Data Structures

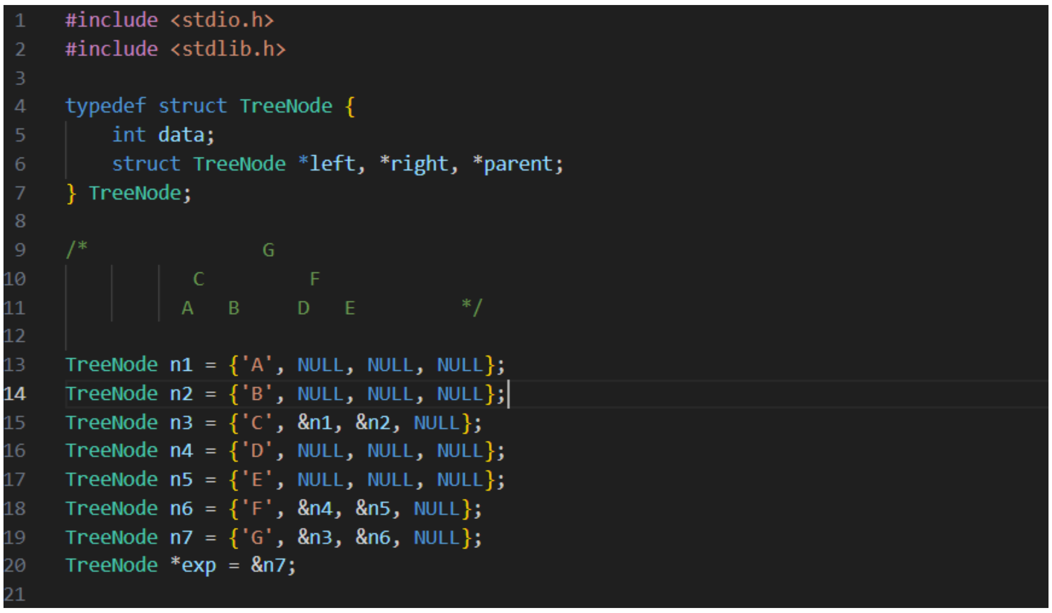
Assignment 05

Technical Report

2176211 Jiwon Yang

I. Problem 2

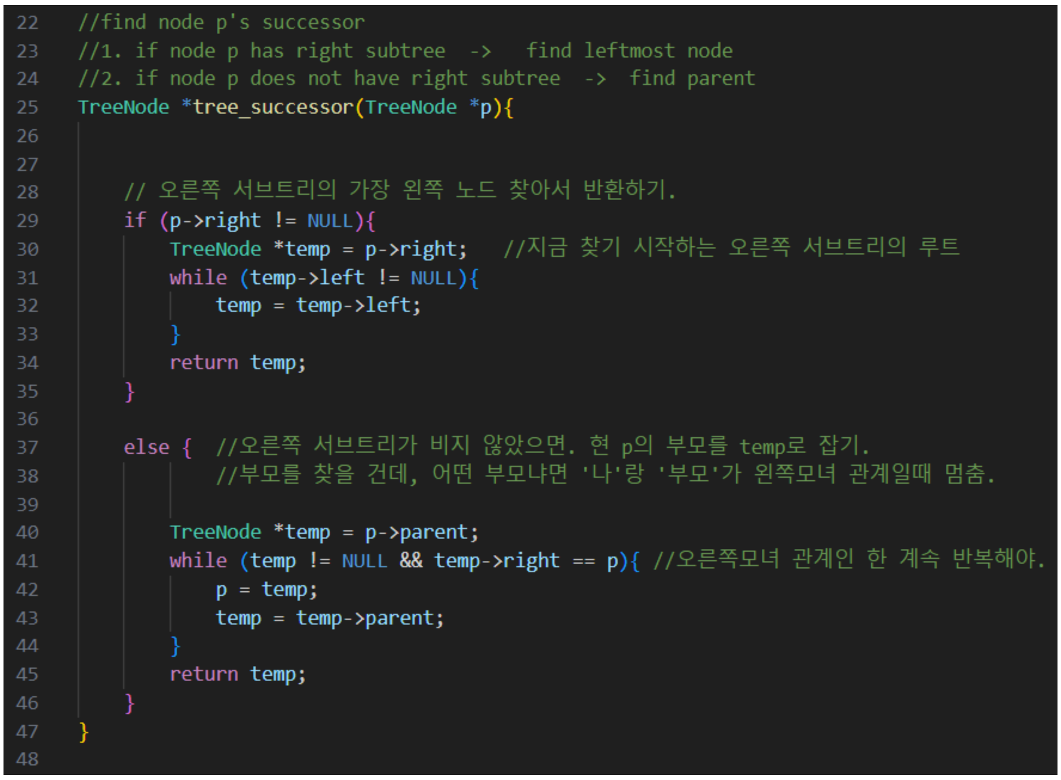
1. Code Explanation



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

typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right, *parent;
} TreeNode;

