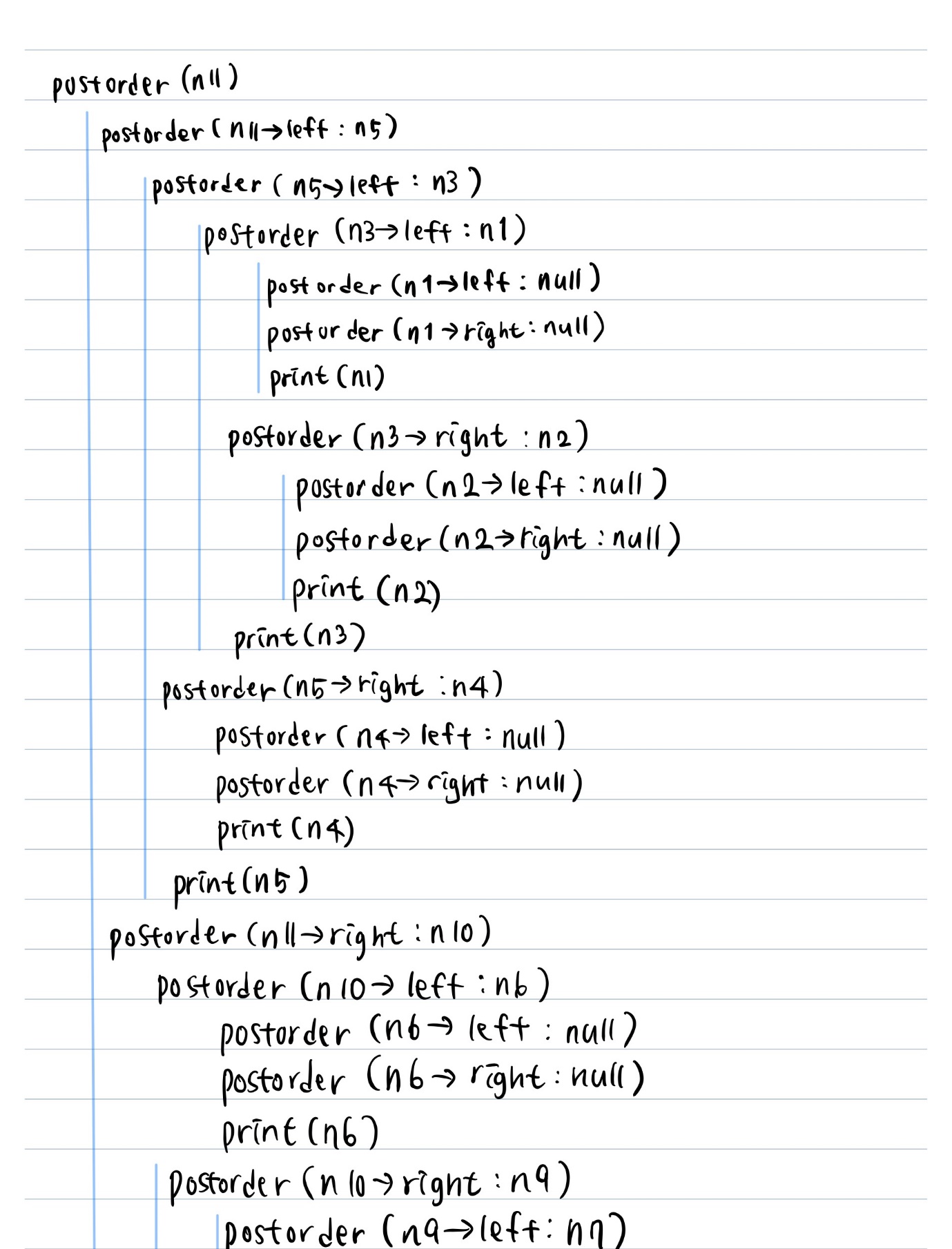
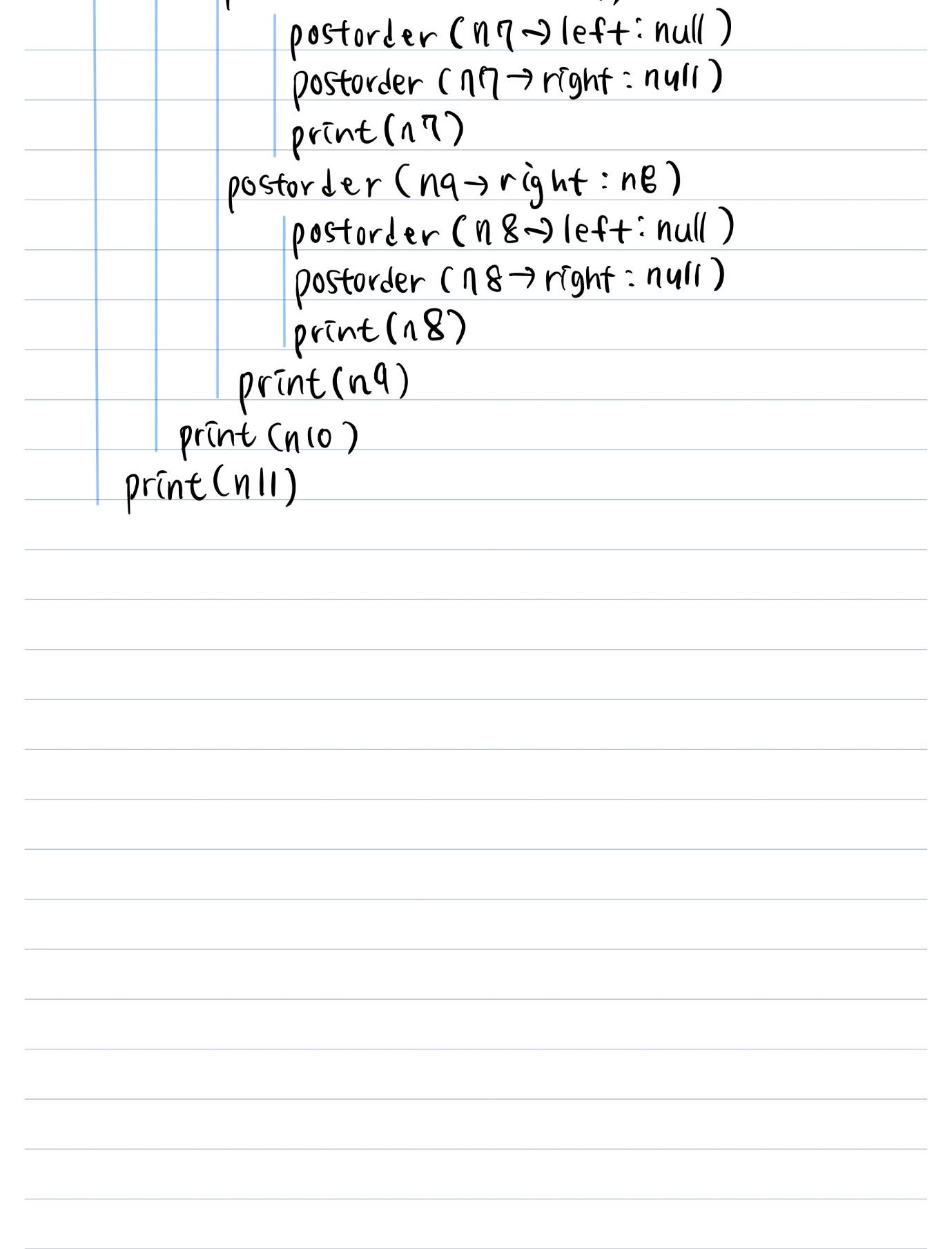
