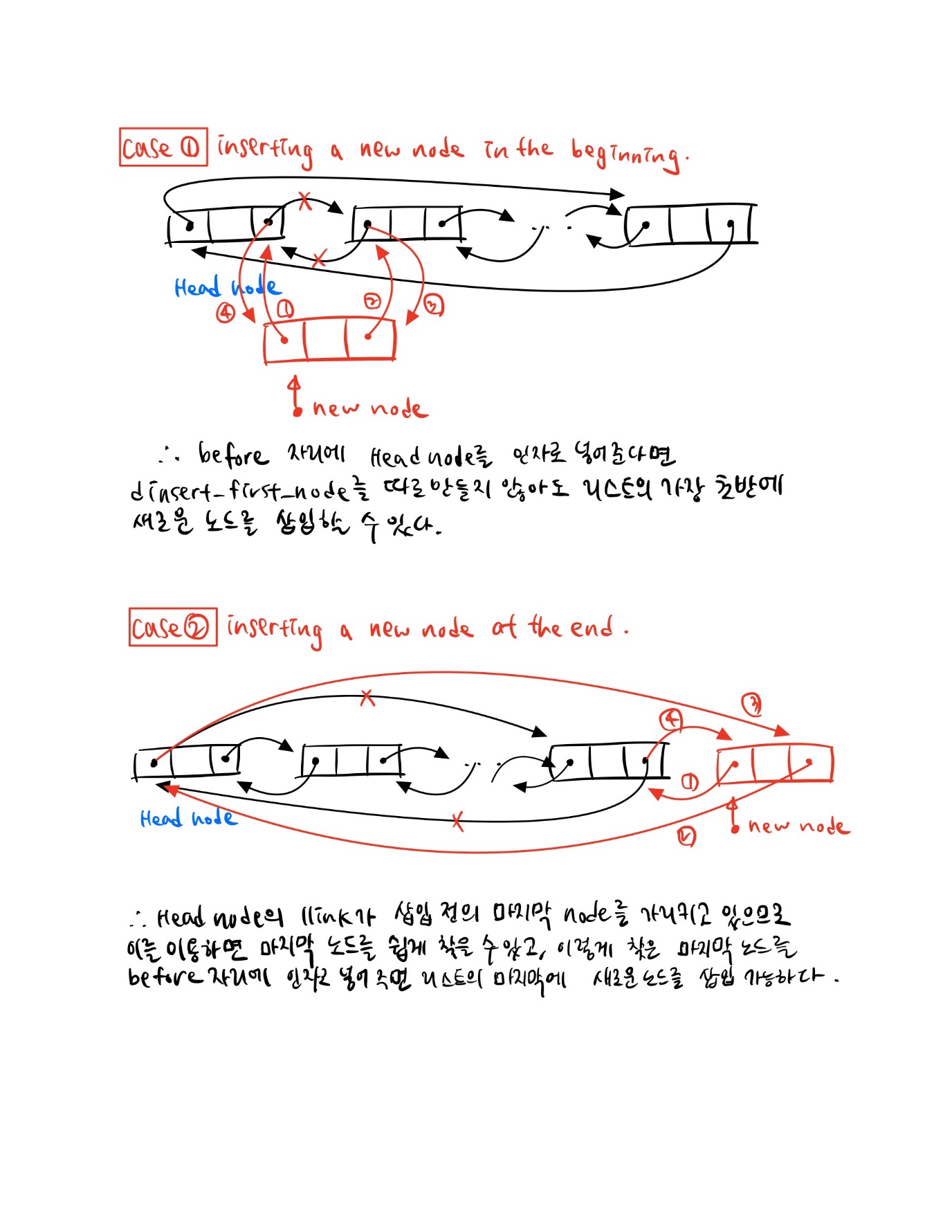
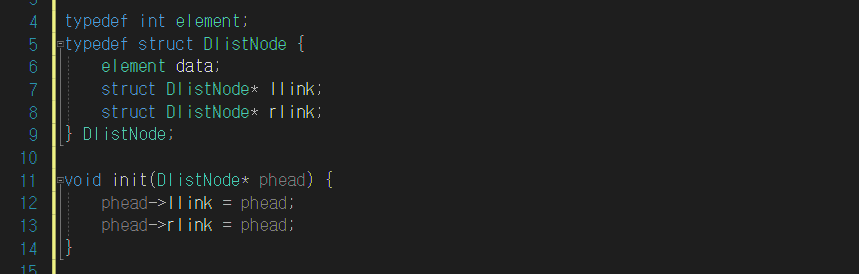
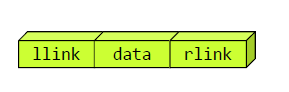
Homework 1



**Variable**

p.40에 있는 doubly linked list의 node를 그대로 사용하였다.

|  |  |
| --- | --- |
| **DistNode 구조체** | |
| **data type** | **name** |
| element (=int) | data |
| DistNode \* | llink |
| DistNode \* | rlink |

**DistNode 구조체**

* data : Linked list의 노드 안에 저장할 값을 담고 있는 구조체 멤버
* llink : 이전 노드를 가리키고 있는 구조체 포인터
* rlink : 다음 노드를 가리키고 있는 구조체 포인터
* Head node로 사용될 경우 actual data는 포함하지 않는다.

