Assignment 5

Course: Data Structures
Course id:14461002
Student id: 202033762

Name: 장민호

Major : 설비소방공학과

Submission Date : 2022_04_02

HW 5-1

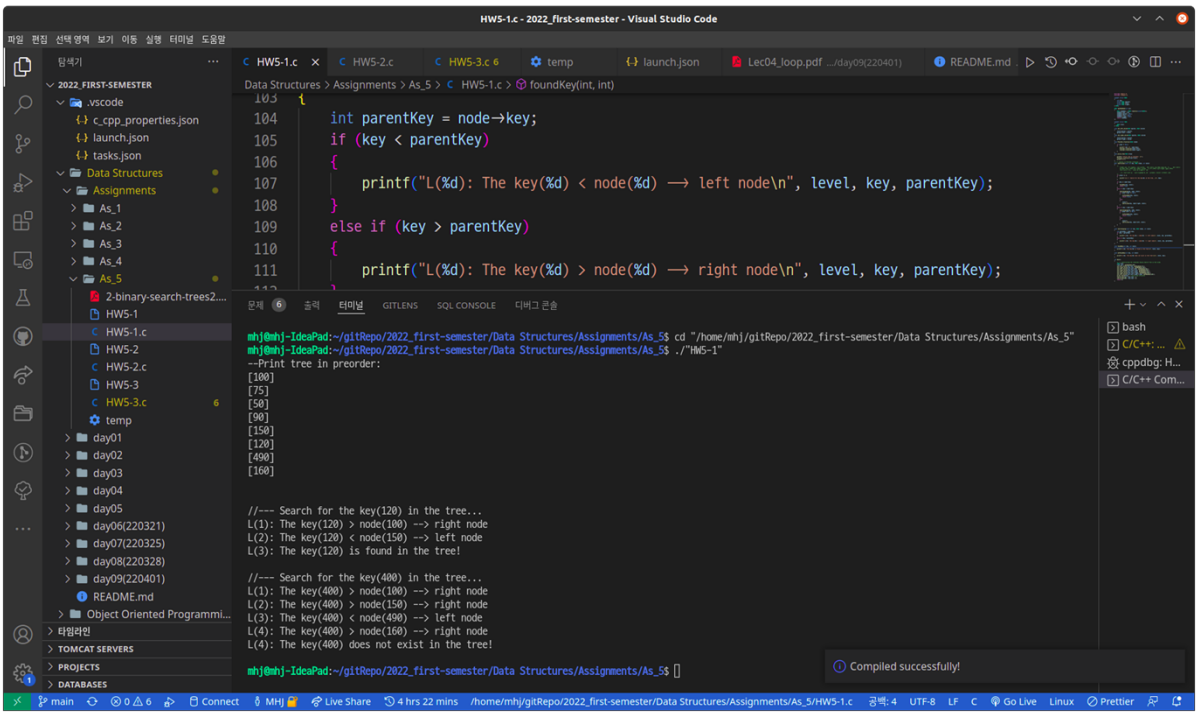
- Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <stdbool.h>
typedef struct NODE
{
   int key;
   struct NODE *parent;
   struct NODE *left;
   struct NODE *right;
} NODE;
NODE *getNewNode(int val)
{
   NODE *newNode = (NODE *)malloc(sizeof(NODE));
   newNode->key = val;
   newNode->parent = NULL;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
};
typedef struct TREE
{
   NODE *root;
} TREE;
void set_left_child(NODE *parent, NODE *child)
{
   child->parent = parent;
   parent->left = child;
}
void set_right_child(NODE *parent, NODE *child)
{
   child->parent = parent;
   parent->right = child;
}
void Preorder_Traversal(NODE *node)
{
   if (node != NULL)
   {
       printf("[%d] \n", node->key);
       Preorder_Traversal(node->left);
       Preorder_Traversal(node->right);
   }
}
void print_tree(TREE *tree)
{
   printf("--Print tree in preorder: \n");
   Preorder_Traversal(tree->root);
   printf("\n");
```

```
}
// --- Similar to Preorder Traversal !
bool search_key(const int key, NODE *node, int level)
{
   /*
    Tree의 root부터 시작하여, root 키 값과 변수로 들어온 key 값을 비교한다. 이때, 변수로 들어온 key 값이 같을 경우는 바로 함수를 끝낸다. 만약 클 경우는 root->right, 작을 경우는 root->left로 해주고 level 을 1증가시켜준다.
    만약, root->left 혹은 root->right가 존재하지 않는다면, false를 리턴하면 된다.
   */
   if (level == 1)
   {
       printf("\n//--- Search for the key(%d) in the tree...\n", key);
   }
   if (key == node->key)
   {
       foundKey(key, level);
       return true;
   }
   else if (key > node->key)
   {
       searchingLog(key, node, level);
       if (node->right == NULL)
       {
           notFoundKey(key, level);
           return false;
       }
       else
       {
           level++;
           search_key(key, node->right, level);
       }
   }
   else if (key < node->key)
   {
       searchingLog(key, node, level);
       if (node->left == NULL)
       {
           notFoundKey(key, level);
           return false;
       }
       else
       {
           level++;
           search_key(key, node->left, level);
       }
   }
}
void searchingLog(const int key, NODE *node, int level)
{
   int parentKey = node->key;
   if (key < parentKey)
   {
       printf("L(%d): The key(%d) < node(%d) --> left node\n", level, key,
parentKey);
   }
   else if (key > parentKey)
   {
```

