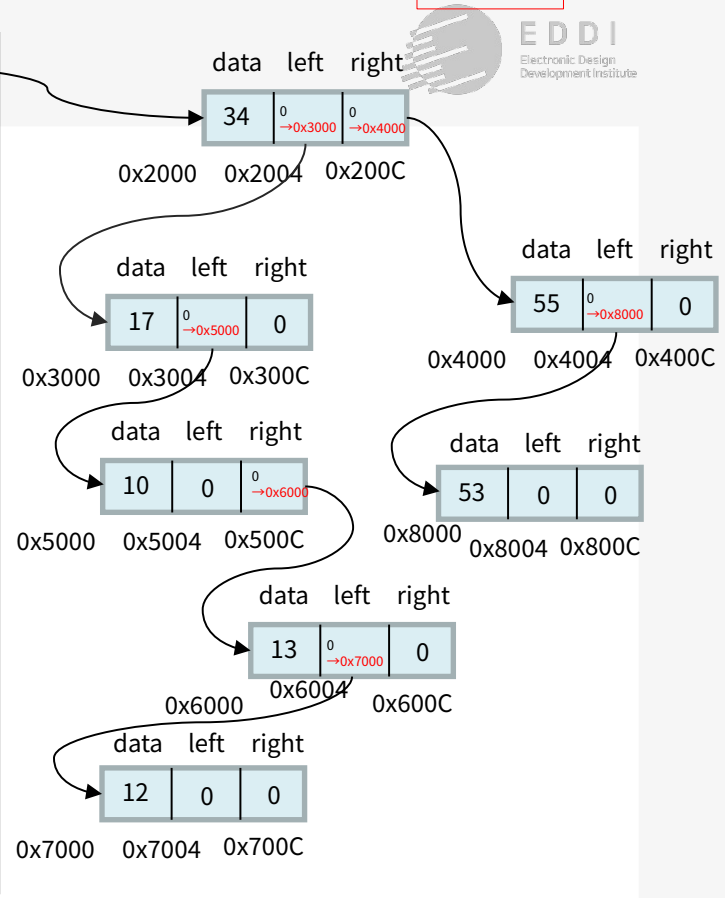
3 (재귀)Insert_tree — 55 와 비교

0x4004 root | 53 data

4 Create_tree

0x8000 tmp

HEAP



Insert_tree_data (7)

```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++)
        {
            1 insert_tree_data(&root, data[i]);
        }
```

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

void insert_tree_data(tree **root, int data)
{
        if (!(*root)) //주소 값을 받은 것 이므로, *root는
        {
            4 *root = create_tree_node();
              (*root)->data = data;
              return;
        }

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

STACK

main

0→0x2000 | 34,17 ⋯ 57 | 0

0x1000 root | data | tmp

1 Insert_tree

0x1000 | 57

root | data

2 (재귀)Insert_tree

34 와 비교

0x200C | 57

root | data

3 (재귀)Insert_tree

55 와 비교

0x400C | 57

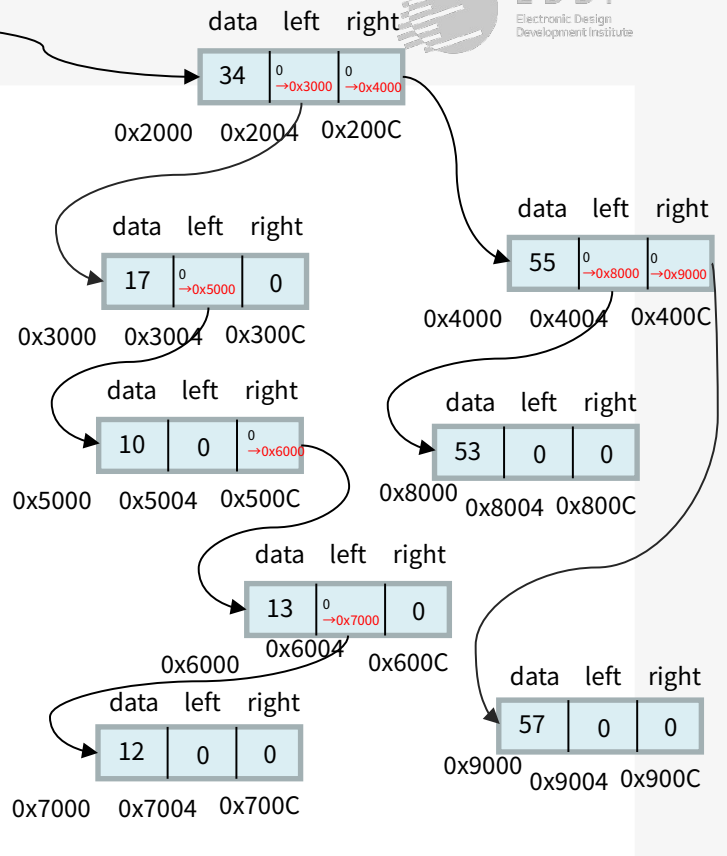
root | data

4 Create_tree

0x9000

tmp

HEAP



delete_tree_data (전략)