**Function**

1. void init(DlistNode\* phead)

main 함수에서 메모리를 할당받은 후, 포인터 값을 받아와 Head\_node를 초기화한다. 이때는 노드가 하나도 연결되지 않은 상태이므로 llink와 rlink에 자기 자신의 포인터 값을 넣어 초기화한다.

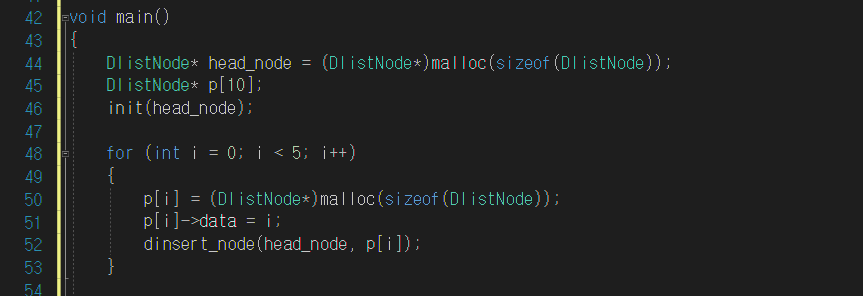
1. void display(DlistNode\* phead)

linked list에 저장되어 있는 모든 노드의 값을 <--- | p->llink | p->data | p->rlink | ---> 형태로 차례대로 출력한다.

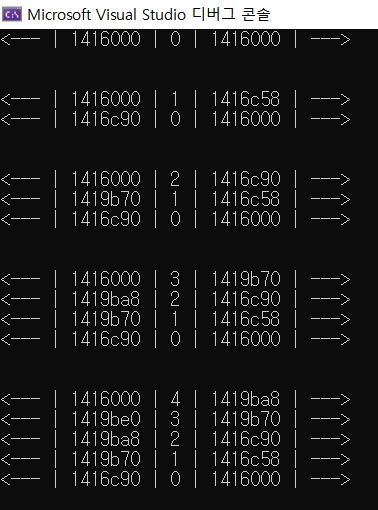
1. void dinsert\_node(DlistNode\* before, DlistNode\* new\_node)

삽입하고자 하는 위치의 직전 노드의 포인터 before과 새롭게 삽입될 노드의 포인터 new\_node를 인자로 받아 linked list의 원하는 위치에 노드를 삽입한다.

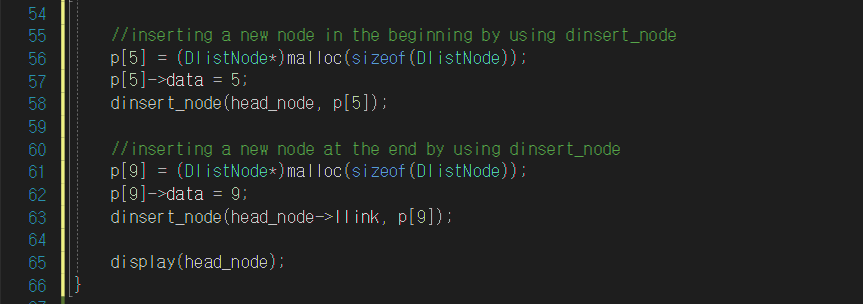
1. void main()



노드 10개를 생성한 후 head\_node를 초기화한다. 0~4까지 data값을 채운 노드를 head\_node의 뒤에 차례로 저장한다. 이 때 for문을 돌면서 저장되는 data 값들을 display 함수를 이용해 출력해보면



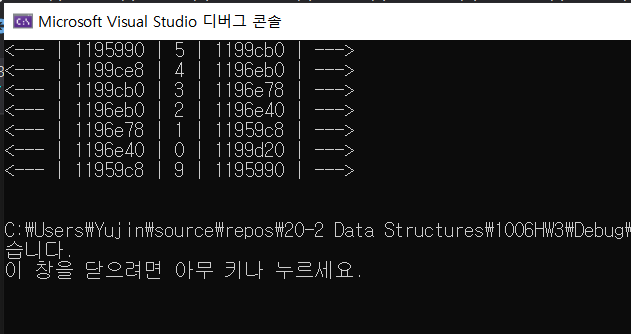
* 이와 같이 head\_node의 바로 뒤, 그러니까 가장 처음 위치에 정상적으로 노드가 저장되는 것을 확인할 수 있다.



위에서도 알 수 있듯 dinsert\_node(head\_node, p[5]); 코드를 실행하여 dinsert\_node() 함수의 before의 자리에 head\_node의 포인터 값을 넣어준다면 5의 data 값을 가진 노드를 doubly linked list의 가장 처음에 삽입할 수 있다.

가장 마지막에 노드를 삽입하고 싶다면 (여기서는 9의 data 값을 가진 노드를 예시로 사용했다) head\_node의 llink에 저장된 값이 가장 마지막 노드가 되므로, 이 마지막 노드의 위치를 before의 인자로 보내주면 된다.