```
        printf("L(%d): The key(%d) > node(%d) --> right node\n", level, key,
parentKey);
    }
}
void foundKey(int key, int level)
{
    printf("L(%d): The key(%d) is found in the tree!\n", level, key);
}
void notFoundKey(int key, int level)
{
    printf("L(%d): The key(%d) does not exist in the tree!\n\n", level, key);
}
int main()
{
    // --- Constructing the (Ordered) Binary Search Tree in the slide.
    TREE Tree;
    Tree.root = getNewNode(100);
    set_left_child(Tree.root, getNewNode(75));
    set_right_child(Tree.root, getNewNode(150));
    set_left_child(Tree.root->left, getNewNode(50));
    set_right_child(Tree.root->left, getNewNode(90));
    set_left_child(Tree.root->right, getNewNode(120));
    set_right_child(Tree.root->right, getNewNode(490));
    set_left_child(Tree.root->right->right, getNewNode(160));
    print_tree(&Tree); // print tree structure
    search_key(120, Tree.root, 1);
    search_key(400, Tree.root, 1);
    return 0;
}
```

- Screenshot



HW 5-2

- Source Code

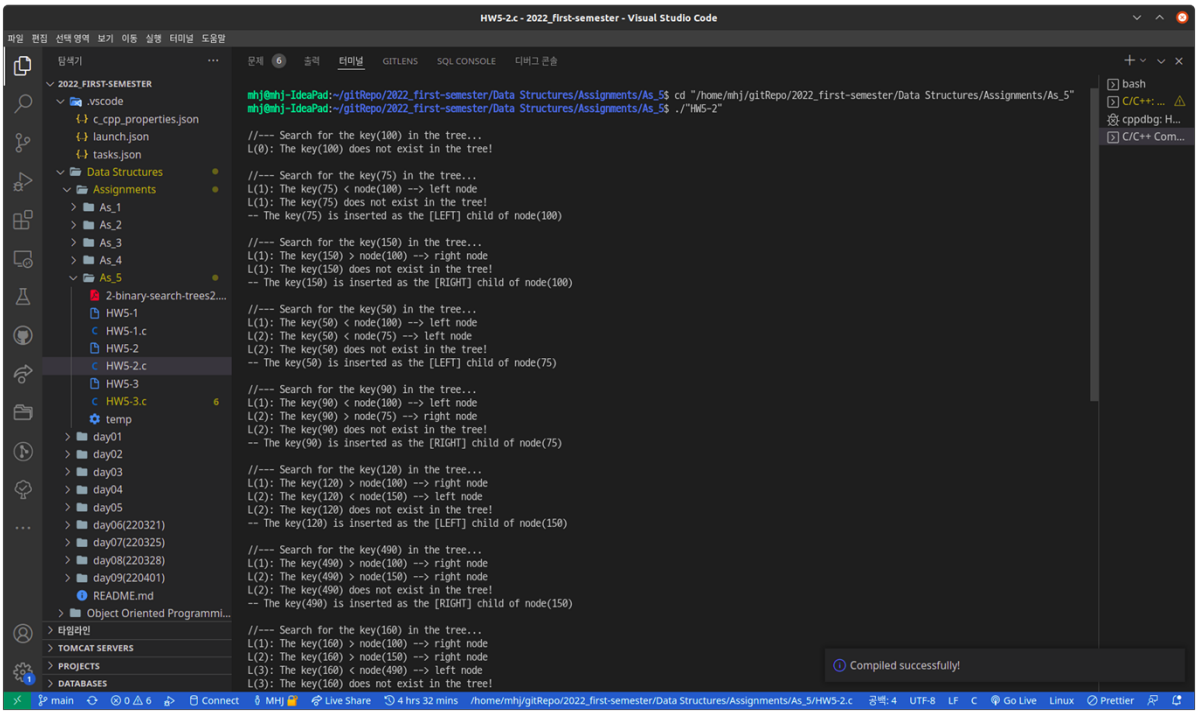
```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <stdbool.h>
typedef struct NODE
{
   int key;
   struct NODE *parent;
   struct NODE *left;
   struct NODE *right;
} NODE;
NODE *getNewNode(int val)
{
   NODE *newNode = (NODE *)malloc(sizeof(NODE));
   newNode->key = val;
   newNode->parent = NULL;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
};
typedef struct TREE
{
   NODE *root;
} TREE;
void tree_init(TREE *tree)
{
   tree->root = NULL;
}
void set_left_child(NODE *parent, NODE *child)
{
   child->parent = parent;
   parent->left = child;
}
void set_right_child(NODE *parent, NODE *child)
{
   child->parent = parent;
   parent->right = child;
}
void Preorder_Traversal(NODE *node)
{
   if (node != NULL)
   {
       printf("[%d] \n", node->key);
       Preorder_Traversal(node->left);
       Preorder_Traversal(node->right);
   }
}
```

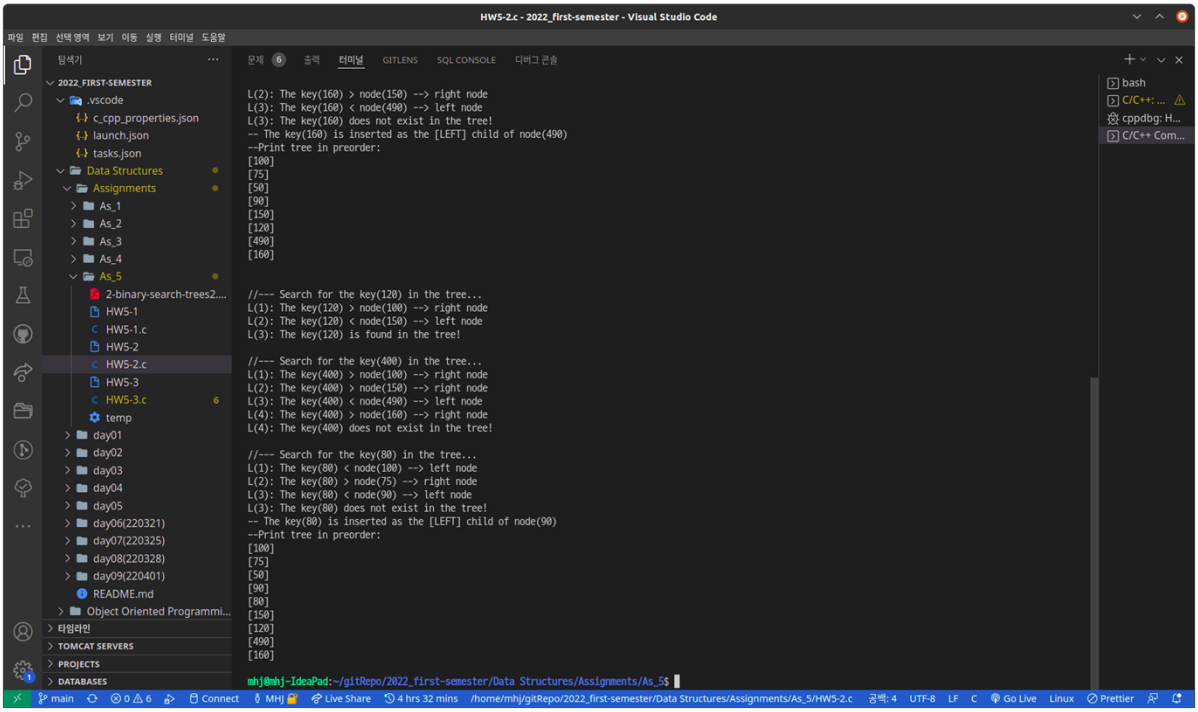
```
void print_tree(TREE *tree)
{
   printf("--Print tree in preorder: \n");
   Preorder_Traversal(tree->root);
   printf("\n");
}
// 아래 search_key 에서 parentNode를 저장하기 위해, 전역변수로 만들었다.
NODE *parentNode;
// --- Similar to Preorder Traversal !
bool search_key(const int key, NODE *node, int level)
{
   // 예외상황: 처음 search를 실행할 경우
   // search를 실행한다는 건, root가 NULL은 아니라는 뜻이다.
   if (level == 1)
   {
       printf("\n//--- Search for the key(%d) in the tree...\n", key);
   }
   if (key == node->key)
   {
       foundKey(key, level);
       return true;
   }
   else if (key > node->key)
   {
       searchingLog(key, node, level);
       if (node->right == NULL)
       {
           parentNode = node;
           notFoundKey(key, level);
           return false;
       }
       else
       {
           level++;
           search_key(key, node->right, level);
       }
   }
   else if (key < node->key)
   {
       searchingLog(key, node, level);
       if (node->left == NULL)
       {
           parentNode = node;
           notFoundKey(key, level);
           return false;
       }
       else
       {
           level++;
           search_key(key, node->left, level);
       }
   }
}
NODE *find_insert_loc(const int key, NODE *node)
{
   /*
    숫자를 삽입할 위치를 찾으면 된다. 삽입이 가능하다는 것은 그 숫자를 찾지 못했다는 의미이다. 즉,
search_key를 통해 그 값이 false를 리턴했을 경우를 의미한다. false를 리턴할 당시, search_key에서
```