◆ 삭제를 할 때는 두가지 전략이 필요하다.

1) left, right 의 값이 둘 다 없을 때
2) left, right 의 값이 한 쪽이라도 없을 때 → 같은 전략!) ex : 왼쪽에 값이 없네요? 오른쪽 return
오른쪽에 값이 없네요? 왼쪽 return

3) left, right 의 값이 둘 다 있을 때 → ex : 1) 좌측에서 최대 값
2) 우측에서 최소 값
이러한 2가지 방법으로 find 해서 지우고 대체!

delete_tree_data (1)

- **Delete_tree_data(&root, 12)**

```
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;
}

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);
}
```

STACK

main

0→0x2000 | 34,17 ⋯ 57 | 0

0x1000 root | data | tmp

① Delete_tree

0x1000 root | 12 data | 0x7000 tmp

② find_tree — 34와 비교

0x2004 root | 12 data

③ (while)find_tree — 17과 비교

0x3004 root | 12 data

④ (while)find_tree — 10과 비교

0x500C root | 12 data

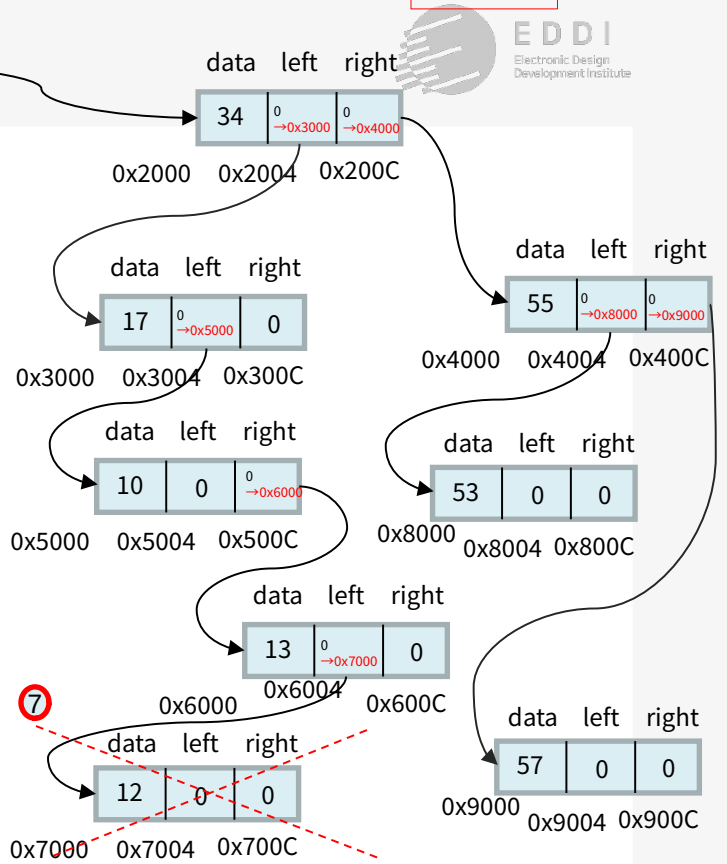
⑤ (while)find_tree — 13과 비교

0x6004 root | 12 data

⑥ (while)find_tree — else

0x7000 root | 12 data

HEAP

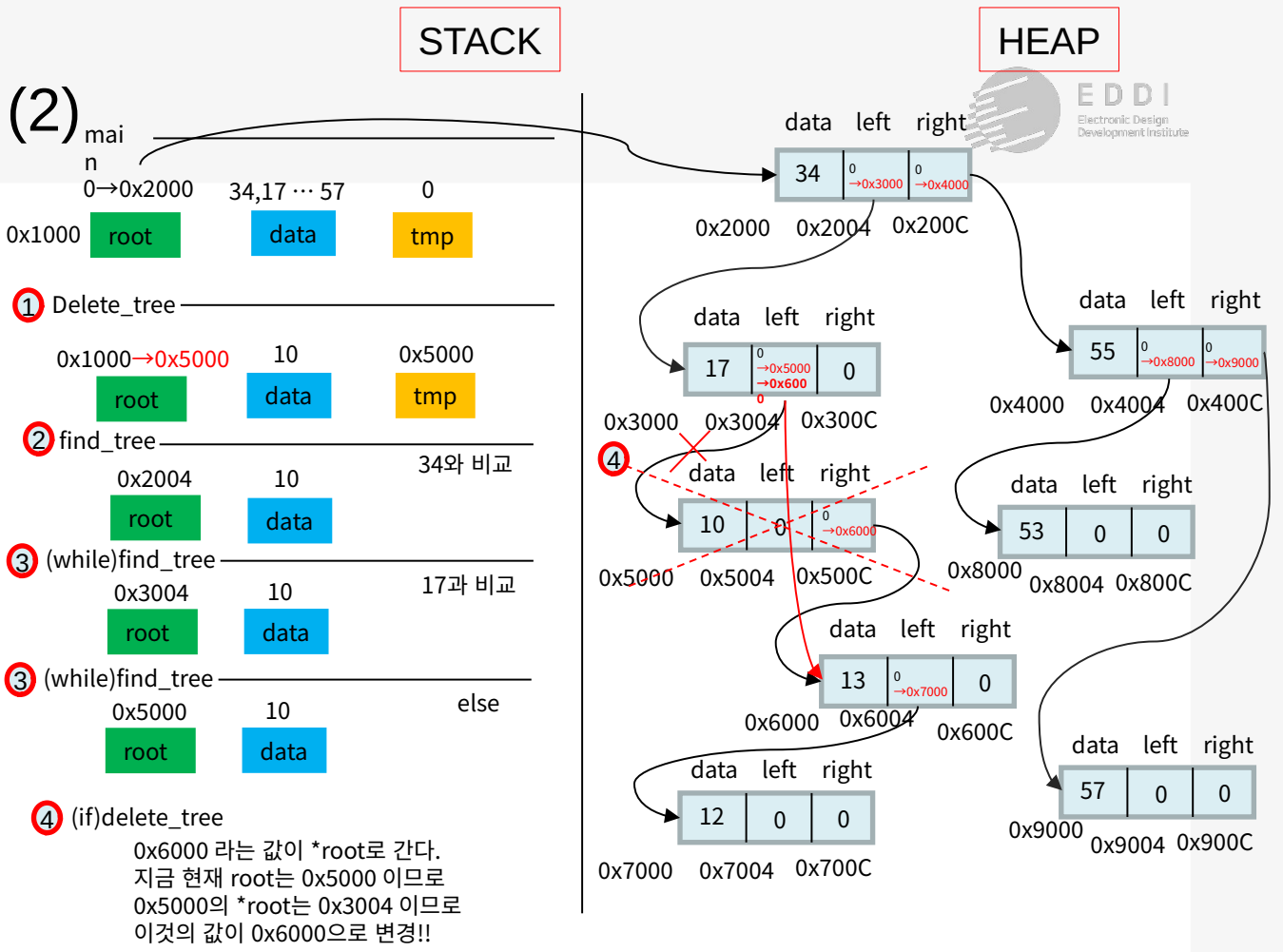


delete_tree_data (2)

- **Delete_tree_data(&root, 10)**

```
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;
}

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);
}
```

2 3 4



Code 전체 (1)

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

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

tree *create_tree_node(void)
{
        tree *tmp;

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

        return tmp;
}

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);
        }
}
```

```c
// 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;
}
*/

void print_tree(tree *root)
{
        if(root)
        {
                printf("tree root = %d\n", root->data);
                print_tree(root->left);
                print_tree(root->right);
        }
}

//find 함수, 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;
}*/
```

Code 전체 (2)

```
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;
}

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);
}
```

```
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");
        delete_tree_data(&root, 12);
        print_tree(root);

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

        return 0;
}
```

Code 전체 (3)


```
tree root = 34
tree root = 17
tree root = 10
tree root = 13
tree root = 12
tree root = 55
tree root = 53
tree root = 57
tmp->data = 13
데이터를 찾을 수 없습니다!
12삭제
tree root = 34
tree root = 17
tree root = 10
tree root = 13
tree root = 55
tree root = 53
tree root = 57
10삭제
tree root = 34
tree root = 17
tree root = 13
tree root = 55
tree root = 53
tree root = 57
```