/*              G
         C          F
        A   B     D   E         */

TreeNode n1 = {'A', NULL, NULL, NULL};
TreeNode n2 = {'B', NULL, NULL, NULL};
TreeNode n3 = {'C', &n1, &n2, NULL};
TreeNode n4 = {'D', NULL, NULL, NULL};
TreeNode n5 = {'E', NULL, NULL, NULL};
TreeNode n6 = {'F', &n4, &n5, NULL};
TreeNode n7 = {'G', &n3, &n6, NULL};
TreeNode *exp = &n7;

```

1. Struct TreeNode has int type data, and TreeNode pointer type left, right, and parent. Each element indicates the node's left child, right child, and parent's address).

2. Create instances of the TreeNode struct and initialize them with data and pointers to their left, right, and parent nodes, based on the tree structure provided in the comment.

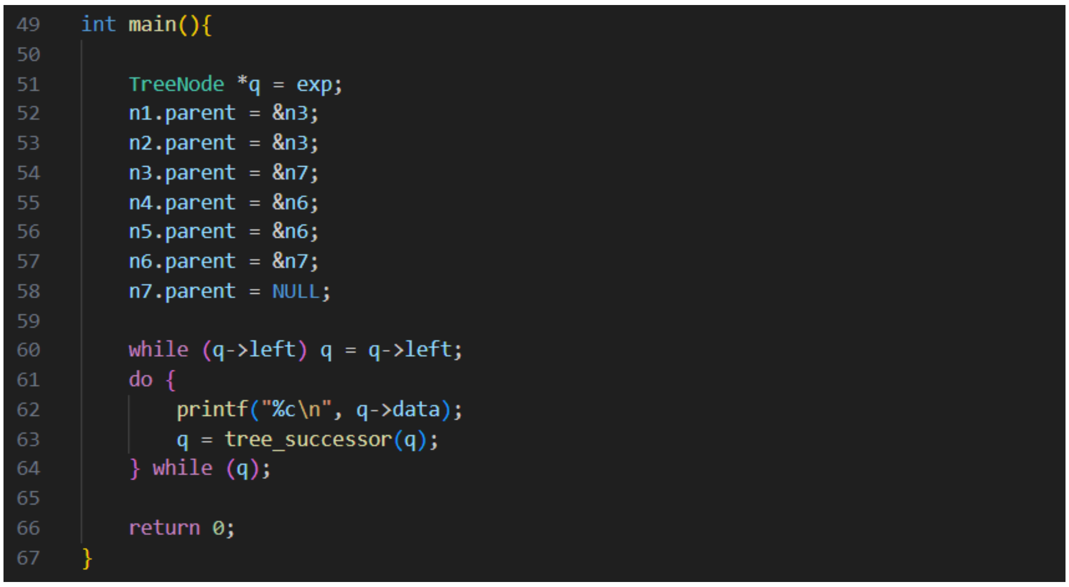


```c
//find node p's successor
//1. if node p has right subtree  ->  find leftmost node
//2. if node p does not have right subtree  ->  find parent
TreeNode *tree_successor(TreeNode *p){


    // 오른쪽 서브트리의 가장 왼쪽 노드 찾아서 반환하기.
    if (p->right != NULL){
        TreeNode *temp = p->right;   //지금 찾기 시작하는 오른쪽 서브트리의 루트
        while (temp->left != NULL){
            temp = temp->left;
        }
        return temp;
    }

    else {  //오른쪽 서브트리가 비지 않았으면. 현 p의 부모를 temp로 잡기.
            //부모를 찾을 건데, 어떤 부모냐면 '나'랑 '부모'가 왼쪽모녀 관계일때 멈춤.

        TreeNode *temp = p->parent;
        while (temp != NULL && temp->right == p){ //오른쪽모녀 관계인 한 계속 반복해야.
            p = temp;
            temp = temp->parent;
        }
        return temp;
    }
}

```

The tree_successor function, which takes TreeNode p as an argument, is used to find and return the successor of p.

1. If p has a right subtree, it searches for the leftmost node in that right subtree to find the successor. To achieve this, it employs a temporary variable called temp. Initially, temp is set to the root of p's right subtree. It continues to move to the left in the right subtree until it encounters a node where the left child is NULL. When this condition is met, the function returns the temp value, which represents the leftmost node in p's right subtree, making it p's successor.

2. If p does not have a right subtree, its successor is found by moving up the tree level by level, searching for the node where p is the left child of its parent. In this case, the temporary variable temp initially points to p's parent node. The loop continues as long as the condition is satisfied, which means that temp is a parent node, and its child relationship with that parent is on the left side. When this condition is no longer met, the loop exits, and the function returns the value of temp, which represents the parent of p. This parent node is the successor of p.

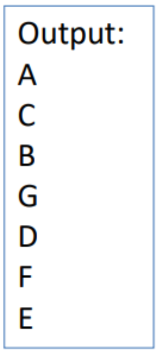


```c
int main(){

    TreeNode *q = exp;
    n1.parent = &n3;
    n2.parent = &n3;
    n3.parent = &n7;
    n4.parent = &n6;
    n5.parent = &n6;
    n6.parent = &n7;
    n7.parent = NULL;

    while (q->left) q = q->left;
    do {
        printf("%c\n", q->data);
        q = tree_successor(q);
    } while (q);

    return 0;
}
```

1. After setting the parent node for each node, make node q point to the root node. Then, position node q to point to the leftmost node in the tree, and move it one by one to its successor to print the tree in inorder traversal.

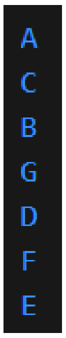
2. Experimental Result

(1) Expected Result :



```
Output:
A
C
B
G
D
F
E
```

(2) Actual Result :



