실습과제 8.1

[과제 설명]

이진 탐색 트리(Binary Search Tree: BST)는 다음의 특징을 갖는다.

-모든 노드는 유일한 키 값을 갖는다.
-부모 노드의 키 값은 왼쪽 서브 트리에서 모든 노드의 키 값보다 크다.
-부모 노드의 키 값은 오른쪽 서브 트리에서 모든 노드의 키 값보다 작다.
-각 서브 트리의 모든 노드의 키 값은 위의 조건을 충족해야 한다.

이진 탐색 트리의 장점은 효율적인 탐색이 가능하다는 것이다.
이진 탐색 트리에서 가장 작은 값은 가장 왼쪽 단말 노드이다. 따라서, 최소값을 찾으려면 계속해서 왼쪽 서브 트리로 이동하면 된다. 반대로, 이진 탐색 트리에서 가장 큰 값은 가장 오른쪽 단말 노드이다. 따라서, 최대값을 찾으려면 계속해서 오른쪽 서브 트리로 이동하면 된다.
또, 이진 탐색 트리에서 주어진 값을 찾을 때는, 루트 노드의 값을 비교해서 찾으려는 키 값이 루트 노드의 키 값보다 작으면 왼쪽으로, 크면 오른쪽으로 탐색한다. 이 탐색 함수는 재귀 방식과 반복문 방식, 2가지로 구현 가능하다. 아래쪽의 코드에서는 재귀 방식으로 구현하였다.

이진 탐색 트리에서 새로운 키 값을 삽입할 때는, 루트 노드부터 시작해서 키 값과 기존 노드의 값을 비교한다. 키 값이 노드의 키 값보다 작으면 왼쪽으로, 크면 오른쪽으로 이동한다. 비교 대상이 없다면 해당 위치에 키 값을 삽입하고, 만약 같은 값이 있다면 삽입하지 않고 리턴한다. 중복된 값이 없어야 하기 때문이다.

반면, 이진 탐색 트리에서 키 값을 삭제하는 것은 3가지 경우를 나눠서 구현 해야 한다.

-삭제할 노드가 단말 노드
삭제할 노드의 정보와 삭제할 노드의 부모 노드의 정보를 획득한다. 부모 노드의 자식 노드 정보를 NULL로 변경하고, 해당하는 노드의 메모리를 해제한다.
-삭제할 노드가 한 개의 서브 트리를 가지고 있는 경우
삭제할 노드의 정보와 삭제할 노드의 부모 노드의 정보를 획득한다. 부모 노드의 서브 트리 부분에 삭제할 노드의 서브 트리를 연결시킨다. 그 뒤, 해당하는 노드의 메모리를 해제한다.
-삭제할 노드에 두 개의 서브 트리를 가지고 있는 경우
먼저 삭제할 노드의 정보와 삭제할 노드의 부모 정보를 획득한다. 그런 뒤, 삭제할 노드

의 왼쪽 서브트리의 최대값과 오른쪽 서브트리의 최소값을 비교한다. 그 중, 더 큰 값이 새로운 부모노드가 된다. 삭제할 노드의 자리를 비교해서 나온 노드로 대체한 후, 마지막으로 삭제할 노드의 메모리를 해제한다.

[코드]

