6.1 리스트 추상 데이터 타입

리스트의 소개 ex) 오늘 해야할 일: 청소, 쇼핑, 영화관람

버킷 리스트: 세계여행, 제2외국어 배우기, 마라톤 뛰기

리스트의 항목 순서∙위치를 가짐

집합과 리스트의 차이 집합 항목간의 순서∙위치가 없음

리스트 항목간의 순서∙위치가 있음

리스트의 연산 삽입연산 리스트에 새로운 항목 추가

삭제연산 리스트에서 항목 삭제

탐색연산 리스트에서 특정한 항목 찾기

리스트 ADT 객체: n개의 element형으로 구성되는 순서있는 모임

연산:

insert(list, pos, item) :: = pos위치에 요소 추가

insert\_last(list, item) :: = 맨 끝에 요소 추가

insert\_first(list, item) :: = 맨 처음에 요소 추가

delete(list, pos):: = pos위치에 요소 제거

get\_entry(list, pos) :: = pos위치의 요소 반환

get\_length(list):: = list의 길이를 구함

is\_empty(list) :: = list가 비었는지 검사

is\_full(list) ::= list가 꽉 찼는지 검사

print\_list(list) ::= 리스트의 모든 요소 표시

리스트의 구현 배열과 연결리스트를 통해 구현 가능

배열 간단, 속도빠름 but 크기가 고정됨

연결리스트 포인터를 이용해 연결리스트를 만듬

장점 크기 제한 x, 중간에 삽입 쉬움

단점 구현 복잡, 시간 많이 걸림

Quiz 1. ListType li; insert\_last(li, 10); insert\_last(li, 20); delete(li, 0); print\_list(li);

6.2 배열로 구현된 리스트

리스트의 순차적 표현 배열을 이용해 리스트를 구현하면 순차적인 메모리 공간이 할당됨

리스트의 정의 배열과 항목개수를 구조체로 묶음

typedef int element;

typedef struct {

element array[MAX\_LIST\_SIZE];

int size;

} ArrayListType;

기초연산 리스트의 연산들을 함수로 구현

모든 연산들은 구조체 포인터를 받음 🡪 원본 구조체를 변환해야 하기 때문

//오류처리함수

void error(char\* message) {

fprintf(stderr, "%s\n", message);

exit(1);

}

//리스트 초기화 함수

void init(ArrayListType\* L) {

L->size = 0;

}

//리스트가 비어있으면 1을 반환, 그렇지 않으면 0을 반환

int is\_empty(ArrayListType\* L) {

return L->size == 0;

}

//리스트가 가득차있으면 1을 반환, 그렇지 않으면 0을 반환

int is\_full(ArrayListType\* L) {

return L->size == MAX\_LIST\_SIZE;

}

//요소가져오기

element get\_entry(ArrayListType\* L, int pos) {

if (pos<0 || pos > L->size) error("위치오류");

return L->array[pos];

}

//요소들 출력

void print\_list(ArrayListType\* L) {

int i;

for (i = 0; i < L->size; i++) printf("%d->", L->array[i]);

printf("\n");

}

항목 추가 연산 리스트 항목 추가

//리스트의 맨 끝에 항목을 추가

void insert\_last(ArrayListType\* L, element item) {

if (L->size >= MAX\_LIST\_SIZE) error("리스트 오버플로우");

L->array[L->size++] = item;

}

//리스트 중간에 항목 삽입

void insert(ArrayListType\* L, int pos, element item) {

if (!is\_full(L) && (pos >= 0) && (pos <= L->size)) {

for (int i = (L->size - 1); i >= pos; i--)

L->array[i + 1] = L->array[i];

L->array[pos] = item;

L->size++;

}

}

항목 삭제 연산 pos위치의 항목을 삭제하는 delete(list, pos)

삭제한 후, array[pos+1]부터 array[size-1]까지 한 칸씩 앞으로 이동해야 함

element delete(ArrayListType\* L, int pos) {

element item;

if (pos < 0 || pos >= L->size) error("위치 오류");

item = L->array[pos];