```
argument값으로 들어간 node는 지금 당장 삽입할 노드의 부모가 된다. 따라서 그 argument 값을 저장해놓
고 find_insert_loc 에서 리턴해주면 된다.
   */
   return parentNode;
}
void insert_key(const int key, TREE *tree)
{
   // 만약 insert를 하려는데 tree->root 값이 NULL 이라면? search를 들어갈 이유가 없다.
   // root 노드 한정해서 하면 된다.
   if (tree->root == NULL)
   {
       tree->root = getNewNode(key);
       printf("\n//--- Search for the key(%d) in the tree...\n", key);
       notFoundKey(key, 0);
       return;
   }
   // 이미 여기에서 search를 수행했다. 따라서 아래에서 한번 더 수행할 필요가 없다.
   if (search_key(key, tree->root, 1))
   {
       printf("(Insert Failed): The key(%d) already exists..\n", key);
       return;
   }
   // loc는 현재 부모가 될 노드이다.
   NODE *loc = find_insert_loc(key, tree->root);
   // 부모노드와 자식노드간 값을 비교한다.
   char direction[10];
   if (loc->key > key)
   {
       strcpy(direction, "LEFT");
       set_left_child(loc, getNewNode(key));
   }
   else
   {
       strcpy(direction, "RIGHT");
       set_right_child(loc, getNewNode(key));
   }
   printf("-- The key(%d) is inserted as the [%s] child of node(%d)\n", key,
direction, loc->key);
}
// 탐색하면서 로그를 남기는 함수
void searchingLog(const int key, NODE *node, int level)
{
   int parentKey = node->key;
   if (key < parentKey)
   {
       printf("L(%d): The key(%d) < node(%d) --> left node\n", level, key,
parentKey);
   }
   else if (key > parentKey)
   {
       printf("L(%d): The key(%d) > node(%d) --> right node\n", level, key,
parentKey);
   }
}
void foundKey(int key, int level)
{
   printf("L(%d): The key(%d) is found in the tree!\n", level, key);
}
```