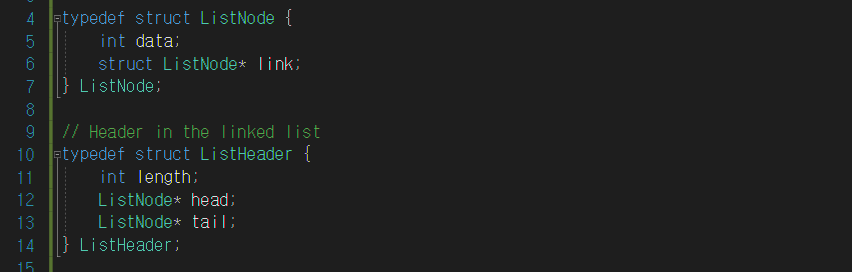
**실행결과**



코드를 실행하면 첫번째 노드에 5가, 마지막 노드에 9가 삽입된 것을 확인할 수 있다. 따라서, dinsert\_node() 함수만 사용해서 doubly linked list의 맨 처음과 끝에 노드를 삽입할 수 있다.

Homework 2

**Variable**



|  |  |
| --- | --- |
| **ListNode 구조체** | |
| **data type** | **name** |
| int | data |
| ListNode \* | link |

|  |  |
| --- | --- |
| **ListHeader 구조체** | |
| **data type** | **name** |
| int | length |
| ListNode \* | head |
| ListNode \* | tail |

1. **ListNode 구조체**

* int : Linked list의 노드 안에 저장할 값을 담고 있는 구조체 멤버
* ListNode : 다음 노드를 가리키고 있는 구조체 포인터

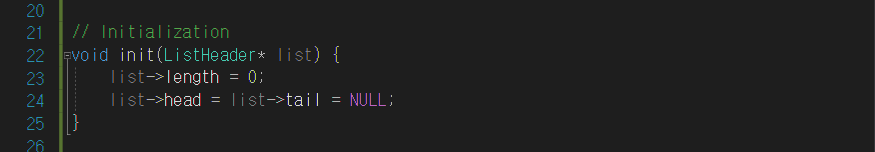
1. **ListHeader 구조체**

* length : LinkedList의 길이 = 노드의 개수를 저장
* head : 제일 처음 노드를 가리키고 있는 구조체 포인터
* tail : 제일 마지막 노드를 가리키고 있는 구조체 포인터
* ListHeader를 위와 같이 구성함으로써, 더블 포인터를 사용할 필요가 없어짐.

**Function**

1. void init(ListHeader\* list)

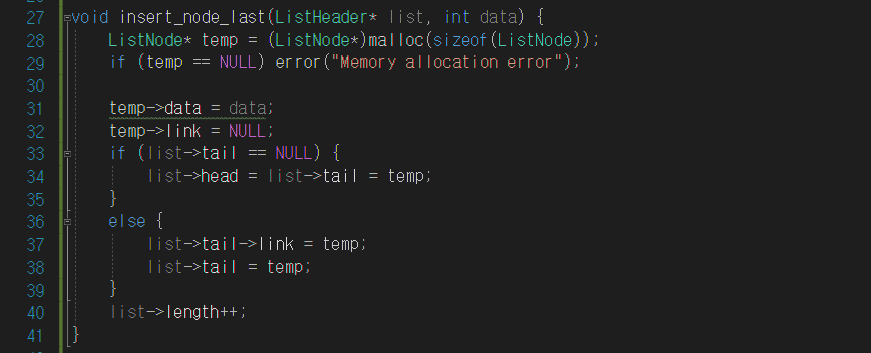
: ListHeader의 초기값을 설정해주는 함수



처음 만들어질 때 listHeader 구조체는 아직 node를 가지고 있지 않으므로 length값은 0, head와 tail pointer는 아무것도 가리키지 않도록 초기화해준다.

1. void insert\_node\_last(ListHeader\* list, int data)