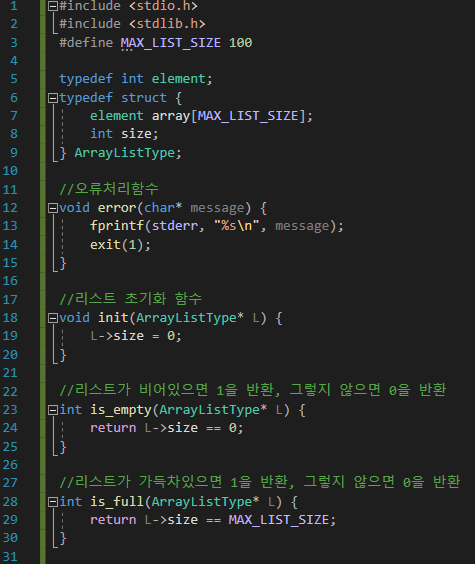
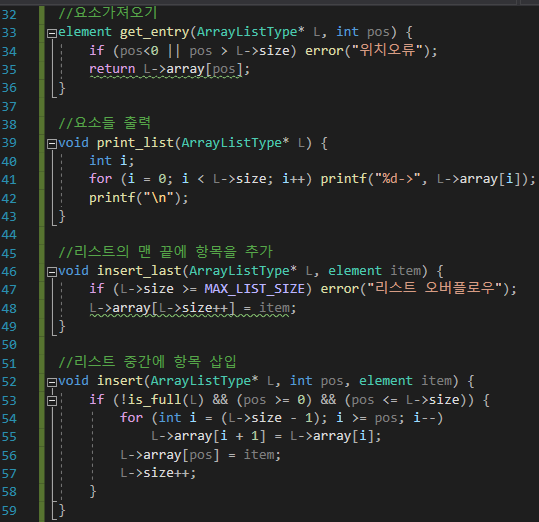
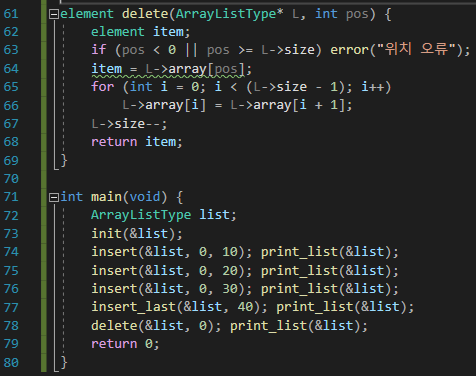
for (int i = 0; i < (L->size - 1); i++)

L->array[i] = L->array[i + 1];

L->size--;

return item;

}

테스트 프로그램 프로그램6.1   

실행시간 분석 배열로 구현한 리스트의 시간 복잡도 get\_entry() 인덱스를 통한 요소 접근 🡪 O(1)

삽입∙삭제연산 다른 항목들을 이동하는 경우가 많음 🡪 O(n)

but 리스트에 맨끝에 삽입하는 경우 🡪 O(n)

도전문제 1. void insert\_first(ArrayListType\* L, element item) {

insert(L, 0, item);

}

2. int main(void) {

ArrayListType\* list = malloc(sizeof(ArrayListType));

init(list);

insert(list, 0, 10); print\_list(list);

insert(list, 0, 20); print\_list(list);

insert(list, 0, 30); print\_list(list);

free(list);

return 0;

}

Quiz 1. void clear(ArrayListType \*L) {

init(L);

}

element replace(ArrayListType\* L, int pos, element item) {

element res;

res = L->array[pos];

L->array[pos] = item;

return res;

}

int get\_length(ArrayListType\* L) {

return L->size;

}

6.3 연결리스트

연결된 표현 포인터를 사용해 데이터를 연결함

여러 자료구조를 구현하는 데도 많이 쓰임

상자를 연결하는 줄 포인터

상자 데이터

장점 동적으로 크기가 변할 수 있고 삭제∙삽입 시 데이터를 이동할 필요가 없음

단점 구현이 어렵고, 오류가 나기 쉬움

연결리스트의 구조 노드 데이터가 담긴 상자

데이터필드 저장하고 싶은 데이터가 들어감

링크필드 다른 노드를 가리키는 포인터가 저장됨

헤드포인터 첫번째 노드를 가리키고 있는 변수

마지막 노드의 링크필드 NULL

연결리스트의 종류 단순연결리스트 하나의 방향으로만 연결되어 있는 연결리스트

= 체인

마지막 노드의 링크는 NULL 값을 가짐

원형연결리스트 마지막 노드의 링크가 첫번째 노드를 가리킴

이중연결리스트 각 노드마다 2개의 링크 존재

하나의 링크는 앞에 있는 노드를 가리키고 또 하나의 링크는 뒤에 있는 노드를 가리킴

6.4 단순 연결 리스트

노드 하나의 링크필드를 가짐

마지막 링크 필드의 값은 NULL

노드 정의 🡨 자기참조구조체

노드 생성 🡨 malloc() 호출

노드 삭제 🡨 free() 호출

노드의 정의 자기참조구조체를 이용해 정의

자기참조구조체 자기 자신을 참조하는 포인터를 포함하는 구조체

typedef int element;

typedef struct ListNode {

element data;

struct ListNode\* link;

} ListNode;

공백리스트의 생성 헤드포인터 선언

ListNode\* head = NULL;

공백검사 헤드포인터가 NULL인지 검사

노드의 생성 필요할 때마다 동적메모리 할당을 통해 노드를 동적으로 생성

malloc()함수를 이용해 노드의 크기만큼 동적메모리 할당 🡪 동적메모리가 하나의 노드가 됨

head = (ListNode\*)malloc(sizeof(ListNode));

head->data = 10;

head->link = NULL;

노드의 연결 동일한 방식으로 두번째 노드를 동적으로 생성

ListNode\* p;

p = (ListNode\*)malloc(sizeof(ListNode));

p->data = 20;

p->link = NULL;