```c
void notFoundKey(int key, int level)
{
   printf("L(%d): The key(%d) does not exist in the tree!\n", level, key);
}
int main()
{
   // --- Constructing the (Ordered) Binary Search Tree in the slide.
   TREE Tree;
   // 초기 Tree root를 NULL로 만들어주는 작업
   tree_init(&Tree);
   insert_key(100, &Tree);
   insert_key(75, &Tree);
   insert_key(150, &Tree);
   insert_key(50, &Tree);
   insert_key(90, &Tree);
   insert_key(120, &Tree);
   insert_key(490, &Tree);
   insert_key(160, &Tree);
   print_tree(&Tree); // print tree structure
   // you should get the same results with the previous version
   search_key(120, Tree.root, 1);
   search_key(400, Tree.root, 1);
   // check the insertion result.
   insert_key(80, &Tree);
   print_tree(&Tree);
   return 0;
}
```

- ScreenShot





HW 5-3

- Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <stdbool.h>
typedef struct NODE
{
   int key;
   struct NODE *parent;
   struct NODE *left;
   struct NODE *right;
} NODE;
NODE *getNewNode(int val)
{
   NODE *newNode = (NODE *)malloc(sizeof(NODE));
   newNode->key = val;
   newNode->parent = NULL;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
};
typedef struct TREE
{
   NODE *root;
} TREE;
void tree_init(TREE *tree)
{
   tree->root = NULL;
```

```
}
void set_left_child(NODE *parent, NODE *child)
{
   child->parent = parent;
   parent->left = child;
}
void set_right_child(NODE *parent, NODE *child)
{
   child->parent = parent;
   parent->right = child;
}
void Preorder_Traversal(NODE *node)
{
   if (node != NULL)
   {
       printf("[%d] \n", node->key);
       Preorder_Traversal(node->left);
       Preorder_Traversal(node->right);
   }
}
void print_tree(TREE *tree)
{
   printf("--Print tree in preorder: \n");
   Preorder_Traversal(tree->root);
   printf("\n");
}
// 아래 search_key 에서 parentNode를 저장하기 위해, 전역변수로 만들었다.
NODE *parentNode;
// --- Similar to Preorder Traversal !
bool search_key(const int key, NODE *node, int level)
{
   // 예외상황: 처음 search를 실행할 경우
   // search를 실행한다는 건, root가 NULL은 아니라는 뜻이다.
   if (level == 1)
   {
       printf("\n//--- Search for the key(%d) in the tree...\n", key);
   }
   if (key == node->key)
   {
       foundKey(key, level);
       return true;
   }
   else if (key > node->key)
   {
       searchingLog(key, node, level);
       if (node->right == NULL)
       {
           parentNode = node;
           notFoundKey(key, level);
           return false;
       }
       else
       {
           level++;
           search_key(key, node->right, level);
       }
   }
   else if (key < node->key)
   {
```

```
        searchingLog(key, node, level);
        if (node->left == NULL)
        {
            parentNode = node;
            notFoundKey(key, level);
            return false;
        }
        else
        {
            level++;
            search_key(key, node->left, level);
        }
    }
}
NODE *find_insert_loc(const int key, NODE *node)
{
    /*
     숫자를 삽입할 위치를 찾으면 된다. 삽입이 가능하다는 것은 그 숫자를 찾지 못했다는 의미이다. 즉,
search_key를 통해 그 값이 false를 리턴했을 경우를 의미한다. false를 리턴할 당시, search_key에서
argument값으로 들어간 node는 지금 당장 삽입할 노드의 부모가 된다. 따라서 그 argument 값을 저장해놓
고 find_insert_loc 에서 리턴해주면 된다.
    */
    return parentNode;
}
void insert_key(const int key, TREE *tree)
{
    // 만약 insert를 하려는데 tree->root 값이 NULL 이라면? search를 들어갈 이유가 없다.
    // root 노드 한정해서 하면 된다.
    if (tree->root == NULL)
    {
        tree->root = getNewNode(key);
        printf("\n//--- Search for the key(%d) in the tree...\n", key);
        notFoundKey(key, 0);
        return;
    }
    // 이미 여기에서 search를 수행했다. 따라서 아래에서 한번 더 수행할 필요가 없다.
    if (search_key(key, tree->root, 1))
    {
        printf("(Insert Failed): The key(%d) already exists..\n", key);
        return;
    }
    // loc는 현재 부모가 될 노드이다.
    NODE *loc = find_insert_loc(key, tree->root);
    // 부모노드와 자식노드간 값을 비교한다.
    char direction[10];
    if (loc->key > key)
    {
        strcpy(direction, "LEFT");
        set_left_child(loc, getNewNode(key));
    }
    else
    {
        strcpy(direction, "RIGHT");
        set_right_child(loc, getNewNode(key));
    }
    printf("-- The key(%d) is inserted as the [%s] child of node(%d)\n", key,
direction, loc->key);
}
```

```
// 탐색하면서 로그를 남기는 함수
void searchingLog(const int key, NODE *node, int level)
{
   int parentKey = node->key;
   if (key < parentKey)
   {
       printf("L(%d): The key(%d) < node(%d) --> left node\n", level, key,
parentKey);
   }
   else if (key > parentKey)
   {
       printf("L(%d): The key(%d) > node(%d) --> right node\n", level, key,
parentKey);
   }
}
void foundKey(int key, int level)
{
   printf("L(%d): The key(%d) is found in the tree!\n", level, key);
}
void notFoundKey(int key, int level)
{
   printf("L(%d): The key(%d) does not exist in the tree!\n", level, key);
}
// You should call this function only if search_key() == true.
/*
  삭제할 노드가 있다는 것은 결국 search_key()가 true값을 리턴했다는 것이다. 즉, 노드가 NULL일 경우는
없을 것이다.
*/
NODE *find_delete_node(const int key, NODE *node)
{
   // node == NULL Never Happens!
   if (key == node->key)
   {
       return node;
   }
   else if (key > node->key)
   {
       return find_delete_node(key, node->right);
   }
   else
   {
       return find_delete_node(key, node->left);
   }
}
// 해당 노드의 자식 개수를 세는 함수.
int num_child(NODE *node)
{
   int count = 0;
   if (node->left)
   {
       count++;
   }
   if (node->right)
   {
       count++;
   }
   return count;
}
```

```
// 현재노드로부터 가장 왼쪽, 즉 해당 노드로부터 가장 작은 값을 가진 노드를 찾는 함수
NODE *find_smallest_node(NODE *node)
{
   if (node->left == NULL)
       return node;
   else
       return find_smallest_node(node->left);
}
void delete_key(const int key, TREE *tree)
{
   // search_key가 탐색에 실패했을 경우 find_delete_node 자체를 실행시키지 않는다.
   if (search_key(key, tree->root, 1) == false)
   {
       printf("(Delete Failed): The key(%d) does not exist..\n", key);
       return;
   }
   NODE *loc = find_delete_node(key, tree->root);
   int num = num_child(loc);
   NODE *one_child = NULL;
   /*
  num 값에 따라 취해야 할 알고리즘이 다르다.
  0 : 그냥 leaf node 이므로 바로 삭제시키면 된다.
  1 : 자식노드를 부모노드의 위치로 바꿔주면 된다.
  2 : 왼쪽 자식 노드중에서 가장 큰 값을 올리거나, 오른쪽 자식 노드중에서 가장 작은 값을 올리면 된다. 이때
우리는 두번째 방법을 사용할 것이다.
   */
   // 만약 삭제시키는 노드가 루트노드라면??? 부모노드가 존재하지 않을 것이다.
   // 삭제하려는 노드의 부모노드를 찾는다.
   NODE *deleteNodeParent = loc->parent;
   // 이때 switch_case 문 대신 if문을 사용한 이유는 현재 VSC에서 GCC로 C파일을 실행하고 있기 때문에,
중괄호로 묶어주지 않으면 case: 내부에서 변수가 선언되지 않기 때문이다. 참고 사이트는 다음과 같다:
https://dojang.io/mod/page/view.php?id=200
   if (num == 0)
   {
       // 삭제시키는 노드가 루트노드가 아닐 경우
       if (deleteNodeParent != NULL)
       {
           // 삭제시킬 노드의 부모가 가리키는 삭제시키는 노드 포인터값을 NULL로 바꿔주어야 한다.
           if (deleteNodeParent->left == loc)
           {
               deleteNodeParent->left = NULL;
           }
           else
           {
               deleteNodeParent->right = NULL;
           }
       }
       else
       {
           // 만약 num = 0인 루트노드가 제거되면 그냥 트리가 사라지는거랑 다름없다. 아래에서 free 해
주면 끝이다.
           tree->root = NULL;
       }
   }
   else if (num == 1)
   {
```

```
        // 삭제하려는 노드의 자식노드를, 삭제하려는 노드의 부모노드의 자식으로 연결해준다. 물론 그 자식의 부모노드도 바꿔주어야 한다.
        // 삭제시키는 노드가 루트노드가 아니라면
        if (deleteNodeParent != NULL)
        {
            // 1. 삭제하려는 노드의 자식노드를 찾는다.
            if (loc->left == NULL)
            {
                one_child = loc->right;
            }
            else
            {
                one_child = loc->left;
            }
            // 2. 만약 삭제하려는 노드가, 부모노드의 왼쪽에 있었을 경우 삭제하려는 노드의 자식노드도 똑같이 왼쪽에 위치시켜주면 된다. 반대의 경우도 마찬가지이다.
            if (deleteNodeParent->left == loc)
            {
                deleteNodeParent->left = one_child;
            }
            else
            {
                deleteNodeParent->right = one_child;
            }
            // 3. 바뀐 자식노드의 부모를 지정해준다.
            one_child->parent = deleteNodeParent;
        }
        else
        {
            // 만약 삭제시키는 노드가 루트노드라면? 그냥 삭제시키는 노드의 자식노드의 parent값을 없애버리면 된다. 또, tree의 root를 지정해주면 된다.
            one_child->parent = NULL;
            tree->root = one_child;
        }
    }
    else if (num == 2)
    {
        /*
        일단, num = 2라는 소리는 삭제시키는 노드의 자식노드가 두개이고, 따라서 반드시 loc->right에 위치한 노드가 존재한다. 그 노드를 기준으로 find_smallest_node() 를 수행해서 해당 노드를 찾는다. 그 노드를 삭제시키는 노드의 위치로 옮겨주어야 한다. 이때 할 일은 해당 노드와 관련된 다른 노드들이 가리키는 포인터값을 모두 수정해주어야 한다는 것이다.
        삭제시키는 노드를 A, 삭제시키는 노드에 들어갈 노드를 B라고 하자. 이때, A와 B에 관련된 노드들을 정리하자. B의 부모를 C, A의 왼쪽 자식을 D, A의 오른쪽 자식을 E, A의 부모를 F, B의 오른쪽 자식 노드를 G라고 하자.
        이것 역시, 삭제시키는 노드가 루트노드가 아닐 경우를 상정하고 한 것이다. 루트노드라면 로직을 바꿔주어야 한다.
            ==================
            0.
        loc = A이다.
        NODE* replaceNode = find_smallest_node(); B를 저장한다.
        NODE* replaceNodeParent = replaceNode->parent; C를 저장한다.
        NODE* deleteNodeLeftChild = loc->left; D를 저장한다.
        NODE* deleteNodeRightChild = loc->right; E를 저장한다.
        deleteNodeParent 는 F이다.
```