```
A
C
B
G
D
F
E
```

3. Result Analysis

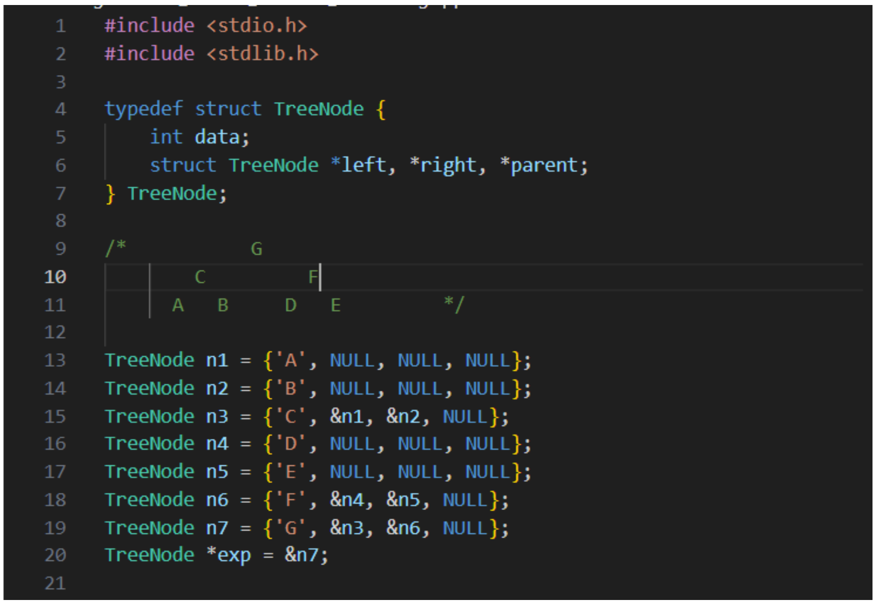
The tree_successor function is used in the do-while loop in the main function to visit all nodes in the tree. The time complexity of the tree_successor function varies depending on the given node p. If the node p has a right subtree, it needs to find the leftmost node in that subtree, and this operation takes time proportional to the height of the right subtree. In general, in a binary tree, this operation typically takes O(log n) time.

However, if the node p does not have a right subtree and needs to move to its parent, the time it takes to move to the parent is proportional to the depth of the tree. In the worst case, this can take O(n) time, as it might involve traversing the entire tree.

Thus, the time complexity of the tree_successor function depends on the structure of the given tree. In the worst case, it can be O(n), while in the best case, it's O(log n) due to the properties of balanced binary trees.

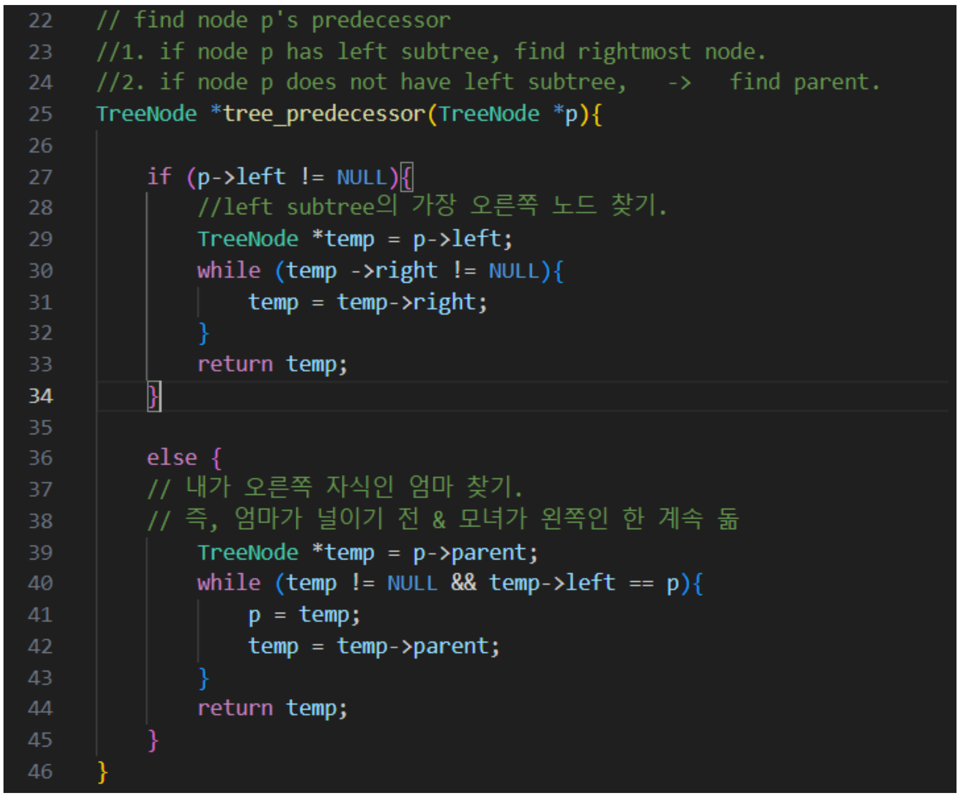
II. Problem 3

1. Code Explanation



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

typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right, *parent;
} TreeNode;

/*          G
        C         F
      A   B     D   E         */

TreeNode n1 = {'A', NULL, NULL, NULL};
TreeNode n2 = {'B', NULL, NULL, NULL};
TreeNode n3 = {'C', &n1, &n2, NULL};
TreeNode n4 = {'D', NULL, NULL, NULL};
TreeNode n5 = {'E', NULL, NULL, NULL};
TreeNode n6 = {'F', &n4, &n5, NULL};
TreeNode n7 = {'G', &n3, &n6, NULL};
TreeNode *exp = &n7;

```

Same as Problem 1.