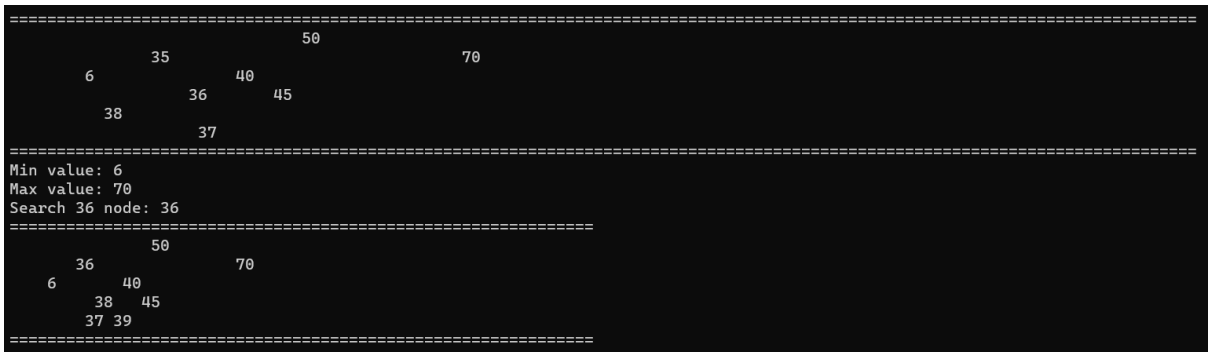
메인 함수와 header 파일, 출력 함수는 교안에서 미리 구현된 것을 사용했으므로 보고서에는 생략하고 구현된 함수 세부 내용만 첨부한다.

```
BT_Node* BST_Create_Node(int newData) {
  BT_Node* newNode = (BT_Node*)malloc(sizeof(BT_Node));
  newNode->data = newData;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}
void BST_Destroy_Node(BT_Node* node) {
  if (node != NULL)
    free(node);
}

//BST 관련 함수
//가장 작은 값 탐색
int BST_Min(BT_Node* root) {
  if (root == NULL) exit(1);

  BT_Node* iter = root;

  while (iter->left != NULL)
    iter = iter->left;
  //가장 왼쪽 단말 노드가 최소값이기 때문에, 왼쪽으로 계속 탐색한다.

  return iter->data;
}

//가장 큰 값 탐색
int BST_Max(BT_Node* root) {
  if (root == NULL) exit(1);

  BT_Node* iter = root;

  while (iter->right != NULL)
    iter = iter->right;
  //가장 오른쪽 값이 최대값이기 때문에, 오른쪽으로 계속 탐색한다.
```

```
  return iter->data;
}

//주어진 키 값을 가진 노드 탐색
//대소 비교를 통해 키 값을 탐색한다.
//재귀 방식 알고리즘과 반복문 중에 재귀 방식으로 구현하였다.
BT_Node* BST_Search(BT_Node* root, int target) {
  if (root == NULL) return NULL;
  if (target == root->data) return root;
  if (target < root->data)
    return BST_Search(root->left, target);
  //타겟이 키 값보다 작으면 왼쪽 서브 트리로 탐색
  else
    return BST_Search(root->right, target);
  //타겟이 키 값보다 크면 오른쪽 서브 트리로 탐색
}

//새로운 키 값을 삽입
void BST_Insert(BT_Node** root, int key) {
  BT_Node* newNode = BST_Create_Node(key);

  if ((*root) == NULL)
    *root = newNode;
  //루트 노드가 비었다면, newNode를 루트 노드로
  else {
    BT_Node* parent = NULL;
    BT_Node* iter = *root;

    while (iter != NULL) {
      parent = iter;
      if (key == iter->data) return; //중복 값은 허용하지 않음
      if (key < iter->data) iter = iter->left;
      else iter = iter->right;
      //키 값이 iter의 데이터보다 작으면 왼쪽 노드로 이동
      //키 값이 iter의 데이터보다 크면 오른쪽 노드로 이동
    }

    if (key < parent->data) parent->left = newNode;
    else parent->right = newNode;
    //말단 노드를 부모 노드로 하고,
    //부모 노드의 데이터보다 작으면 왼쪽 자손으로 만들고
    //크면 오른쪽 자손으로 만든다.
  }
}

//주어진 키 값을 삭제
void BST_Delete(BT_Node** root, int key) {
  BT_Node* vRoot = (BT_Node*)malloc(sizeof(BT_Node));
  vRoot->right = *root; //가상 노드 생성
  BT_Node* iter = *root; //가상의 노드가 루트 노드를 가리키게 한다.
  BT_Node* parent = vRoot; //초기 부모 노드

  while (iter != NULL && iter->data != key) {
    parent = iter;
    if (key < iter->data) iter = iter->left;
    else iter = iter->right;
  }
```

```
  if (iter == NULL) {
    puts("해당하는 키 값이 없습니다.");
    return;
  }//키 값이 없으면 오류 메세지 출력 후 함수 종료

  BT_Node* delNode = iter; //iter를 이용해 탐색한 노드를 삭제할 노드로 지정

  //case 1: 단말 노드 삭제
  if (delNode->left == NULL && delNode->right == NULL) {
    if (parent->left == delNode) parent->left = NULL;
    else parent->right = NULL;
  }

  //case 2: 하나의 서브 트리를 가지는 노드를 삭제
  else if (delNode->left == NULL || delNode->right == NULL) {
    BT_Node* child;

    if (delNode->left == NULL) child = delNode->right;
    else child = delNode->left;
    //하나의 서브 트리를 가지기 때문에 한 쪽이 NULL이면 반대쪽에 서브 트리가 있다.

    if (parent->left == delNode) parent->left = child;
    else parent->right = child;
    //삭제할 노드의 부모에 가지고 있던 서브 트리를 연결한다.
  }

  //case 3: 두 개의 서브 트리를 가지는 노드를 삭제
  else {
    BT_Node* Suc = delNode->right; //대체 노드를 저장한다.
    BT_Node* pSuc = delNode; //대체 노드의 부모 노드를 저장한다.

    while (Suc->left != NULL) {
      pSuc = Suc;
      Suc = Suc->left;
    } //왼쪽 서브 트리로 이동

    delNode->data = Suc->data; //대체 노드의 키 값을 복사한다.
    if (pSuc->left == Suc)
      pSuc->left = Suc->right;
    else
      pSuc->right = Suc->right;
    delNode = Suc;
  }

  if (vRoot->right != *root)
    *root = vRoot->right;
  //루트 노드가 변경되었다면, 새로운 루트 노드 정보를 업데이트한다.

  BST_Destroy_Node(vRoot);
  BST_Destroy_Node(delNode);
  //가상 노드와 삭제할 노드 모두 메모리 해제
}
```

[실행 결과]



[과제에 대한 고찰]

이진 탐색 트리는 이름에서도 알 수 있듯, 효율적인 탐색에 집중을 맞춘 자료구조인 것 같다. 그래서인지 삽입까지는 크게 복잡하지 않게 되는 것 같지만, 삭제할 때는 복잡해진다. 그래서 이것에 대해 고민해봤는데, 이진 탐색 트리는 검색이 자주 필요한 데이터 모음 중에, 많은 데이터를 초기에 삽입하고 (중간에 삽입은 많이 해도) 삭제는 잘 하지 않는 데이터에 사용하는 것이 좋을 것 같다고 생각한다.