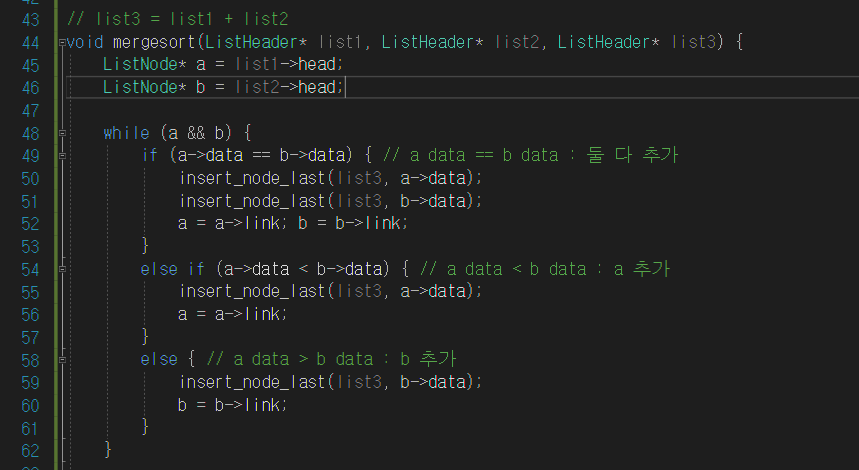
: Linked list의 가장 마지막 노드에 데이터를 삽입하는 함수



ListNode 포인터 변수 temp를 만들고, malloc 함수를 이용하여 크기에 맞는 메모리를 동적으로 할당해준다. 만약 메모리 할당에 실패하면 에러 메세지를 출력한다. ListNode의 포인터 변수 temp를 이용하여 listNode 구조체의 멤버에 접근해 입력받은 data값을 저장하고, (linked list의 맨 마지막 위치에 삽입될 것이므로) link 값은 아무것도 가리키지 않도록 설정한다. 만약 리스트가 비어있어 listHeader의 tail 값이 Null을 가리키고 있다면, list의 head와 tail값이 모두 temp 를 가리키도록 설정한다. 그렇지 않다면 (리스트가 비어있지 않다면), 삽입하기 전 마지막 노드의 link가 temp를 가리키도록 연결하고 listHeader의 tail 값이 temp를 가리키도록 변경해준다. 노드가 하나 삽입되었으므로 listHeader의 length값을 1 증가시킨다.

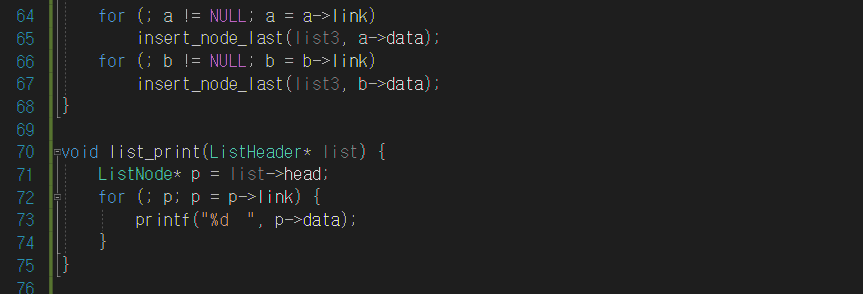
1. void mergesort(ListHeader\* list1, ListHeader\* list2, ListHeader\* list3)

: linked list1과 2에 저장된 값을 오름차순으로 list3에 추가해주는 함수.



인자로 전달받은 List 1과 2의 ListHeader를 이용하여 첫번째 노드부터 차례로 비교한다.

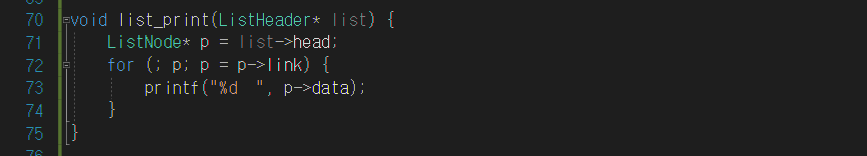
둘 중 하나의 노드가 Null을 가리킬 때까지, 두 노드의 data값의 같으면 둘 다 list3의 마지막 부분에 추가한다. 구조체 포인터 변수 a와 b 모두 다음 노드를 가리키게 된다. 만약 a의 data 값이 b의 data값보다 작으면 작은값(a의 data값)을 list3에 추가한 후, a의 포인터만 다음 노드로 이동하게 된다. (b는 그대로) 반대로, b의 data값이 a의 data 값보다 작으면 b의 data 값을 list3에 추가한 후 b의 포인터만 다음 노드로 이동하게 된다.



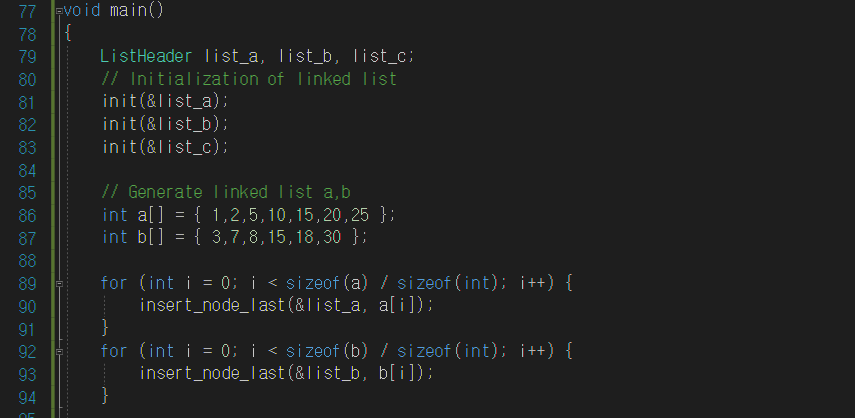
만약 두 list 중 하나의 data값만 모두 추가되어 Null을 가리키게 되면, 남은 list의 data 값을 그대로 list3의 마지막 부분에 추가한다.

