실습과제 7.3

[과제 설명]

완전이진트리와 일반이진트리 중에서 완전이진트리는 배열 자료구조가 적합하고, 그 외의 일반적인 이진트리는 연결 리스트 자료구조가 더 적합한 편이다.

연결 리스트를 이용한 이진트리는, 왼쪽 자식 노드의 주소를 저장하는 포인트 변수, 데이터를 저장하는 정수형 변수, 오른쪽 자식 노드의 주소를 저장하는 포인트 변수로 이루어진 Node 구조체를 사용한다.

이진 트리의 노드를 생성할 때와 소멸할 때는 일반적인 양방향 리스트에서 사용하는 노드를 사용하는 방법과 똑같이 한다. 자식 노드를 연결할 때는, 입력 받은 부모노드의 좌우 링크를 확인하고 비어있지 않다면 해당 노드를 삭제하고 그 자리에 새로 입력받은 서브 트리를 연결한다.

루트 노드, 왼쪽 서브 트리, 오른쪽 서브 트리를 순차적으로 방문하는 방법을 순회 알고리즘이라고 하는데, 이진트리의 순회 알고리즘에는 방문 순서에 따라서 크게 전위, 중위, 후위 알고리즘으로 분류할 수 있다. 이 방문 순서의 기준은 루트 노드가 언제 방문되는지에 따라 달라진다.

-전위 순회(Preorder Traversal): 루트 노드를 먼저 방문한다(Root → L → R).
-중위 순회(Inorder Traversal): 루트 노드를 중간에 방문한다(L → Root → R).
-후위 순회(Postorder Traversal): 루트 노드를 마지막에 방문한다(L → R → Root).

[코드]