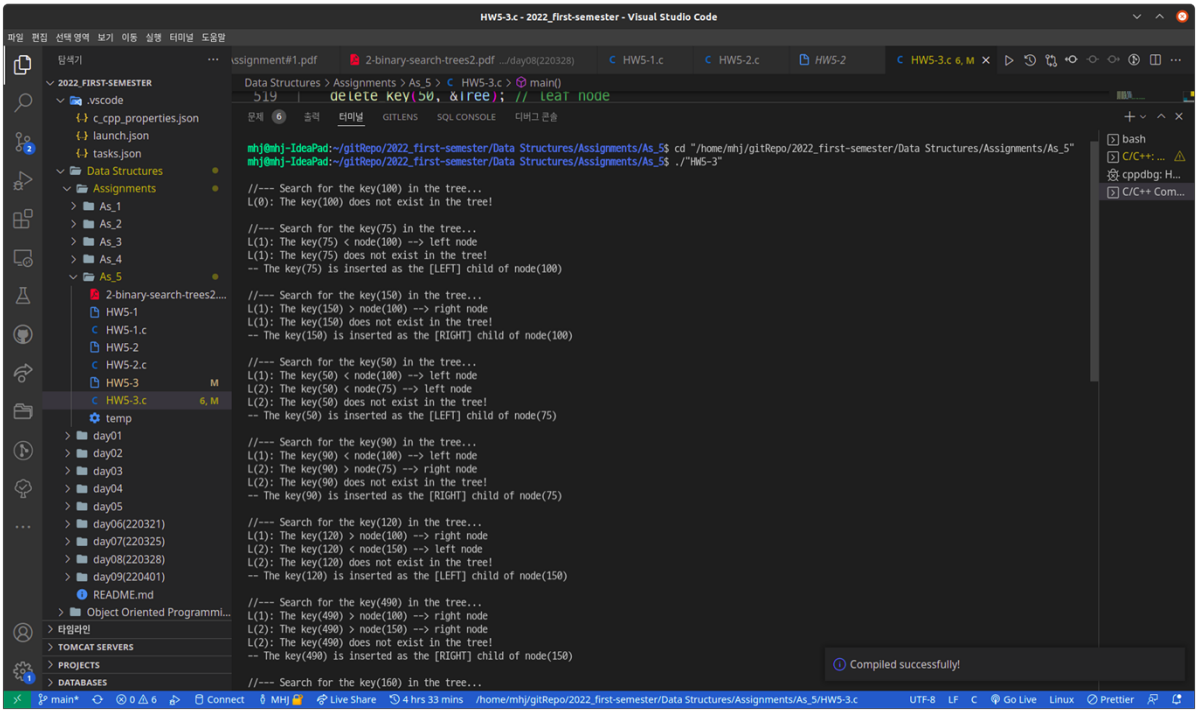
NODE* replaceNodeRightChild = replaceNode->right; G를 저장한다.
노드 사이 관계에 따라 로직이 변한다.
*****
#1. 서로 모두 다른 노드일 경우
*****
포인터 변경 순서는 다음과 같다.
1. B의 노드를 저장했으니, 이제 B는 위로 올려보낼 준비가 되었다.
이때 G가 존재한다면, G와 C간에 포인터 변경을 해주어야 한다.
만약 G가 NULL이라면
- B의 부모 C가 B를 가리키는 위치는 C->left이다. C->left = NULL; 해주자.
G가 NULL이 아니라면
- C->left = G로 해주고 G->parent = C 로 해주면 된다.
2. A와 관련된 노드들의 정보를 모두 저장했으므로, A는 더이상 필요없다. 이제 B를 A위치로 옮기자. 먼저 부모 F에 대해, loc의 자식위치를 알아내고 그 위치에다 B를 넣는다.
3. B의 parent를 F로 변경한다.
4. B의 자식을 각각 D, E로 변경한다.
5. D, E의 부모를 B로 변경한다.
*****
#2. C=E 일 경우
*****
#1과 마찬가지로 하면 된다.
*****
#3. A=C, B=E 일 경우
*****
더 간단해진다.
B를 그냥 한층 위로 끌고오면 된다. 이때, F의 자식노드의 위치를 찾고 그 위치에 B를 넣는다. 이후 B의 왼쪽 자식노드를 D로하고, B의 부모를 F로 한다.
*****
#분기점. A가 root 노드일 경우
*****
F의 자식을 찾고, B의 부모를 정하는 작업을 스킵한다.
자식을 모두 배치한 뒤 B를 tree->root로 만들어주면 된다.
*/
```
// loc = A
// NODE *replaceNode = find_smallest_node(&loc->right); // B
// NODE *replaceNodeParent = replaceNode->parent;       // C
// NODE *deleteNodeLeftChild = loc->left;               // D
// NODE *deleteNodeRightChild = loc->right;             // E
// // deleteNodeParent = F
// NODE *replaceNodeRightChild = replaceNode->right; // G
NODE *replaceNode = find_smallest_node(loc->right); // B
NODE *replaceNodeParent = replaceNode->parent;      // C
NODE *deleteNodeLeftChild = loc->left;              // D
NODE *deleteNodeRightChild = loc->right;            // E
// deleteNodeParent = F
NODE *replaceNodeRightChild = replaceNode->right; // G
// A=C, B=E인 경우
if (replaceNode->key == deleteNodeRightChild->key)
{
    // 만약 제거하는 노드가 루트노드가 아닐경우
    if (deleteNodeParent != NULL)
    {
        // B를 그냥 한층 위로 끌고오면 된다. 이때, F의 자식노드의 위치를 찾고 그 위치에 B를 넣는다. 이후 B의 왼쪽 자식노드를 D로하고, B의 부모를 F로 한다. D의 부모는 B로 한다.
        if (deleteNodeParent->left == loc)
```