1. void list\_print(ListHeader\* list)

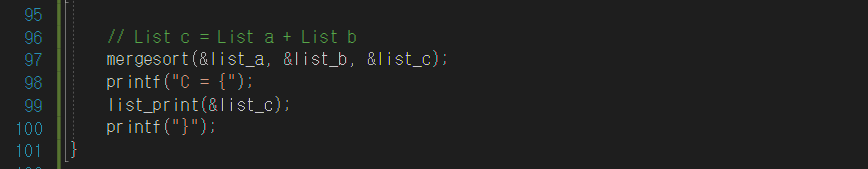
: ListHeader의 주소를 인자로 받아 linked list에 저장된 모든 data값을 차례로 출력하는 함수.



1. void main()



ListHeader a,b,c를 선언하고 초기화해준다. 이후 a와 b의 data값을 정렬된 상태로 linked list에 저장한다.



앞에서 정의한 mergesort와 List\_print 함수를 이용하여 list a,b에 있는 값을 오름차순으로 합쳐서 출력한다. List\_a,b,c가 ListHeader data type으로 정의되어 있으므로, 함수를 실행할 때 &연산자를 이용하여 포인터 형태로 보내줘야 한다.

**시간복잡도**

시간복잡도는 mergesort()와 insert\_node\_last() 함수를 통해 계산할 수 있다.

List\_a의 data 개수를 n, list\_b의 data 개수를 m이라고 가정하면,

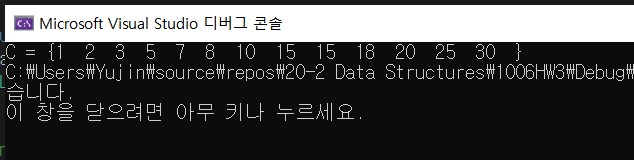
mergesort() 함수 내부에서 while 문을 돌면서 a->data와 b->data를 최악의 경우 n+m번 비교한다.

비교 결과에 따라 if, else if, else 중 하나의 조건문이 실행되고 각 조건문은 if문의 경우 insert\_node\_last() 함수를 통한 삽입연산 2회, 다음 노드 값을 가리키도록 link값을 대입해주는 연산 2회가 실행되며 else if, else문의 경우 각각 삽입연산 1회, 대입연산 1회가 실행된다.

하지만 insert\_node\_last()의 경우 LisHeader의 구현을 통해 while문을 돌면서 값을 찾지 않아도 되어서 O(1)시간 안에 모든 연산을 끝마칠 수 있다. 대입연산 역시 n이나 m값에 상관 없이 O(1)시간 안에 종료 가능하다.

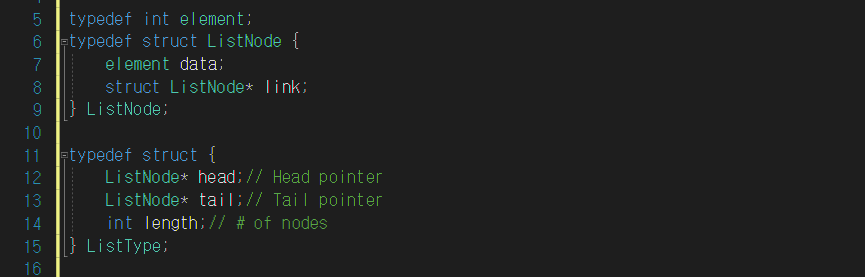
그러므로 시간 복잡도는 **O(n)**이 된다.

**실행결과**



Homework 3

**Variable**



ADT의 구조는 Homework2번과 동일

* int 형과 동일한 element type으로 정의되었다는 것과 ListHeader 라는 이름으로 정의되었던 두번째 구조체가 ListType으로 구조체 type의 이름이 바뀐 것 말고는 모두 동일한다.

**Function**

1. void error(const char\* message)

: (잘못된 명령이 들어왔을 때) 에러 메세지 출력 후 프로그램을 종료하는 함수.

1. void init(ListType\* list)

: head node를 초기화하는 함수. 처음에는 아무 노드도 연결되어 있지 않은 상태(=비어있는 상태)이므로 구조체 멤버 값인 head와 tail 포인터에 NULL을 입력, length값은 0으로 초기화한다.

1. int is\_empty(ListType\* list)

: Linked list가 비어있는지 확인하는 함수. 비어있으면 1(true), 아니면 0(false)를 리턴.

1. int get\_length(ListType\* list)

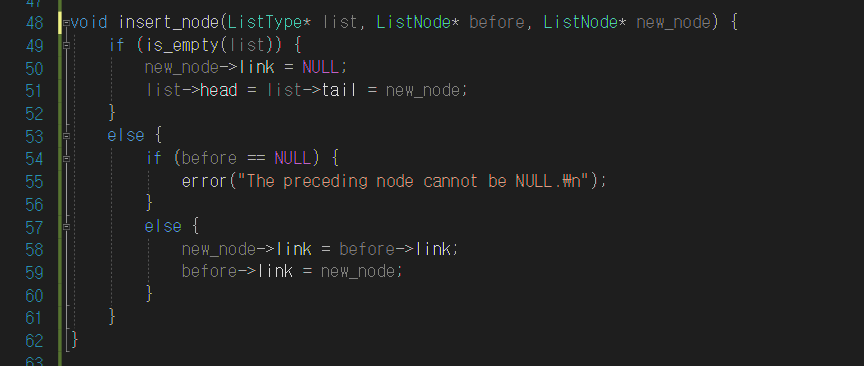
: 전체 노드의 개수를 알려주는 함수. ListType 구조체 포인터(head node역할)에 저장되어 있는 length값을 리턴.