그 다음 첫번째 노드의링크가 두번째 노드를 가리키게 함

head->link = p;

Quiz 1. 헤드포인터

2. 링크

3. 삽입∙삭제 연산이 쉬움, 동적으로 개수 조절가능

6.5 단순 연결 리스트의 연산 구현

insert\_first() 리스트의 시작 부분에 항목을 삽입하는 함수

insert() 리스트의 중간 부분에 항목을 삽입하는 함수

delete\_first() 리스트의 첫번째 항목을 삭제하는 함수

delete() 리스트의 중간항목을 삭제하는 함수

print\_list() 리스트를 방문하여 모든 항목을 출력하는 함수1

단순 연결 리스트 정의 단순연결리스트는 원칙적으로 헤드포인터만 있으면 됨

삽입연산 insert\_first() 리스트의 처음에 새로운 노드를 추가하는 경우

새로운 노드를 생성하고 새로운 노드의 link에 현재의 head의 link값을 저장 후, head의 link를 현재의 노드의 주소로 변경

알고리즘6.1 insert\_first(head, value):

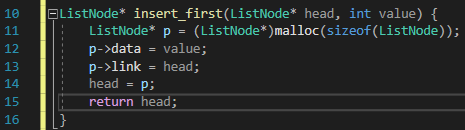
1. p <-- malloc() //동적메모리 할당을 통하여 새로운 노드 p를 생성한다

2. p->data <-- value //p->data에 value를 저장한다

3. p->link <-- head //p->link를 현재의 head값으로 변경한다

4. head <-- p //head를 p값으로 변경

5. return head //return head

프로그램6.1 

삽입연산 insert() 연결리스트의 중간에 새로운 노드를 추가함

이때는 반드시 삽입되는 위치의 선행노드를 알아야 삽입이 가능함

알고리즘6.2 insert(head, pre, value):

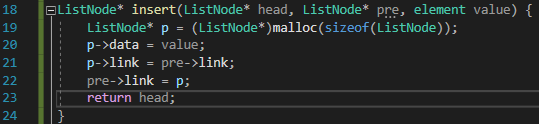
1. p <-- malloc() //새로운 노드를 생성해 변수 p로 가리킴

2. p->data <-- value //p의 데이터필드에 value를 저장함

3. p->link <-- pre->link //p의 링크필드가 pre의 링크필드를 저장함

4. pre->link <--p //pre의 링크필드가 p를 가리키게 함

5. return head //변경된 헤드포인터 반환

프로그램6.3 

삭제연산 delete\_first() 첫번째 노드 삭제

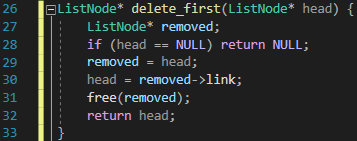
알고리즘6.3 delete\_first(head):

1. remove <-- head //헤드포인터를 removed에 복사

2. head <-- head->link //헤드포인터를 head->link(다음 노드)로 변경

3. free(removed) //removed가 가리키는 동적메모리 반환

4. return head //변경된 헤드 포인터 반환

프로그램6.4 

삭제연산 delete() 리스트의 중간에서 삭제하는 알고리즘

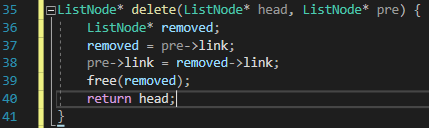
알고리즘6.4 delete(head, pre):

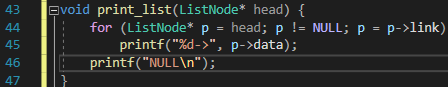
1. removed <-- pre->link //삭제할 노드(pre의 다음노드)를 removed에 넣음

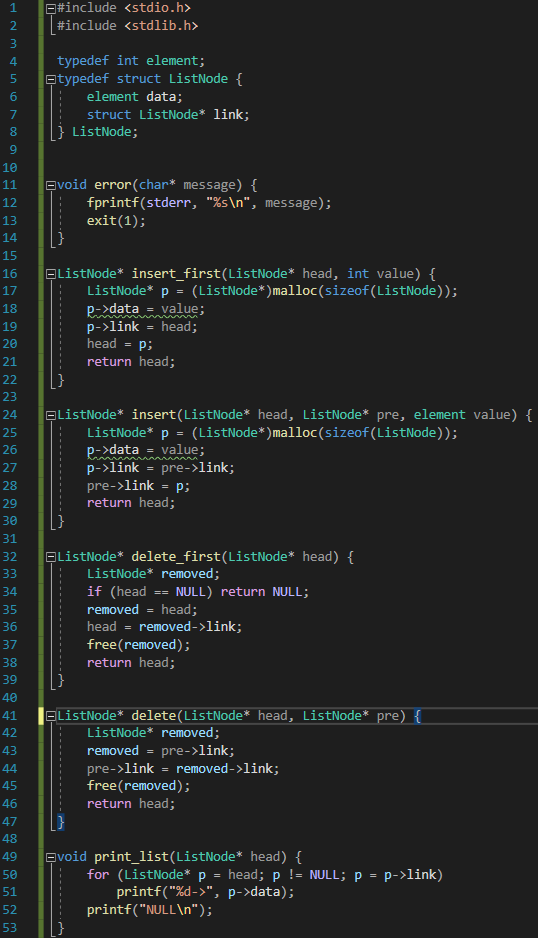
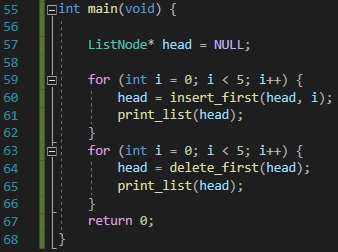
2. pre->link = removed->link //pre의 다음노드를 삭제할노드의 다음노드로 변경