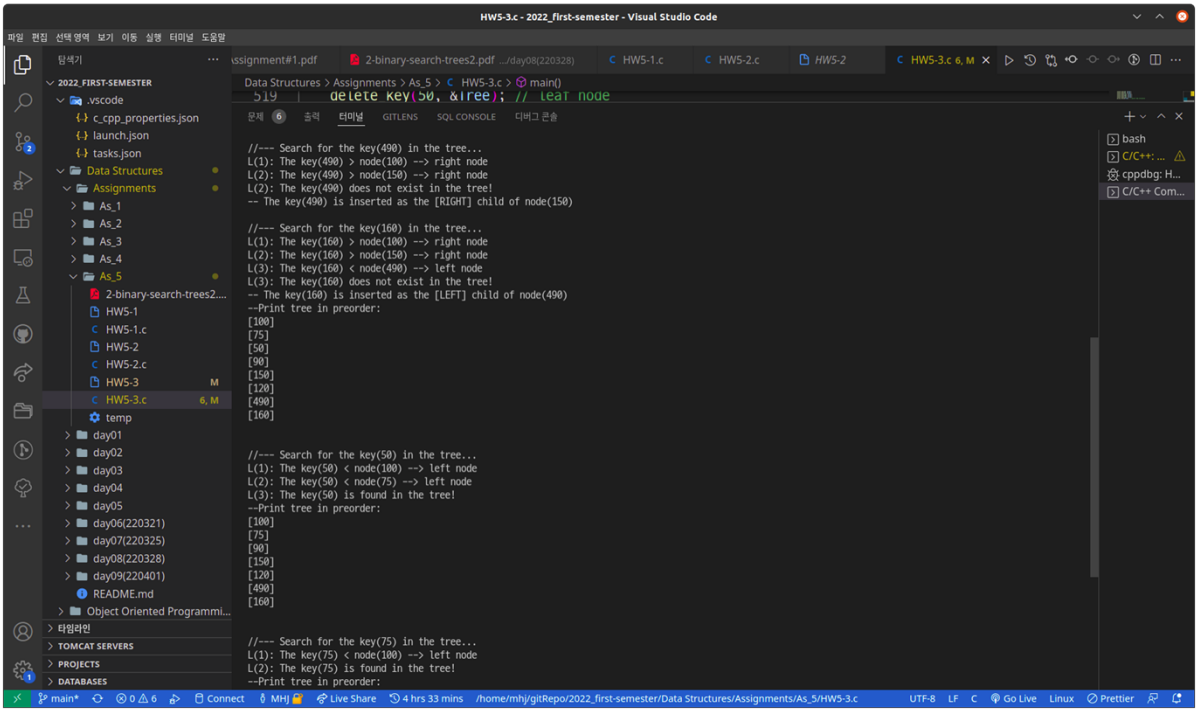
```
            {
                deleteNodeParent->left = replaceNode;
                replaceNode->parent = deleteNodeParent;
                replaceNode->left = deleteNodeLeftChild;
                deleteNodeLeftChild->parent = replaceNode;
            }
            else
            {
                deleteNodeParent->right = replaceNode;
                replaceNode->parent = deleteNodeParent;
                replaceNode->left = deleteNodeLeftChild;
                deleteNodeLeftChild->parent = replaceNode;
            }
        }
        else
        {
            // 제거하려는 노드가 루트노드
            replaceNode->left = deleteNodeLeftChild;
            replaceNode->parent = NULL;
            deleteNodeLeftChild->parent = replaceNode;
            tree->root = replaceNode;
        }
    }
    else
    {
        if (deleteNodeParent != NULL)
        {
            /*
      1. B의 노드를 저장했으니, 이제 B는 위로 올려보낼 준비가 되었다.
      이때 G가 존재한다면, G와 C간에 포인터 변경을 해주어야 한다.
      만약 G가 NULL이라면
        - B의 부모 C가 B를 가리키는 위치는 C->left이다. C->left = NULL; 해주자.
      G가 NULL이 아니라면
        - C->left = G로 해주고 G->parent = C 로 해주면 된다.
      2. A와 관련된 노드들의 정보를 모두 저장했으므로, A는 더이상 필요없다. 이제 B를 A위치로 옮기자.
먼저 부모 F에 대해, loc의 자식위치를 알아내고 그 위치에다 B를 넣는다.
      3. B의 parent를 F로 변경한다.
      4. B의 자식을 각각 D, E로 변경한다.
      5. D, E의 부모를 B로 변경한다.
            */
            // 1번
            if (replaceNodeRightChild == NULL)
            {
                replaceNodeParent->left = NULL;
            }
            else
            {
                replaceNodeParent->left = replaceNodeRightChild;
                replaceNodeRightChild->parent = replaceNodeParent;
            }
            // 2번
            if (deleteNodeParent->left == loc)
            {
                deleteNodeParent->left = replaceNode;
            }
            else
            {
```