1. ListNode\* get\_node\_at(ListType\* list, int pos)

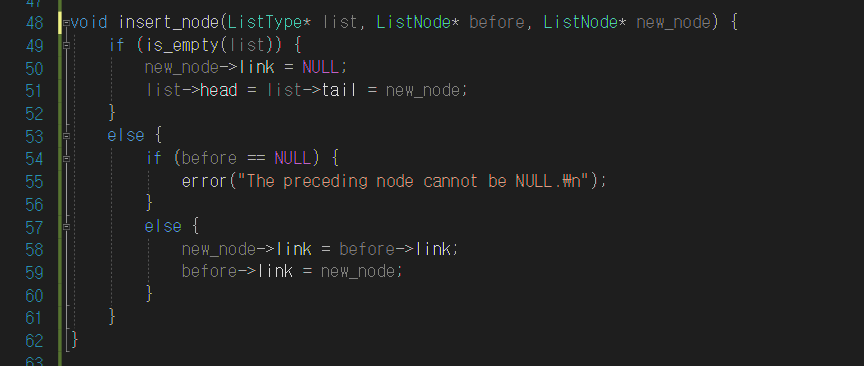
: 인자로 들어온 list의 pos번째 노드를 찾아 노드의 포인터를 리턴해주는 함수. for문을 돌면서 노드의 link 값을 통해 다음 노드로 접근하여 pos번째에 있는 노드를 찾는다.

1. void insert\_node(ListType\* list, ListNode\* before, ListNode\* new\_node)

: 삽입할 위치의 직전 노드인 before의 위치를 인자로 입력받아 list의 특정 위치에 new\_node를 삽입하는 함수.



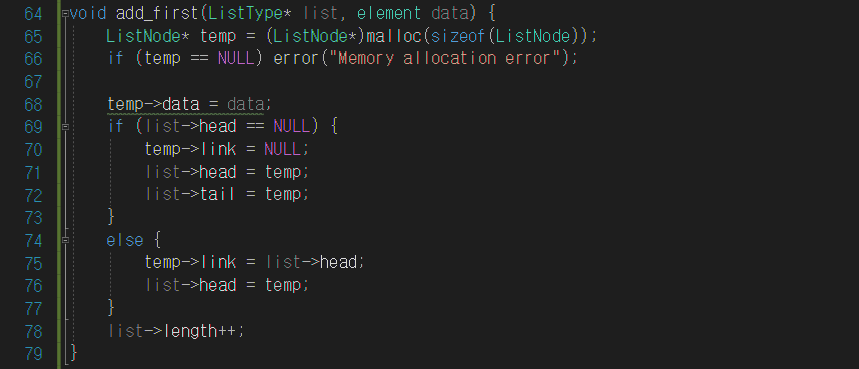
만약 리스트가 비어있다면, 삽입하고자 하는 노드의 link에는 null 값을 입력하고 (마지막 노드가되므로), list의 head와 tail은 둘 다 삽입하고자 하는 노드와 연결시켜준다.



리스트가 비어있지 않고 전달받은 before 값이 null이 아니라면, 삽입하고자 하는 노드의 link 값을 before 노드의 link값과 연결하고, before 노드의 link는 삽입하는 노드를 가리키도록 한다.

1. void add\_first(ListType\* list, element data)

: add 함수를 통해서는 linked list의 맨 앞에 노드를 삽입할 수 없다. (=position이 0일 경우) 따라서 add\_first() 함수를 따로 정의해준다.



linked list의 맨 앞에 삽입할 때는 (head node의 역할을 하는) list의 head값을 이용한다.

먼저 리스트에 삽입할 temp라는 노드를 새로 생성해준다.

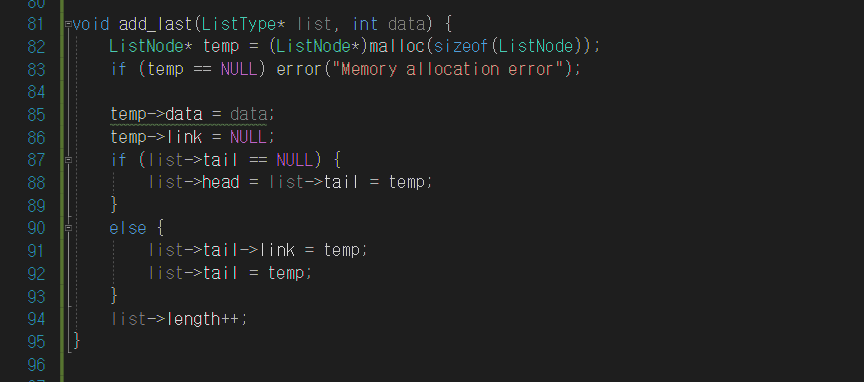
만약 리스트가 비어있다면, temp의 link는 NULL로 설정한 후 list의 tail과 head 값에 모두 temp 노드를 연결해준다.