3. free(removed) //removed 삭제

4. return head //head포인터 반환

프로그램6.4 

print\_list () 프로그램6.6 

전체 테스트 프로그램 프로그램6.7  

도전문제 ListNode\* get\_entry(ListNode\* L, int index) {

ListNode\* n= L;

for (; index >=0; index--) {

n = n->link;

}

return n;

}

Quiz 1. 삭제하려는 노드의 이전 노드

2. int get\_length(ListNode\* L) {

int length = 1;

while (L->link != NULL) {

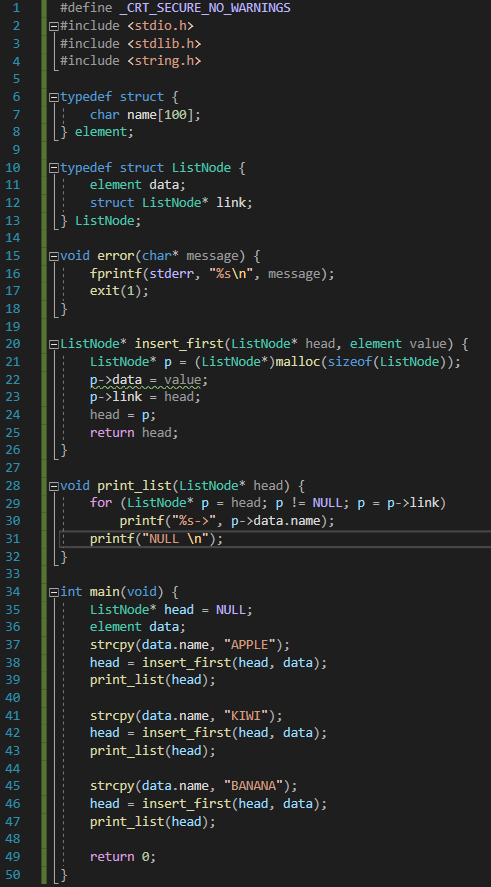
L = L->link;

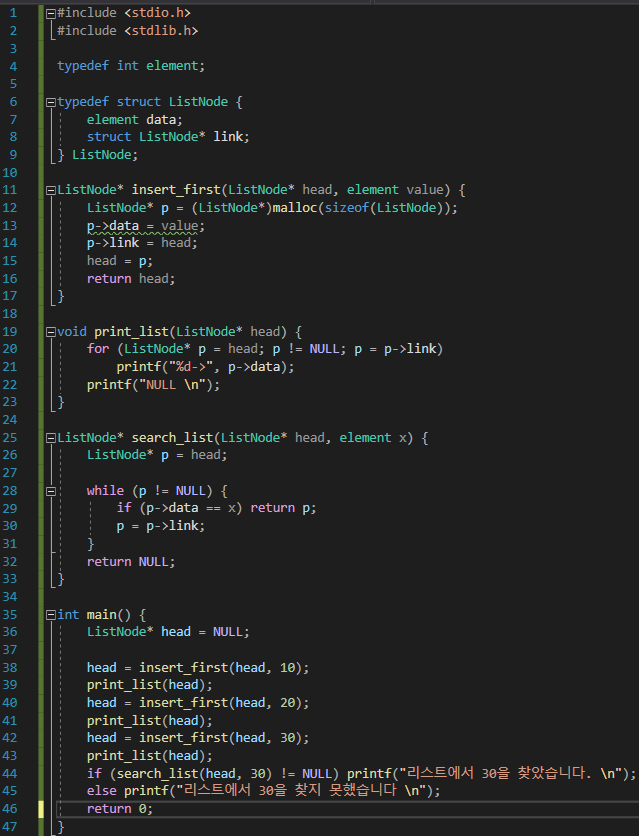
length++;