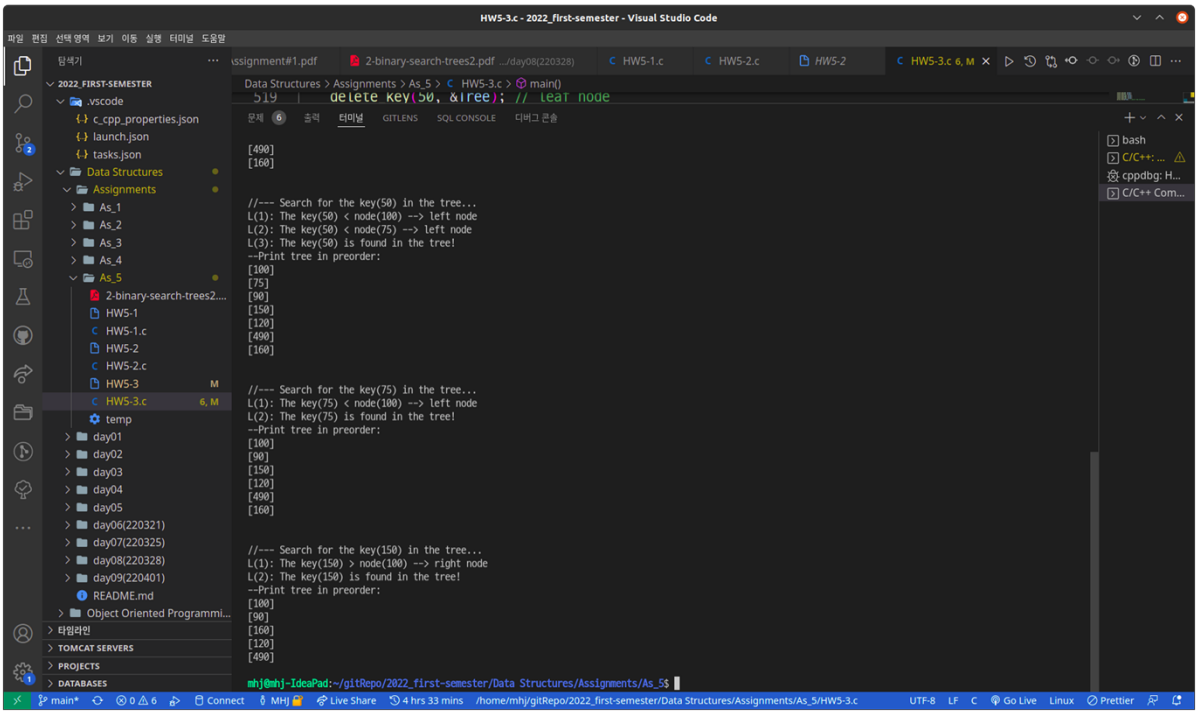
```
                    deleteNodeParent->right = replaceNode;
                }
                replaceNode->parent = deleteNodeParent;
                replaceNode->left = deleteNodeLeftChild;
                replaceNode->right = deleteNodeRightChild;
                deleteNodeLeftChild->parent = replaceNode;
                deleteNodeRightChild->parent = replaceNode;
            }
            else
            {
                // 제거하려는 노드가 루트노드
                if (replaceNodeRightChild == NULL)
                {
                    replaceNodeParent->left = NULL;
                }
                else
                {
                    replaceNodeParent->left = replaceNodeRightChild;
                    replaceNodeRightChild->parent = replaceNodeParent;
                }
                replaceNode->parent = NULL;
                replaceNode->left = deleteNodeLeftChild;
                replaceNode->right = deleteNodeRightChild;
                deleteNodeLeftChild->parent = replaceNode;
                deleteNodeRightChild->parent = replaceNode;
                tree->root = replaceNode;
            }
        }
    }
    free(loc);
}
int main()
{
    // --- Constructing the (Ordered) Binary Search Tree in the slide.
    TREE Tree;
    // 초기 Tree root를 NULL로 만들어주는 작업
    tree_init(&Tree);
    insert_key(100, &Tree);
    insert_key(75, &Tree);
    insert_key(150, &Tree);
    insert_key(50, &Tree);
    insert_key(90, &Tree);
    insert_key(120, &Tree);
    insert_key(490, &Tree);
    insert_key(160, &Tree);
    print_tree(&Tree); // print tree structure
    // check the deletion results.
    delete_key(50, &Tree); // leaf node
    print_tree(&Tree);
    delete_key(75, &Tree); // interior node with one child node
    print_tree(&Tree);
    delete_key(150, &Tree); // interior node with two child nodes
    // Use option 2: promote the smallest node on the right subtree
    print_tree(&Tree);
    // 추가로 실험한 코드
    // delete_key(150, &Tree);
    // delete_key(100, &Tree);
    // print_tree(&Tree);
    // insert_key(130, &Tree);
```

```
    // insert_key(140, &Tree);
    // insert_key(135, &Tree);
    // print_tree(&Tree);
    // delete_key(120, &Tree);
    // print_tree(&Tree);
    // delete_key(130, &Tree);
    // print_tree(&Tree);
    // delete_key(160, &Tree);
    // print_tree(&Tree);
    // delete_key(10, &Tree);
    return 0;
}
// case 내부에서 변수 선언 불가
```