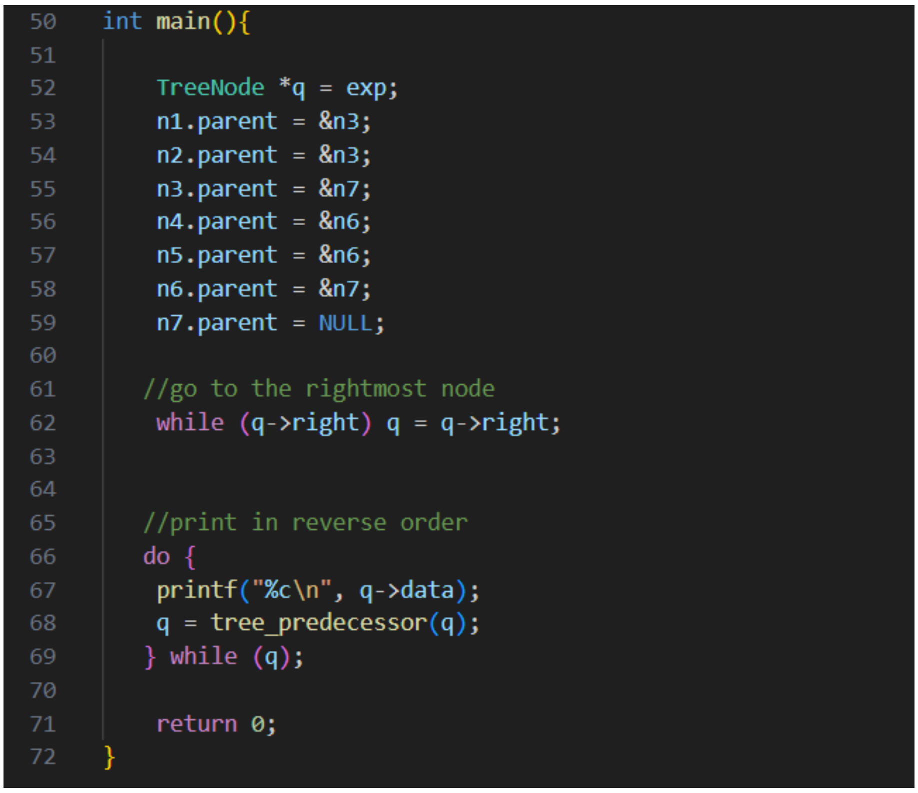


```c
// find node p's predecessor
//1. if node p has left subtree, find rightmost node.
//2. if node p does not have left subtree,   ->   find parent.
TreeNode *tree_predecessor(TreeNode *p){

    if (p->left != NULL){
        //left subtree의 가장 오른쪽 노드 찾기.
        TreeNode *temp = p->left;
        while (temp ->right != NULL){
            temp = temp->right;
        }
        return temp;
    }

    else {
    // 내가 오른쪽 자식인 엄마 찾기.
    // 즉, 엄마가 널이기 전 & 모녀가 왼쪽인 한 계속 돎
        TreeNode *temp = p->parent;
        while (temp != NULL && temp->left == p){
            p = temp;
            temp = temp->parent;
        }
        return temp;
    }
}
```

The "tree_predecessor" function finds and returns the predecessor of the given TreeNode "p." Here are the two scenarios for determining the predecessor:

1. If the given "p" has a left subtree, its predecessor is the rightmost node in that left subtree. To find this node, we declare a temporary variable "temp" and initialize it with the root node value of "p's" left subtree. Then, we continue moving "temp" to the right child's position until there are no more nodes to the right (i.e., until "temp" represents the rightmost node). When the loop exits, the value returned by "temp" becomes the predecessor of "p."

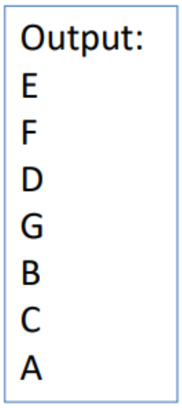
2. If the given "p" does not have a left subtree, its predecessor is found by moving up the tree to the parent nodes. In this case, the predecessor is the parent node where the relationship between the current node and its parent is a rightward connection. To implement this, we initialize a temporary variable "temp" with "p's" parent node. While "temp" is not NULL and its relationship with its child is on the left side, we continue moving up the tree by assigning "temp" to its parent. When the condition is no longer met, the "temp" variable, at that point, represents the first node that was the predecessor of "p."


```c
int main(){

    TreeNode *q = exp;
    n1.parent = &n3;
    n2.parent = &n3;
    n3.parent = &n7;
    n4.parent = &n6;
    n5.parent = &n6;
    n6.parent = &n7;
    n7.parent = NULL;

   //go to the rightmost node
    while (q->right) q = q->right;


   //print in reverse order
   do {
    printf("%c\n", q->data);
    q = tree_predecessor(q);
   } while (q);

    return 0;
}
```

1. Unlike in problem 2, in this case, we start from the rightmost node of the tree and systematically traverse the nodes in reverse order by moving from one node to its predecessor using the "tree_predecessor" function.

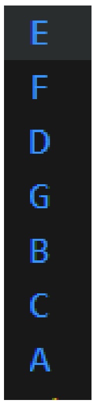
2. Experimental Result

(1) Expected Result :



```
Output:
E
F
D
G
B
C
A
```

(2) Actual Result :