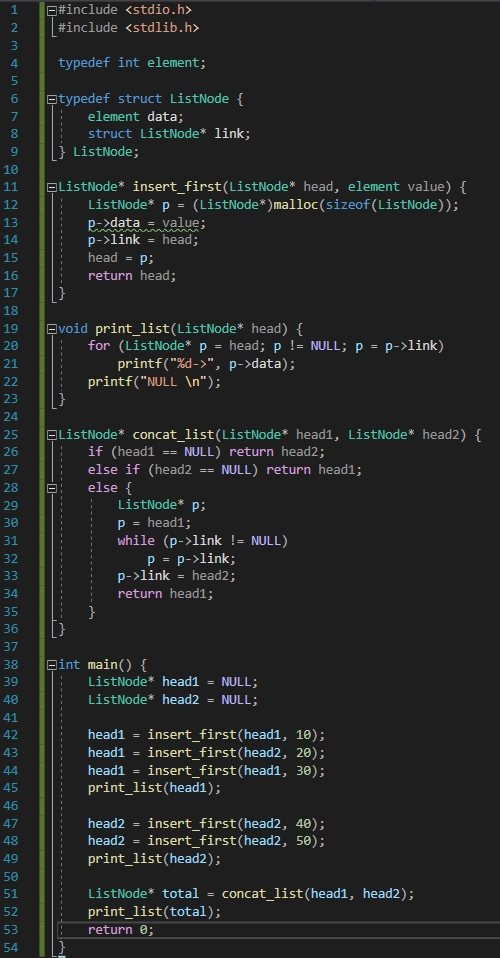
}

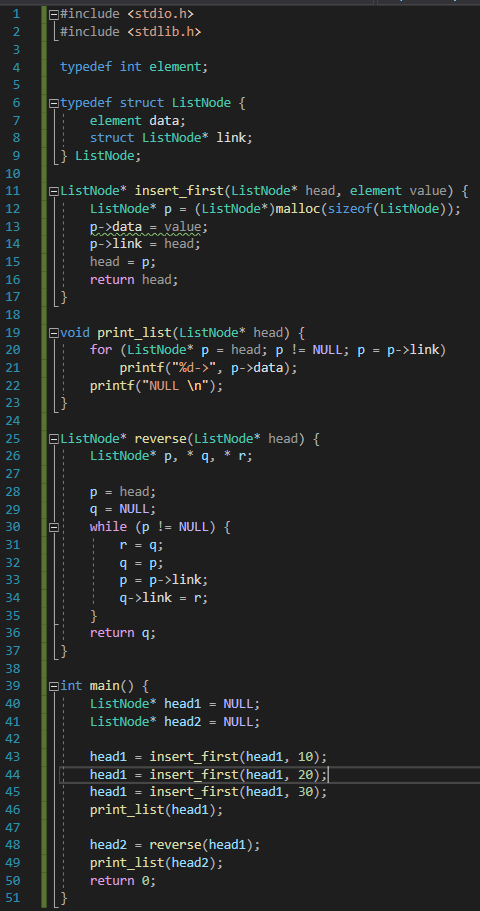
return length;

}

Lab1, 단어들을 저장하는 연결 리스트를 만들어보자. 

Lab2, 특정한 값을 탐색하는 함수를 작성해보자. 

Lab3, 두개의 리스트를 하나로 합치는 함수 작성 

Lab4, 리스트를 역순으로 만드는 연산 

6.6 연결 리스트의 응용: 다항식

연결리스트로 다항식 표현

각 항을 노드로 표현 typedef struct ListNode {

int coef;

int expon;

struct ListNode\* link;

} ListNode;

각 다항식은 다항식의 첫번째 항을 가리키는 포인터로 표현

항의 지수에 따라 3가지로 나누어 처리 1. p.expon == q.expon 계수를 더해 0이 아니면, 새로운 식을 만들어 다항식 c에 추가

2. p.expon < q.expon q가 지시하는 항을 새로운 항으로 복사해, 다항식 c에 추가

3. p.expon > q.expon p가 지시하는 항을 새로운 항으로 복사해, 다항식 c에 추가

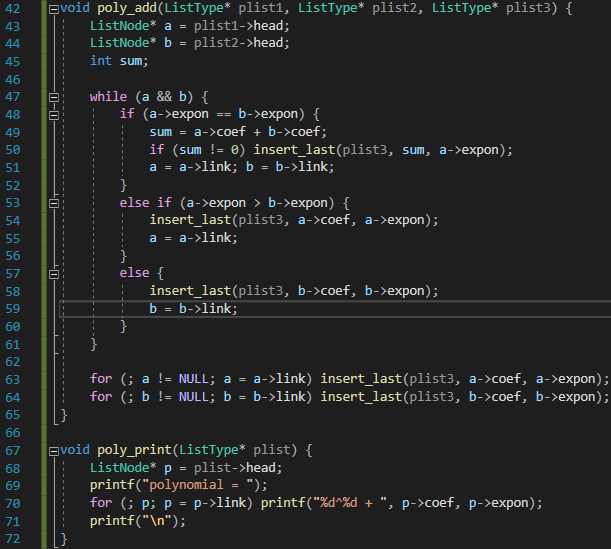
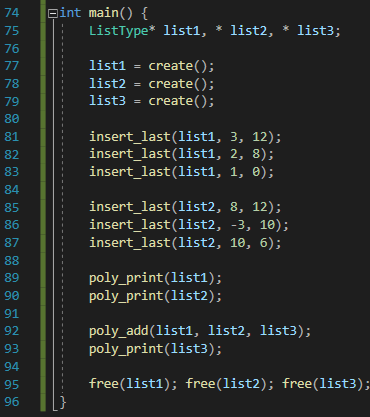
위 과정을 p, q가 NULL이 될 때까지 되풀이