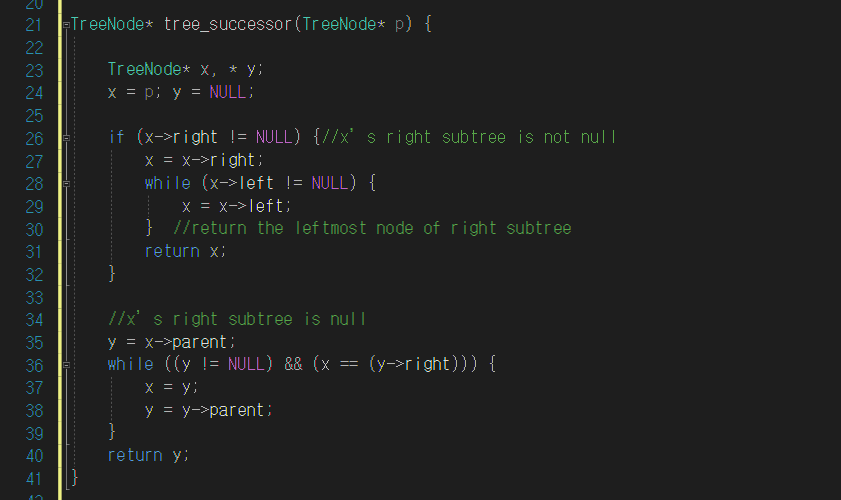
Homework #1