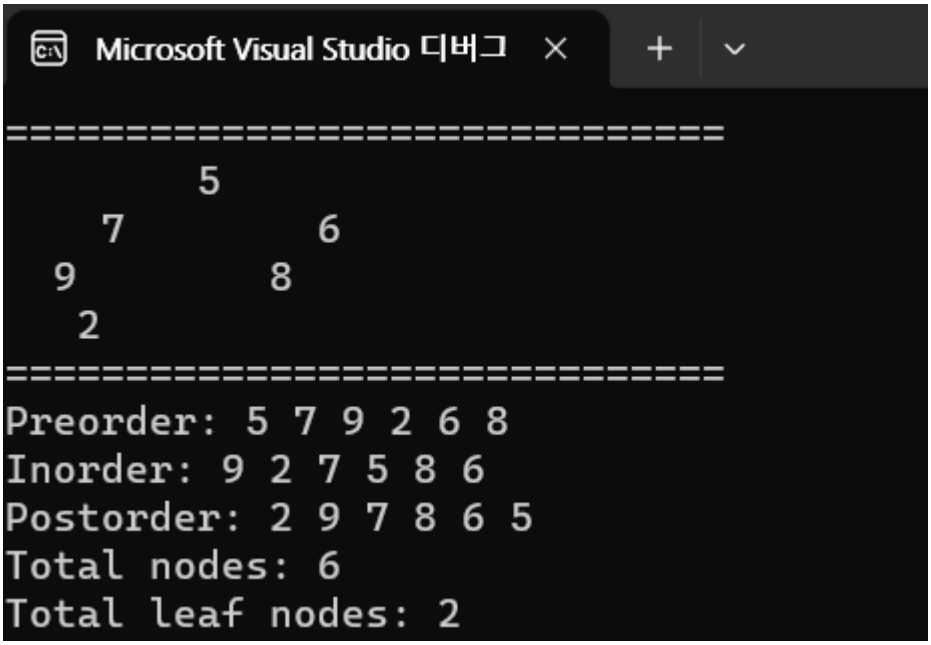
메인 함수와 header 파일, 출력 함수는 교안에서 미리 구현된 것을 사용했으므로 보고서에는 생략하고 구현된 함수 세부 내용만 첨부한다.

```
#include "MyBinaryTree.h"

//노드 생성
BT_Node* BT_Create_Node(int newData) {
```

```
  BT_Node* newNode = (BT_Node*)malloc(sizeof(BT_Node)); //동적 할당을 통한 노드 생성
  newNode->data = newData;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

//노드 소멸
void BT_Destroy_Node(BT_Node* node) {
  if (node != NULL)
    free(node);
}

//왼쪽 자식으로 연결
//입력 받은 부모 노드의 왼쪽에 입력 받은 서브 트리를 구성한다.
void BT_Make_Left_Sub_Tree(BT_Node* parent, BT_Node* sub) {
  if (parent->left != NULL)
    BT_Destroy_Node(parent->left);
  //왼쪽 자식이 비었는지 확인하고, 비어있지 않다면 왼쪽 자식을 소멸시킨다.
  parent->left = sub;
}

//오른쪽 자식으로 연결
//입력 받은 부모 노드의 오른쪽에 입력 받은 서브 트리를 구성한다.
void BT_Make_Right_Sub_Tree(BT_Node* parent, BT_Node* sub) {
  if (parent->right != NULL)
    BT_Destroy_Node(parent->right);
  //오른쪽 자식이 비었는지 확인하고, 비어있지 않다면 오른쪽 자식을 소멸시킨다.
  parent->right = sub;
}

//전위 순회 함수
//Root 노드를 먼저 방문한다.
//Root → L → R
void BT_Preorder_Traversal(BT_Node* node) {
  if (node != NULL) {
    printf("%d ", node->data);
    BT_Preorder_Traversal(node->left);
    BT_Preorder_Traversal(node->right);
  }
}

//중위 순회 함수
//Root 노드를 중간에 방문한다.
//L → Root → R
void BT_Inorder_Traversal(BT_Node* node) {
  if (node != NULL) {
    BT_Inorder_Traversal(node->left);
    printf("%d ", node->data);
    BT_Inorder_Traversal(node->right);
  }
}

//후위 순회 함수
//Root 노드를 마지막에 방문한다.
//L → R → Root
void BT_Postorder_Traversal(BT_Node* node) {
```

```
  if (node != NULL) {
    BT_Postorder_Traversal(node->left);
    BT_Postorder_Traversal(node->right);
    printf("%d ", node->data);
  }
}

//이진 트리의 노드의 수 계산
//전체 트리를 순회하면서 노드의 수를 센다.
//루트 노드를 먼저 방문하는 전위 순회 방식을 이용한다.
void BT_Count_Node(BT_Node* node, int* count) {
  if (node != NULL) {
    (*count)++;
    BT_Count_Node(node->left, count);
    BT_Count_Node(node->right, count);
  }
}

//리프 노드(말단 노드)의 수 계산
//중위 순회를 통해 리프 노드를 확인한다.
void BT_Count_Leaf(BT_Node* node, int* count) {
  if (node != NULL) {
    BT_Count_Leaf(node->left, count);
    if (node->left == NULL && node->right == NULL)
      (*count)++;
    BT_Count_Leaf(node->right, count);
  }
}
```

[실행 결과]



[과제에 대한 고찰]

왼쪽, 오른쪽 자식으로 연결하는 함수를 보았을 때, 만약 입력 받은 부모 노드의 왼쪽, 오른쪽에 이미 자식이 있어서 NULL이 아니라면 그 부분을 free로 날려주고 입력 받은 자식 노드를 붙여주는 방식이다. 이런 방식에서는 free로 날려줄 때 해당하는 하나의 노드만 날리기 때문에 부모 노드에 연결되어 있는 서브 트리가 하나의 노드가 아닌 경우에는 메모리를 낭비하게 되는 것 같다. 부모 노드에 연결하고 싶은 서브 트리는 하나의 노드가 아니어도, 링크를 연결해주는 방식이기 때문에 상관 없이 잘 연결되는 것 같다. 이런 부분을 보완하는 방법에는 뭐가 있을지 더 학습해보는 것도 좋을 것 같다.