q의 노드의 지수가 p의 노드의 지수보다 높으면, c에 q의 노드와 동일한 노드를 만들고 q를 전진시킴

헤더노드 하나의 연결 리스트를 두개의 포인터 head와 tail로 표현

길이, 헤드, 노드

연결리스트를 다루는 것이 편해짐 ex) 리스트 맨 마지막에 요소를 추가하는 경우

프로그램6.9   

연습문제

1. 2

2. 1

3. 3

4. c

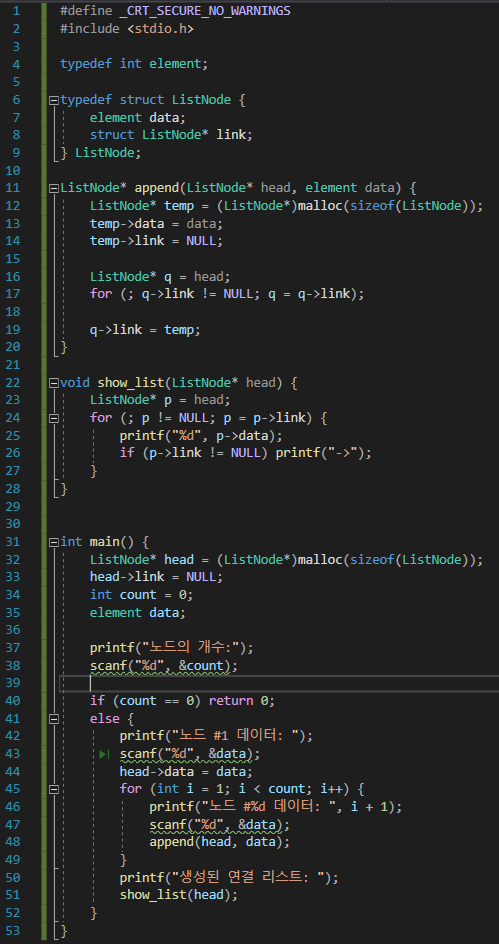
5. p=p->link

6. q=p

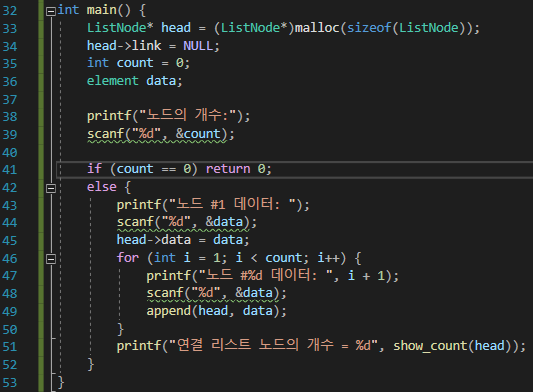
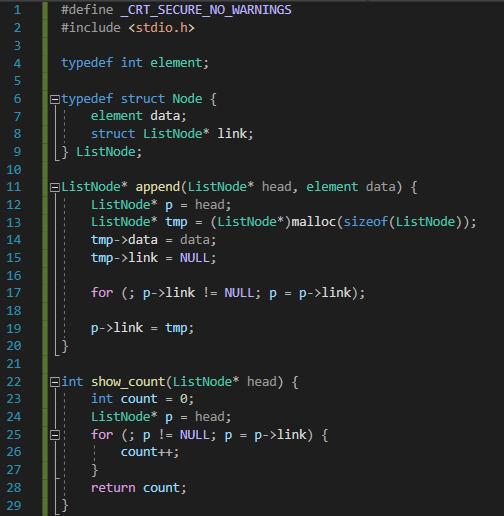
7. D

8. 4

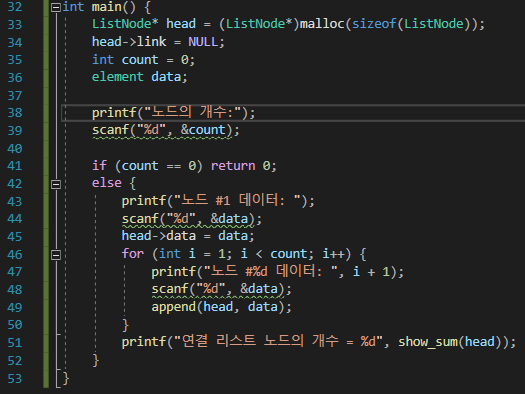
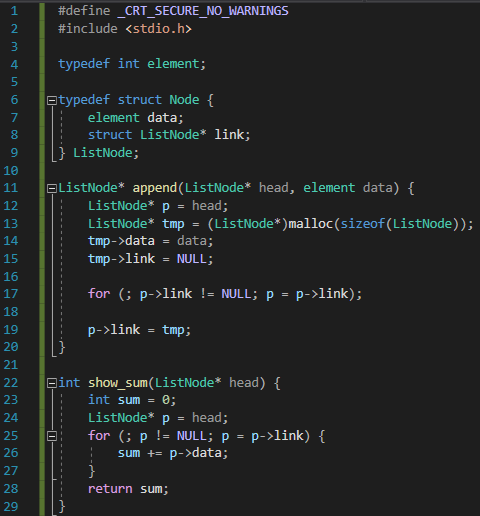
9.