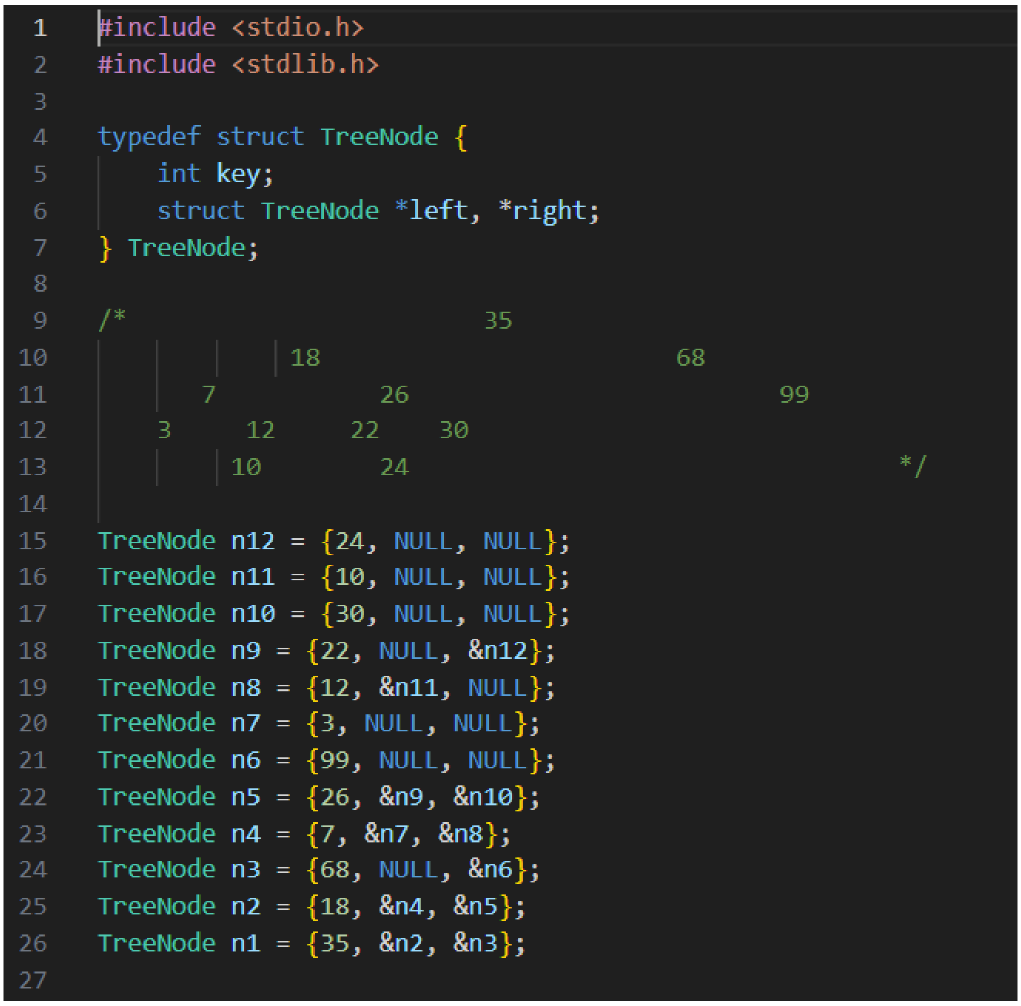
```
E
F
D
G
B
C
A
```

3. Result Analysis

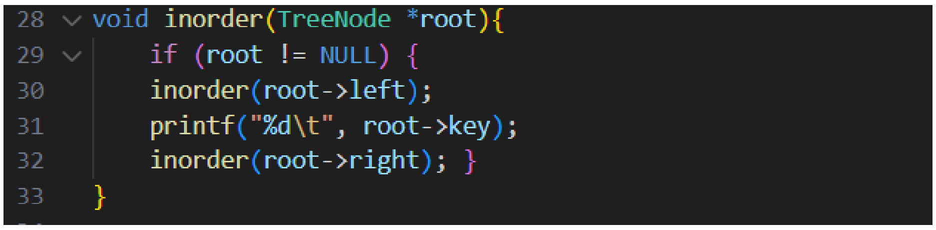
The time complexity, when using the "tree_predecessor" function, remains the same, with a maximum of O(n) and a minimum of O(log n) required.

III. Problem 4

1. Code Explanation

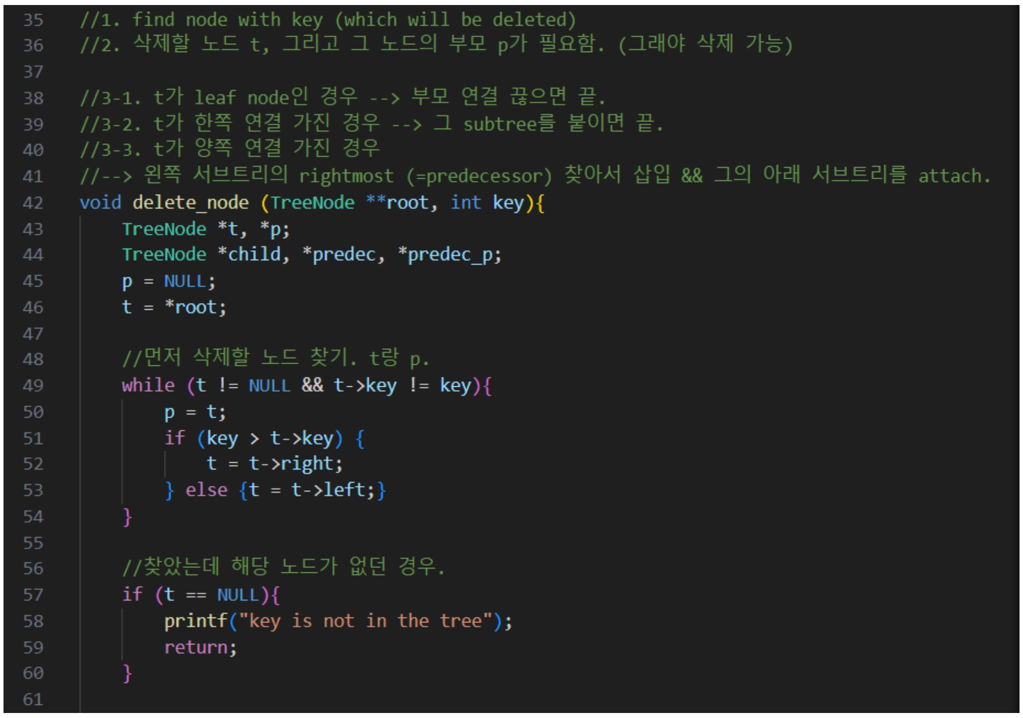


```c
#include <stdio.h>
#include <stdlib.h>

typedef struct TreeNode {
    int key;
    struct TreeNode *left, *right;
} TreeNode;

/*                        35
             18                        68
       7           26                        99
    3     12     22     30
         10        24                                  */

TreeNode n12 = {24, NULL, NULL};
TreeNode n11 = {10, NULL, NULL};
TreeNode n10 = {30, NULL, NULL};
TreeNode n9 = {22, NULL, &n12};
TreeNode n8 = {12, &n11, NULL};
TreeNode n7 = {3, NULL, NULL};
TreeNode n6 = {99, NULL, NULL};
TreeNode n5 = {26, &n9, &n10};
TreeNode n4 = {7, &n7, &n8};
TreeNode n3 = {68, NULL, &n6};
TreeNode n2 = {18, &n4, &n5};
TreeNode n1 = {35, &n2, &n3};