리스트가 비어있지 않다면, list의 head값에 있던 첫번째 노드를 temp의 link값에 저장하여 temp 노드 뒤로 연결시킨다. 그 후 temp 포인터를 list의 head값에 넣는다.

노드의 삽입이 이루어졌으므로, length 값을 1 증가시킨다.

1. void add\_last(ListType\* list, int data)

: add 함수를 통해서는 linked list의 맨 뒤에 노드를 삽입할 수 없다. (=position이 length값과 동일할 경우) 따라서 add\_last() 함수를 따로 정의해준다.



linked list의 맨 뒤에 삽입할 때는 (head node의 역할을 하는) list의 tail값을 이용한다.

먼저 리스트에 삽입할 temp라는 노드를 새로 생성해준다.

저장할 data 값을 temp 노드의 data 자리에 저장하고, 맨 마지막에 삽입할 것이므로 link에는 Null을 저장한다.

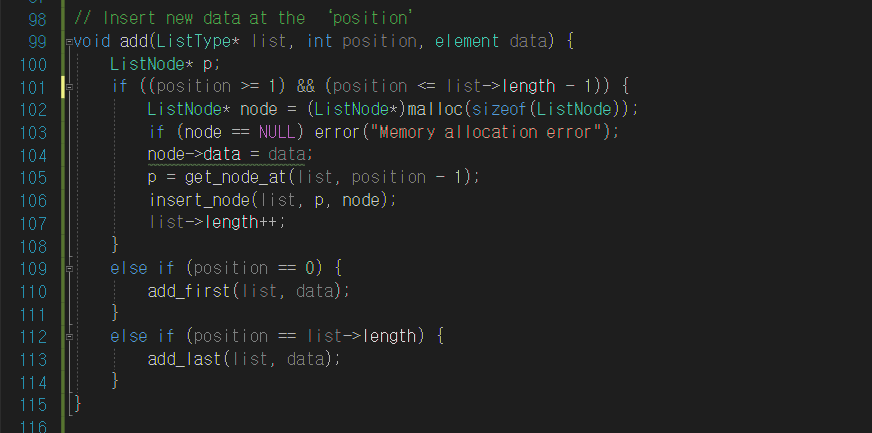
만약 리스트가 비어있다면, list의 tail과 head 값에 모두 temp 노드를 연결해준다.

리스트가 비어있지 않다면, 맨 끝에 있던 노드의 link에 Null 대신 temp를 연결하고, list의 tail은 temp를 가리키도록 한다.

노드의 삽입이 이루어졌으므로, length 값을 1 증가시킨다.

1. void add(ListType\* list, int position, element data)

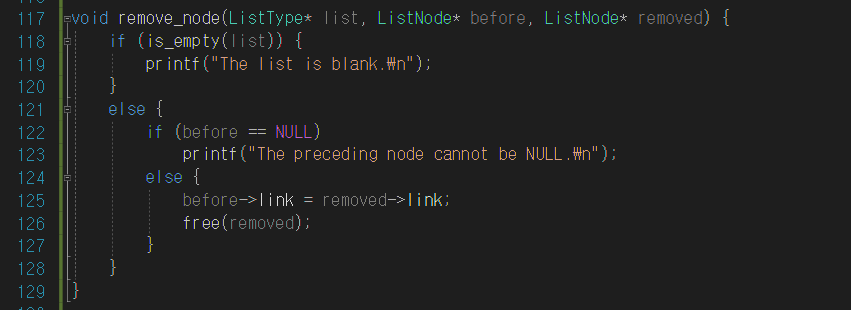
: linked list의 특정 position에 data를 삽입하는 함수. (단, position이 0과 length 값일 때는 각각 add\_first, add\_last 함수를 실행시킨다.)



원하는 위치에 삽입이 이루어지도록 insert\_node() 함수를 실행하기 위해서는 반드시 before(삽입하고자 하는 위치의 직전 노드)의 포인터를 알아야한다. 따라서 앞서 정의한 get\_node\_at 함수를 활용하여 삽입하고자 하는 위치의 직전 노드의 포인터를 찾아 인자로 보내준다. 이후 삽입을 통해 노드의 개수가 1 증가하였으므로 length 값을 1 증가시킨다.

1. void remove\_node(ListType\* list, ListNode\* before, ListNode\* removed)

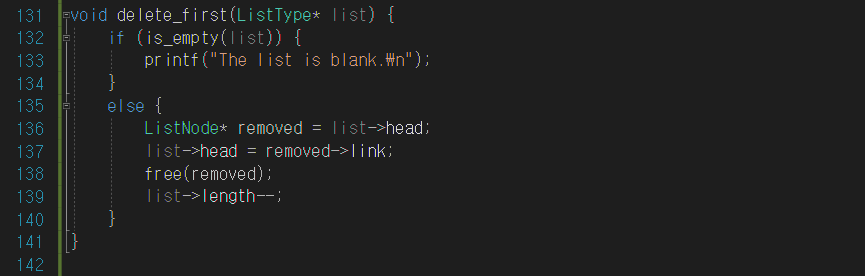
: 삭제할 위치의 직전 노드인 before의 위치를 인자로 입력받아 list의 특정 위치에 있는 노드 removed를 제거하는 함수.