Homework #2

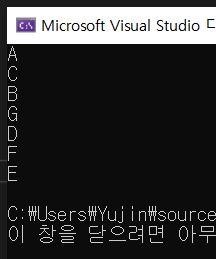
**문제해결방법**



Inorder Traversal의 경우, L->V->R의 순서로 탐색하기 때문에 successor를 다음과 같은 방식으로 찾을 수 있다.

1. 현재 노드 p의 오른쪽 서브트리가 존재하는 경우 : 오른쪽 서브트리의 가장 하단에 있는 왼쪽 노트를 찾아 반환한다.
2. 현재 노드 p의 오른쪽 서브트리가 존재하지 않는 경우 : 부모 노드가 NULL이 아니고 R단계(x==y->right)라면, 반복문을 한번 더 반복하고 그 외의 경우 부모 노드가 successor가 되어 반환된다.

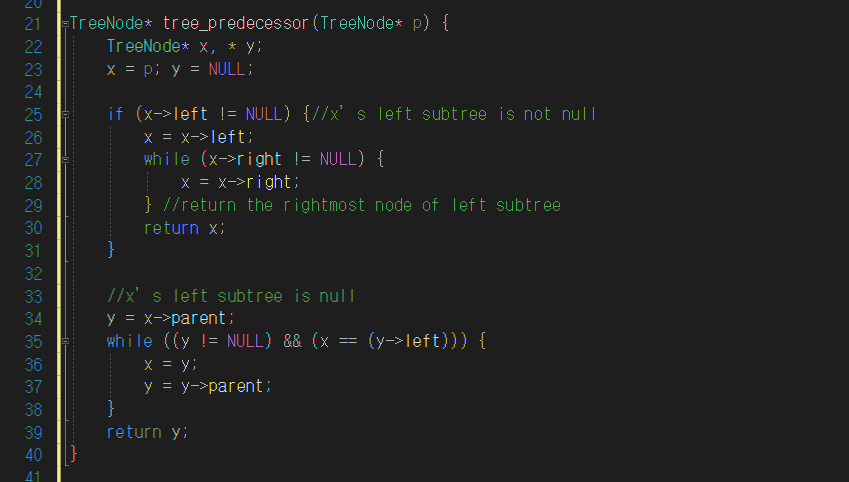
**실행결과**



Homework #3

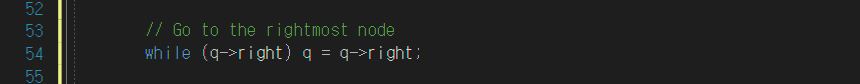
**문제해결방법**

2번에서 작성했던 Successor를 찾는 함수에서 방향만 반대로 바꾸어준다.

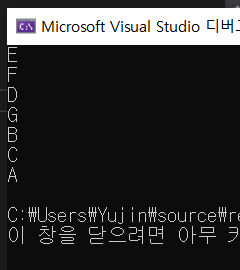


1. 현재 노드 p의 왼쪽 서브트리가 존재하는 경우 : 왼쪽 서브트리의 가장 하단에 있는 오른쪽 노트를 찾아 반환한다.
2. 현재 노드 p의 왼쪽 서브트리가 존재하지 않는 경우 : 부모 노드가 NULL이 아니고 L단계(x==y->left)라면, 반복문을 한번 더 반복하고 그 외의 경우 부모 노드가 successor가 되어 반환된다.

또한, main 함수 내에서 가장 오른쪽에 위치한 노드로 이동하기 위해서 다음 코드를 추가로 작성하였다.



**실행결과**



Homework #4

**문제해결방법**

2번과 3번 문제의 관계에서 알 수 있듯이, successor와 predecessor는 탐색하는 트리와 찾아가는 방향만 변경해주면 된다. 이를 고려하여 deletion 함수의 case3을 수정한다.

**Code Explanation :** void delete\_node(TreeNode\*\* root, int key)

* **변수 설명**
  + pred\_p : predecessor의 부모 노드
  + pred : predecessor이다.
  + 초기화 : 현재 노드를 t로 설정했고 pred\_p를 t로, pred를 t의 왼쪽 자식 노드로 설정한다.
* **Case 3** : If the node to be deleted has both subtrees

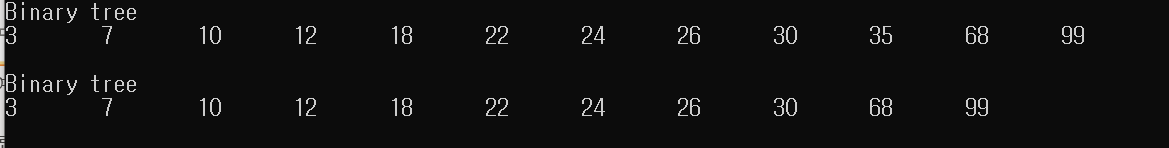
1. pred의 오른쪽 자식 노드가 NULL이 아닐 때까지 한 레벨씩 내려가면서 pred\_p와 pred를 업데이트 한다.
2. predecessor의 부모 노드(pred\_p)의 오른쪽 노드가 predecessor이면, 부모 노드의 오른쪽 노드를 predecessor의 왼쪽 노드로 변경한다.
3. predecessor의 부모 노드의 오른쪽 노드가 predecessor가 아니라면, 부모 노드의 왼쪽 노드를 predecessor의 왼쪽 노드로 업데이트 한다.
4. 현재 노드를 가리키는 t의 key 값은 predecessor의 키 값이 되고 t는 predecessor로 업데이트 된다.

**실행결과**

key = 18



key = 35



key = 7