```



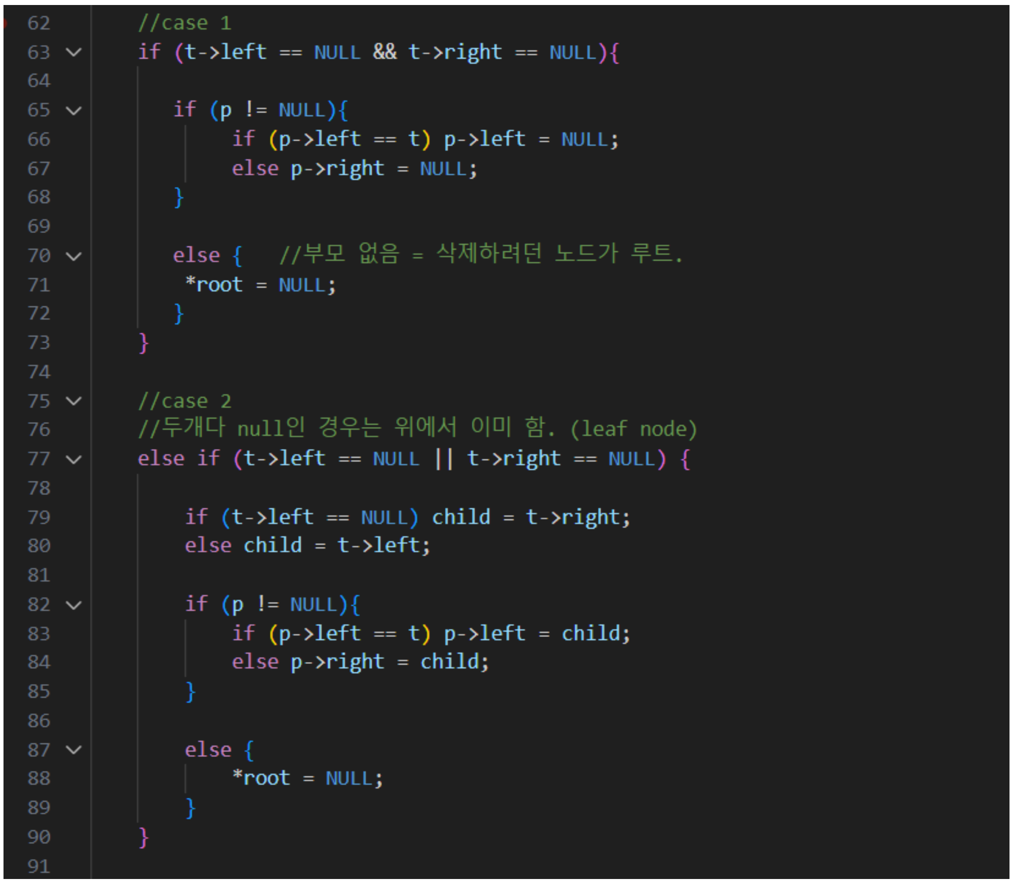
```c
void inorder(TreeNode *root){
    if (root != NULL) {
    inorder(root->left);
    printf("%d\t", root->key);
    inorder(root->right); }
}
```

1. This function, inorder, performs an inorder traversal of a binary tree. It takes the root node of the tree as an argument and prints the nodes in ascending order (sorted) by recursively visiting the left subtree, printing the current node's key, and then recursively visiting the right subtree.



```c
//1. find node with key (which will be deleted)
//2. 삭제할 노드 t, 그리고 그 노드의 부모 p가 필요함. (그래야 삭제 가능)

//3-1. t가 leaf node인 경우 --> 부모 연결 끊으면 끝.
//3-2. t가 한쪽 연결 가진 경우 --> 그 subtree를 붙이면 끝.
//3-3. t가 양쪽 연결 가진 경우
//--> 왼쪽 서브트리의 rightmost (=predecessor) 찾아서 삽입 && 그의 아래 서브트리를 attach.
void delete_node (TreeNode **root, int key){
    TreeNode *t, *p;
    TreeNode *child, *predec, *predec_p;
    p = NULL;
    t = *root;

    //먼저 삭제할 노드 찾기. t랑 p.
    while (t != NULL && t->key != key){
        p = t;
        if (key > t->key) {
            t = t->right;
        } else {t = t->left;}
    }

    //찾았는데 해당 노드가 없던 경우.
    if (t == NULL){
        printf("key is not in the tree");
        return;
    }