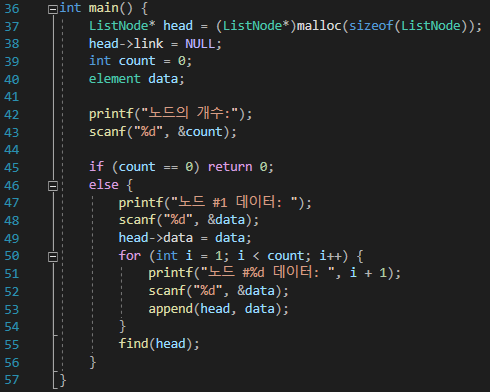
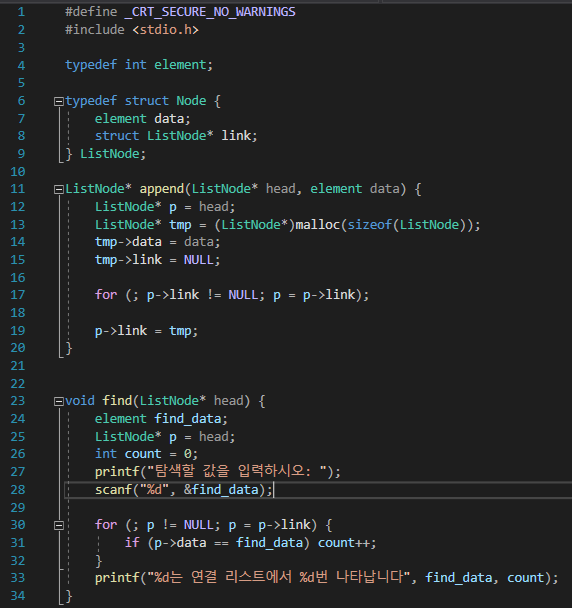
10.



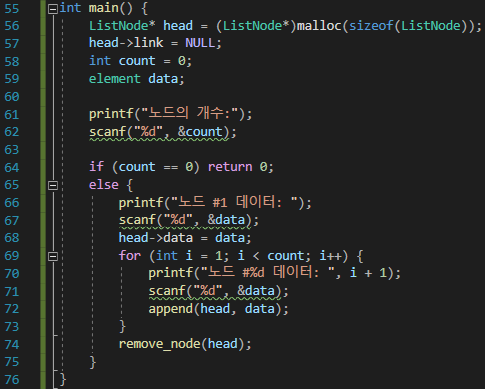
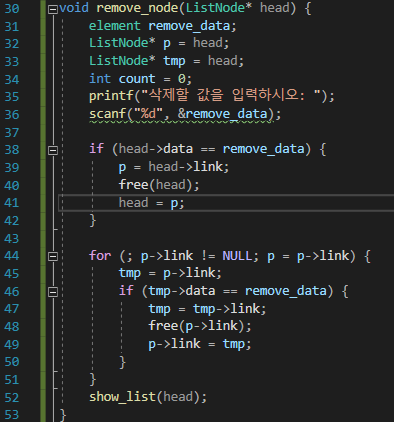
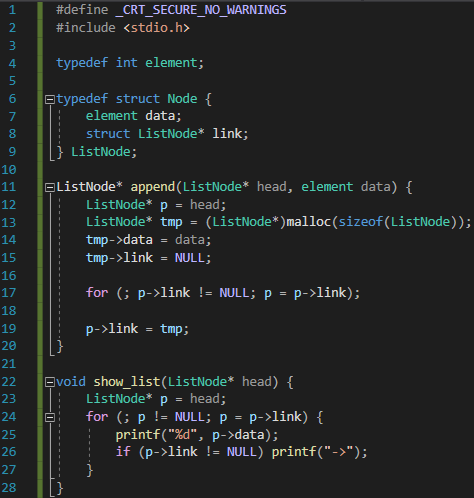
11.



