실습과제 3.2

[과제 설명]

과제 3.1에서는 한 파일에 모든 코드를 구현했다면, 3.2에서는 헤더 파일 MyLinkedList.h 과 헤더 파일에 있는 함수의 세부 내용을 구현하는 MyLinkedList.c, 그리고 코드의 main이 들어가있는 main.c를 분리해서 코드를 구현한다.

이 때, main파일에는 #include "MyLinkedList.h" 를 이용해 헤더 파일인 MyLinkedList.h를 포함시킨다. #include를 할 때 <>와 ""의 차이는 크게 없으나, 관용적으로 stdio.h 같은 standard library header 같은 종류에 쓰고, ""은 user defined header, 그러니까 사용자가 직접 정의한 헤더 파일에 주로 쓴다.

이런 식으로 코드를 여러 파일로 분리시키는 이유는, 전체 프로그램 관리도 용이한데다가(코드가 길어지면 가독성이 떨어지기 때문이다) 코드의 재사용에도 용이하기 때문이다.

▪노드 생성: Node* Create_Node(int newData)
▪노드 소멸: void Destroy_Node(Node* node)
▪노드 추가: void Append_Node(Node** head, Node* newNode)
▪노드 출력: Print_Linked_List(Node* head)

이 네 가지 함수는 3.1에서 이미 구현했기 때문에, head 파일의 함수의 세부 내용을 구현하는 MyLinkedList.c 파일에 그대로 코드를 넣었고, 아래의 세 함수만 새로 구현했다.

▪노드 탐색: Node* Get_Node(Node* head, int pos)
노드의 특정 위치로 찾는 기능을 수행한다.
이 때 노드 포인터 변수로 된 반복자를 생성해서 head의 주소를 넣는다. 특정 위치와 같은지는 따로 flag라는 변수를 이용해서 같을 때까지 반복문을 돌리면서 1씩 증가한다. 그와 동시에 반복자도 순차적으로 다음 노드의 주소로 옮겨간다. 그리고 특정 위치와 같으면 해당 위치의 주소를 반환한다.

▪노드 삭제: void Remove_Node(Node** head, Node* targetNode)
특정 위치에 있는 노드를 삭제하는 기능을 수행한다.
노드 소멸을 시키는 함수, Destroy_Node와 차이점은, 특정 위치에 있는 노드도 삭제가 가능하다는 것이다. 먼저 head가 비어있다면 함수를 바로 종료시키고, 만약 그렇지 않는다면 head의 주소를 담는 노드 더블 포인터 변수로 된 반복자를 생성시킨다. 그렇게 해서 반복자와 삭제하려는 노드가 같다면, 해당 노드를 가리키고 있던 이전 노드의 link에 삭제하려는 노드의 다음 주소를 넣고, Destroy_Node 함수를 호출해 해당 노드를 삭제시킨다.

▪노드 삽입: void Insert_Node_After(Node* currentNode, Node* newNode)
특정 위치에 새로운 노드를 삽입하는 기능을 수행한다.
삽입하려는 위치와 새로 삽입하려는 노드를 인자로 받는데, currentNode의 link가 가리키고 있던 것을 먼저 newNode의 링크에 복사한다. 그러고 나서 currentNode의 link가 newNode를 가리키게 하면 된다.

[MyLinkedList.h 코드]

```
#ifndef __MY_LINKED_LIST_H__
#define __MY_LINKED_LIST_H__
#include <stdio.h>

typedef struct Node { //Node 구조체
  int data; //데이터 필드
  struct Node* link;// 다음 노드의 주소를 저장
}Node;

Node* Create_Node(int newData); //노드 생성
void Destroy_Node(Node* node); //노드 소멸
void Append_Node(Node** head, Node* newNode);//노드 추가
Node* Get_Node(Node* head, int pos); //노드 탐색
void Remove_Node(Node** head, Node* targetNode);//노드 삭제
void Insert_Node_After(Node* currentNode, Node* newNode); //노드 삽입
void Print_Liked_List(Node* head); //연결리스트 출력 함수

#endif
```