```

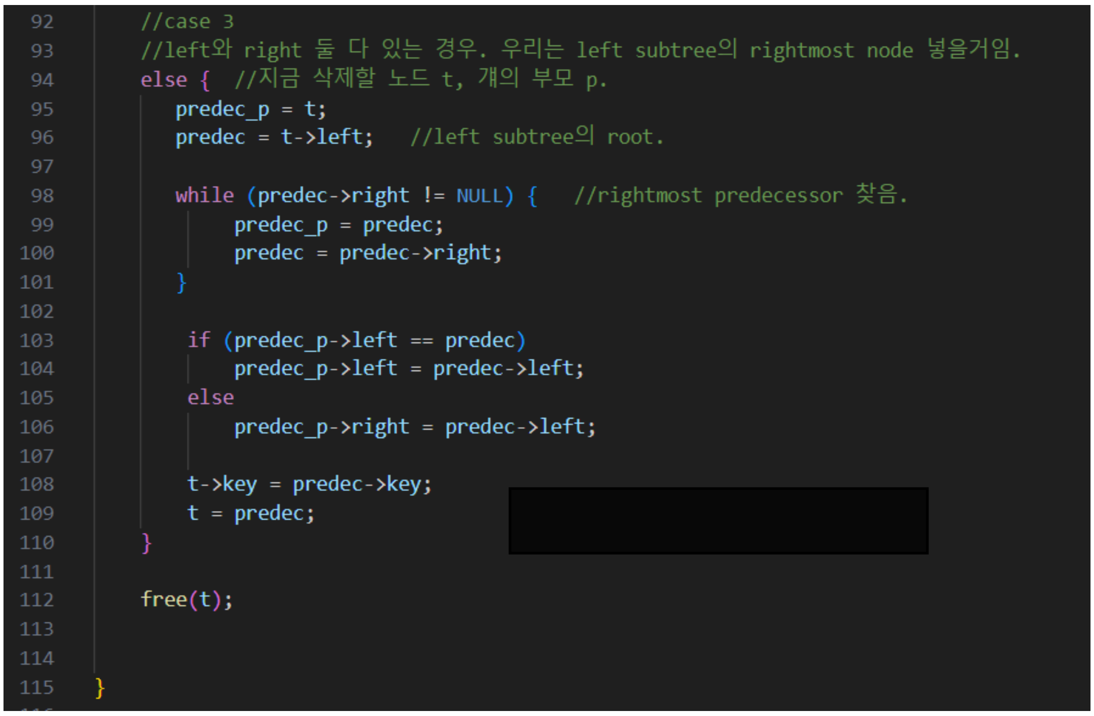
This function, delete_node, is responsible for deleting a node with a given key from a binary search tree. It first searches for the node with the specified key (key) and maintains the parent node (p). It then handles three different cases of deletion:



```c
    //case 1
    if (t->left == NULL && t->right == NULL){

        if (p != NULL){
            if (p->left == t) p->left = NULL;
            else p->right = NULL;
        }

        else {   //부모 없음 = 삭제하려던 노드가 루트.
         *root = NULL;
        }
    }

    //case 2
    //두개다 null인 경우는 위에서 이미 함. (leaf node)
    else if (t->left == NULL || t->right == NULL) {

        if (t->left == NULL) child = t->right;
        else child = t->left;

        if (p != NULL){
            if (p->left == t) p->left = child;
            else p->right = child;
        }

        else {
            *root = NULL;
        }
    }

```

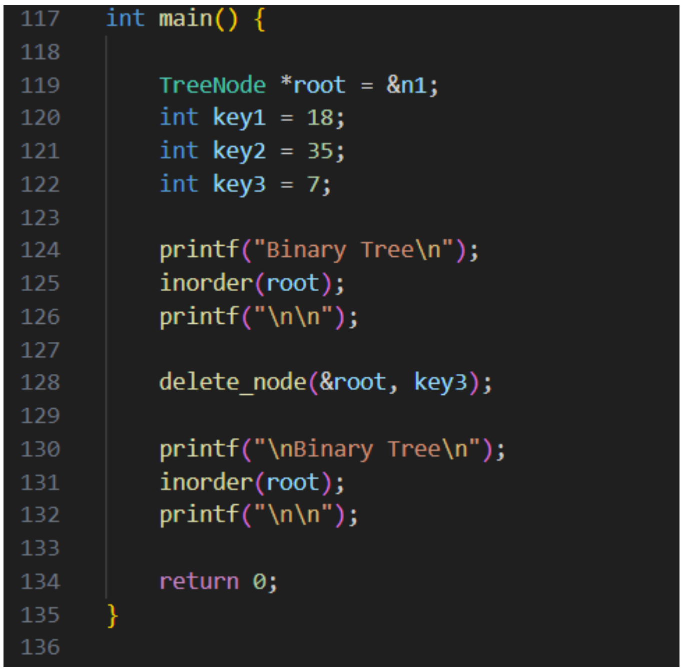
1. Case 1: If the node to be deleted (t) is a leaf node (has no children), it is simply removed by updating the parent's pointer.

2. Case 2: If the node to be deleted has only one child, it is removed by linking the parent to the single child node.



```c
    //case 3
    //left와 right 둘 다 있는 경우. 우리는 left subtree의 rightmost node 넣을거임.
    else {  //지금 삭제할 노드 t, 걔의 부모 p.
        predec_p = t;
        predec = t->left;   //left subtree의 root.

        while (predec->right != NULL) {   //rightmost predecessor 찾음.
            predec_p = predec;
            predec = predec->right;
        }

         if (predec_p->left == predec)
             predec_p->left = predec->left;
         else
             predec_p->right = predec->left;

         t->key = predec->key;
         t = predec;              [redacted]
    }

    free(t);


}
```

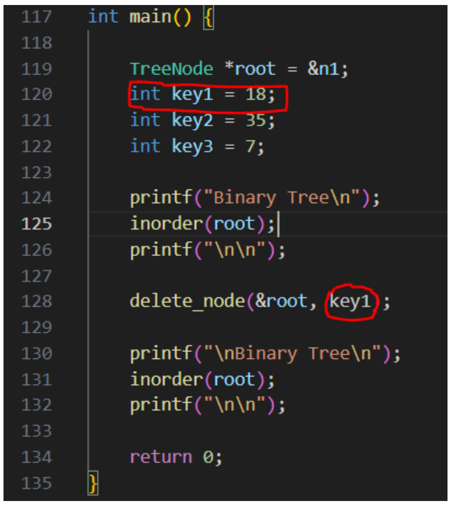
3. Case 3: If the node to be deleted has two children, it is more complex. The code finds the predecessor node (predec) which is the maximum value node in the left subtree of the node to be deleted. The key of the predecessor node is copied to the node to be deleted, and then the predecessor node is deleted.