리스트가 비어있지 않고 전달된 직전 노드 값이 null값이 아닐 때만 삭제를 진행하며, 지우려고 하는 노드의 link 값(지우려는 노드의 다음 노드)을 직전의 노드의 link 값과 연결해준다. 그 후, 동적으로 할당했던 메모리를 해제한다.

1. void delete\_first(ListType\* list)

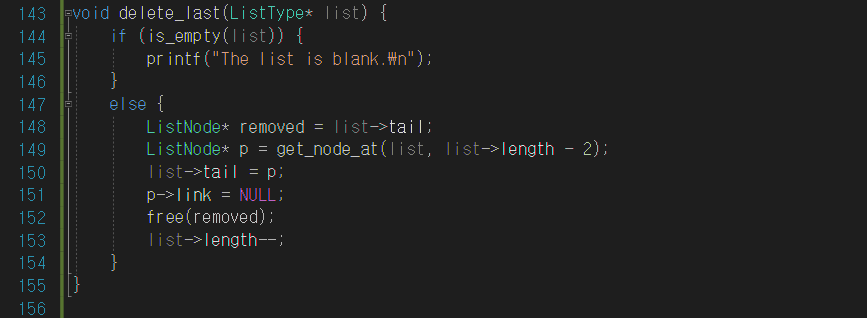
: deletefunc() 함수를 통해서는 linked list의 맨 앞에 위치한 노드를 제거할 수 없다. (=position이 0일 경우) 따라서 delete\_first() 함수를 따로 정의해준다.



리스트가 비어있지 않을 경우에만 제거하는 기능이 수행되며, list->head를 통해 첫번째 노드에 접근할 수 있다. List의 head 값이 제거하고자 하는 노드의 다음 노드(두번째 노드)를 가리키도록 바꿔주고, 동적으로 할당했던 메모리를 해제한다. 제거를 통해 노드의 개수가 1 감소하였으므로 length 값을 1 감소시킨다.

1. void delete\_last(ListType\* list)

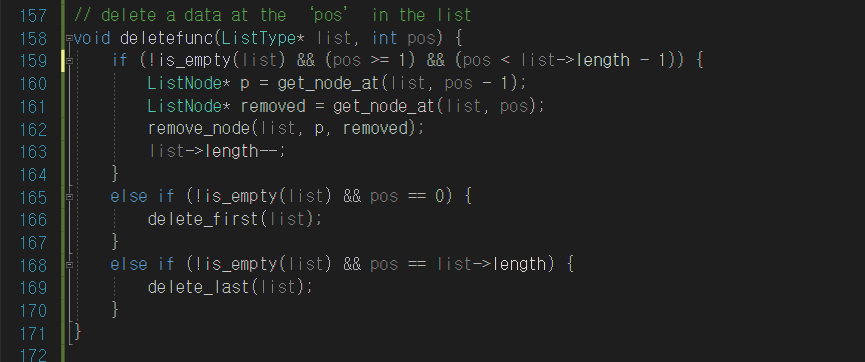
: deletefunc() 함수를 통해서는 linked list의 맨 뒤에 위치한 노드를 제거할 수 없다. (=position이 length일 경우) 따라서 delete\_last() 함수를 따로 정의해준다.



리스트가 비어있지 않을 경우에만 제거하는 기능이 수행되며, list->tail를 통해 마지막 노드에 접근할 수 있다. 또한, get\_node\_at() 함수를 통해 뒤에서 두번째에 위치해 있는 노드(p)를 찾아서 list의 tail 값이 p를 가리키도록 바꿔주고, p의 link가 가리키고 있는 값은 Null을 할당한다. 그 후 제거하고자 하는 노드의 메모리를 해제한다. 제거를 통해 노드의 개수가 1 감소하였으므로 length 값을 1 감소시킨다.

1. void deletefunc(ListType\* list, int pos)

: linked list의 특정 position에 위치한 노드를 제거하는 함수. (단, position이 0과 length 값일 때는 각각 delete\_first, delete\_last 함수를 실행시킨다.)



원하는 위치에 있는 요소가 제거되도록 remove\_node() 함수를 실행하기 위해서는 반드시 before(삽입하고자 하는 위치의 직전 노드)의 포인터를 알아야한다. 따라서 앞서 정의한 get\_node\_at 함수를 활용하여 제거하고자 하는 위치의 직전 노드의 포인터를 찾아 인자로 보내준다. 이후 제거를 통해 노드의 개수가 1 감소하였으므로 length 값을 1 감소시킨다.

1. element get\_entry(ListType\* list, int pos)

: pos번째 노드의 data 값을 리턴하는 함수.

1. void display(ListType\* list)

: linked list에 있는 모든 노드가 가진 data 값을 차례로 출력하는 함수.

1. int is\_in\_list(ListType\* list, element item)

: item 값과 일치하는 data 값을 가지는 노드가 있는지 탐색하는 함수. Linked list 안에 그러한 노드가 있으면 true, 없으면 false를 반환.

**실행결과**