- Screenshot




HW5-3.c - 2022_first-semester - Visual Studio Code
delete key(50, &tree); // leaf node
//--- Search for the key(490) in the tree...
L(1): The key(490) > node(100) --> right node
L(2): The key(490) > node(150) --> right node
L(2): The key(490) does not exist in the tree!
-- The key(490) is inserted as the [RIGHT] child of node(150)
//--- Search for the key(160) in the tree...
L(1): The key(160) > node(100) --> right node
L(2): The key(160) > node(150) --> right node
L(3): The key(160) < node(490) --> left node
L(3): The key(160) does not exist in the tree!
-- The key(160) is inserted as the [LEFT] child of node(490)
--Print tree in preorder:
[100]
[75]
[50]
[90]
[150]
[120]
[490]
[160]
//--- Search for the key(50) in the tree...
L(1): The key(50) < node(100) --> left node
L(2): The key(50) < node(75) --> left node
L(3): The key(50) is found in the tree!
--Print tree in preorder:
[100]
[75]
[90]
[150]
[120]
[490]
[160]
//--- Search for the key(75) in the tree...
L(1): The key(75) < node(100) --> left node
L(2): The key(75) is found in the tree!
--Print tree in preorder:


HW5-3.c - 2022_first-semester - Visual Studio Code
delete key(50, &tree); // leaf node
[490]
[160]
//--- Search for the key(50) in the tree...
L(1): The key(50) < node(100) --> left node
L(2): The key(50) < node(75) --> left node
L(3): The key(50) is found in the tree!
--Print tree in preorder:
[100]
[75]
[90]
[150]
[120]
[490]
[160]
//--- Search for the key(75) in the tree...
L(1): The key(75) < node(100) --> left node
L(2): The key(75) is found in the tree!
--Print tree in preorder:
[100]
[90]
[150]
[120]
[490]
[160]
//--- Search for the key(150) in the tree...
L(1): The key(150) > node(100) --> right node
L(2): The key(150) is found in the tree!
--Print tree in preorder:
[100]
[90]
[160]
[120]
[490]
mhj@mhj-IdeaPad:~/gitRepo/2022_first-semester/Data Structures/Assignments/As_5$