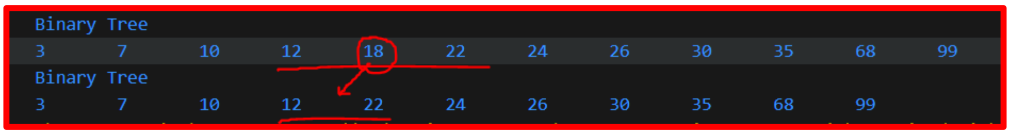
```c
int main() {

    TreeNode *root = &n1;
    int key1 = 18;
    int key2 = 35;
    int key3 = 7;

    printf("Binary Tree\n");
    inorder(root);
    printf("\n\n");

    delete_node(&root, key3);

    printf("\nBinary Tree\n");
    inorder(root);
    printf("\n\n");

    return 0;
}
```

2. Experimental Result

(1) Deleting node with key = 18

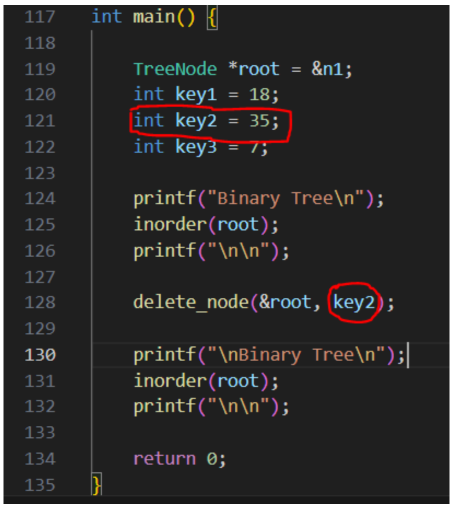


```
int main() {

    TreeNode *root = &n1;
    int key1 = 18;
    int key2 = 35;
    int key3 = 7;

    printf("Binary Tree\n");
    inorder(root);
    printf("\n\n");

    delete_node(&root, key1);

    printf("\nBinary Tree\n");
    inorder(root);
    printf("\n\n");

    return 0;
}
```

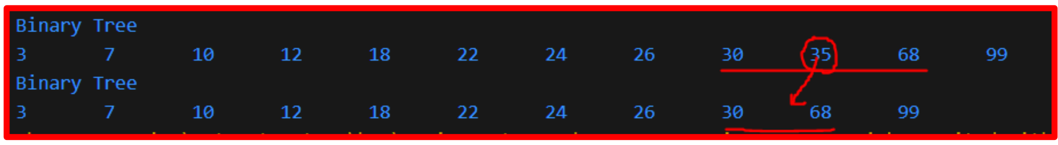


```
Binary Tree
3       7       10      12      18      22      24      26      30      35      68      99
Binary Tree
3       7       10      12      22      24      26      30      35      68      99
```

(2) Deleting node with key = 35

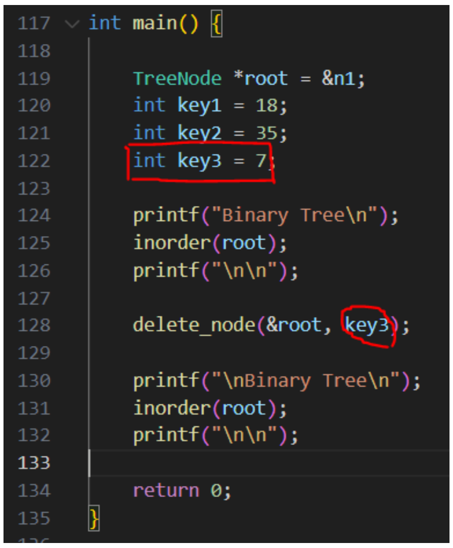


```
int main() {

    TreeNode *root = &n1;
    int key1 = 18;
    int key2 = 35;
    int key3 = 7;

    printf("Binary Tree\n");
    inorder(root);
    printf("\n\n");

    delete_node(&root, key2);

    printf("\nBinary Tree\n");
    inorder(root);
    printf("\n\n");

    return 0;
}
```

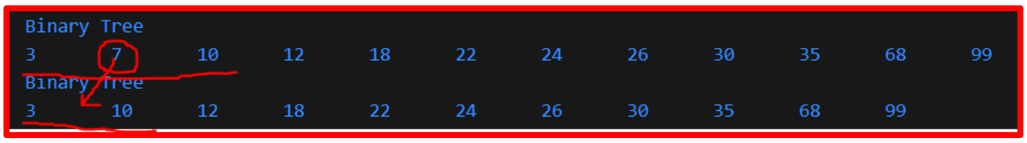


```
Binary Tree
3       7       10      12      18      22      24      26      30      35      68      99
Binary Tree
3       7       10      12      18      22      24      26      30      68      99
```

(3) Deleting node with key = 7



```
int main() {

    TreeNode *root = &n1;
    int key1 = 18;
    int key2 = 35;
    int key3 = 7;

    printf("Binary Tree\n");
    inorder(root);
    printf("\n\n");

    delete_node(&root, key3);

    printf("\nBinary Tree\n");
    inorder(root);
    printf("\n\n");

    return 0;
}
```



```
Binary Tree
3       7       10      12      18      22      24      26      30      35      68      99
Binary Tree
3       10      12      18      22      24      26      30      35      68      99
```

3. Result Analysis

Delete_node function, which deletes the delivered parameter node p in the tree, first searches the position of node p.

When searching for the node to delete, in the worst-case, the function traverses the binary search tree from the root to the node to be deleted. The time complexity of this is O(h), where h is the height of the binary search tree.

When handling three different deletion cases, the worst time complexity would take O(h) time, in the case3 with traversing the left subtree of the node to be deleted.

Thus, in the best case, the height of the tree is O(log n), making the time complexity O(log n). In the worst case, the height of the tree can be O(n), leading to a worst-case time complexity of O(n).