[MyLinkedList.c 코드]

```
#include <stdlib.h>
#include "MyLinkedList.h"

Node* Create_Node(int newData) //노드 생성
{
  Node* newNode = (Node*)malloc(sizeof(Node));
  //함수가 종료된 후에도 노드가 유지되게 동적할당을 사용
  newNode->data = newData;
  newNode->link = NULL; //쓰레기 값 대신 null로 한다.

  return newNode; //노드의 주소값 반환
}

void Destroy_Node(Node* node) { //노드 소멸
  if (node != NULL) {
```

```
    free(node);
  }
}

void Append_Node(Node** head, Node* newNode) {//노드 추가
  if ((*head) == NULL) //현재 head가 비어있다면
    *head = newNode; //새로운 노드를 head로 지정
  else { //비어있지 않다면
    Node* tail = (*head);
    while (tail->link != NULL) {
      tail = tail->link;
    } //비어있는 노드까지 탐색
    tail->link = newNode; //제일 끝부분에 새로운 노드 추가
  }
}

Node* Get_Node(Node* head, int pos) { //노드 탐색
  Node* iter = head;
  int flag = 0;

  while (iter != NULL) {
    if (flag == pos) //반복하면서 특정 노드 위치와 같은 곳을 찾는다.
      return iter; //특정 위치와 같으면 주소를 반환한다.
    iter = iter->link;
    flag++;
    //특정 위치와 같지 않으면 반복자를 그 다음 것으로 넘기고 1 증가시킨다.
  }

  return NULL;
}

void Remove_Node(Node** head, Node* targetNode) {//노드 삭제
  if (head == NULL) //head가 비어있으면 바로 함수를 종료시킨다.
    return;

  Node** iter = head;

  while (*iter != NULL) {
    //head부터 시작하는 반복자, iter가 비어있는지를 검사해서 반복하기 때문에 head인지 아닌지를 구분하지 않아도 된다.
    if ((*iter) == targetNode) { //반복자와 삭제하려는 노드 비교
      Node* target = *iter;
      (*iter) = (*iter)->link; //해당하는 노드의 link를 다음 주소로 옮긴다음
      Destroy_Node(target); //목표를 삭제한다.
      return;
    }
    iter = &(*iter)->link; //해당하는 목표가 아니라면 다음 노드로
  }
}

void Insert_Node_After(Node* currentNode, Node* newNode) { //노드 삽입
  newNode->link = currentNode->link;
  //삽입될 노드에 그 전의 노드가 가지고 있던 주소를 복사한다.
  currentNode->link = newNode;
  //그 전의 노드의 link에 새로 삽입될 노드의 주소값을 넣어준다.
}

void Print_Liked_List(Node* head) { //연결리스트 출력 함수
  Node* iter = head;
  int i = 0;
  while (iter != NULL) {
    printf("node[%d]: %d", i, iter->data);
    iter = iter->link;
    if (iter != NULL) printf(" → ");
```

```
    i++;
  } //반복자가 비어있을 때까지 탐색하면서 하나씩 출력한다.
  puts("");
}
```

[main 코드]

```
#include "MyLinkedList.h"
//MyLinkedList.h에 stdio.h를 include 했기 때문에 main에는 오히려 포함시키지 않아도 된다.

int main(void) {
  Node* head = NULL;
  //기본 노드 생성
  Append_Node(&head, Create_Node(15));
  Append_Node(&head, Create_Node(31));
  Print_Liked_List(head);

  //Get_Node() 함수 기능 테스트
  Node* temp = Get_Node(head, 0);
  printf("Get_Node() test: %d\n", temp->data);

  //Insert_Node_After() 함수 기능 테스트
  Insert_Node_After(Get_Node(head, 0), Create_Node(25));
  Print_Liked_List(head);

  //Remove_Node() 함수 기능 테스트
  Remove_Node(&head, Get_Node(head, 0));
  Print_Liked_List(head);

  return 0;
}
```

[과제에 대한 고찰]

파일을 분리하는 건 자바에서만 해보고 c에서는 해보지 못했는데 이렇게 분리해보니까 생각했던 것보다 코딩하기도, 코드를 보기도 더 편한 것 같다. 이렇게 하기 전에는 main 부분이 잘 안 보이다 보니, 코드의 양이 일정 이상 넘어가면 위쪽에 사용자 정의 함수의 프로토 타입만 모아두고 아래쪽에 세부 내용을 넣는 편이었는데 이 편이 코드 정리하기에는 훨씬 더 좋은 것 같다.

연결 리스트에서 노드를 삭제하는 함수도 여러 가지고 구현할 수 있었는데, 특히 참고와 에러 해결을 위해 구글에서 찾아보니 대부분 더블 포인터 변수가 아닌 포인터 변수와 타겟 값도 노드 포인터가 아닌 정수로 값을 받아오는 게 많았다. 대부분의 포스팅에 더블 포인터로 받는 것과 차이가 나는지

는 적혀 있지 않아서, 리소스적인 측면에서 차이가 많이 있는지 궁금했다.
  또, 노드 삽입과 반대로 하면 될 것 같다고 생각했는데, 생각했던 것처럼 되지는 않아서 다소 힘들었던 같다.