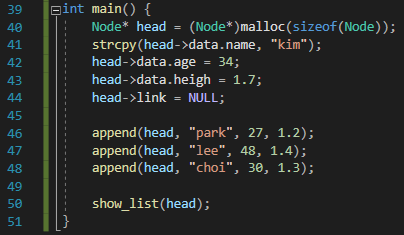
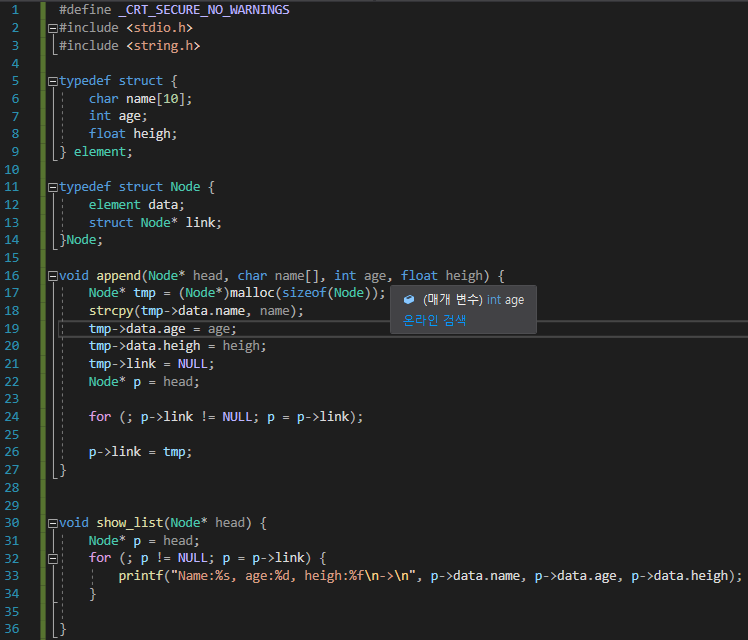
12.



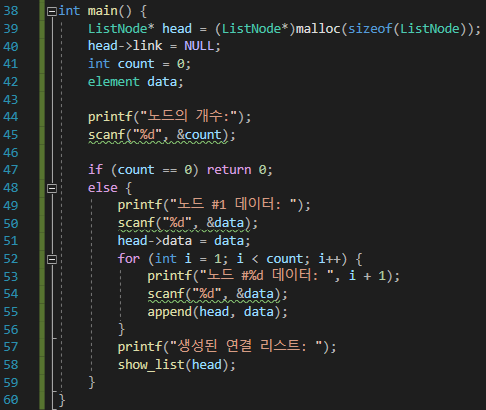
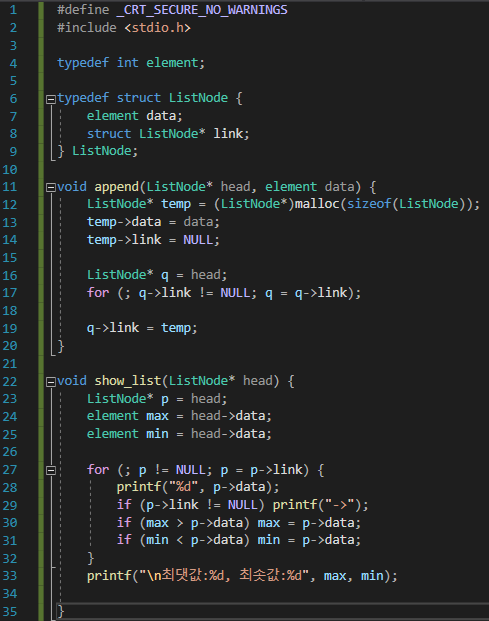
13.



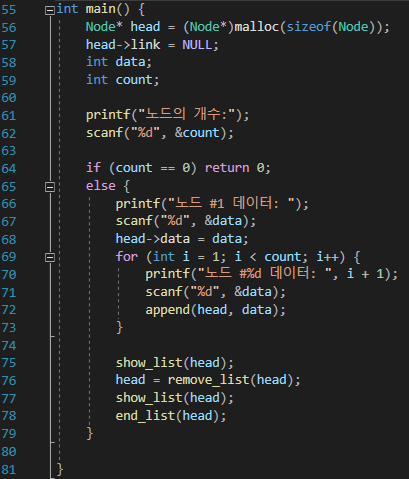
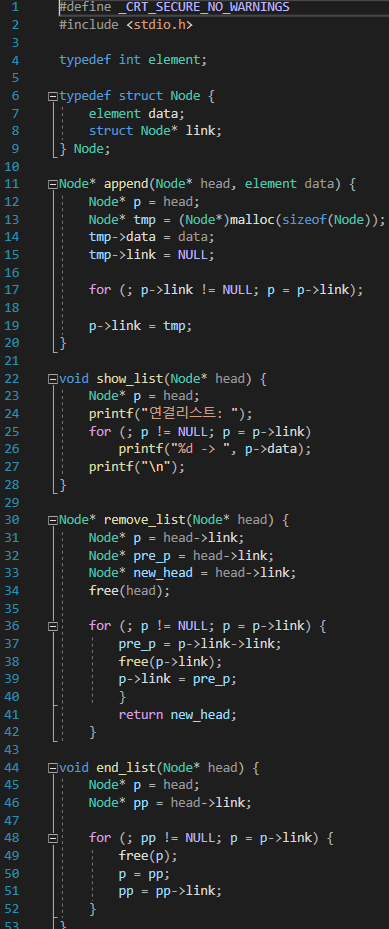
14.



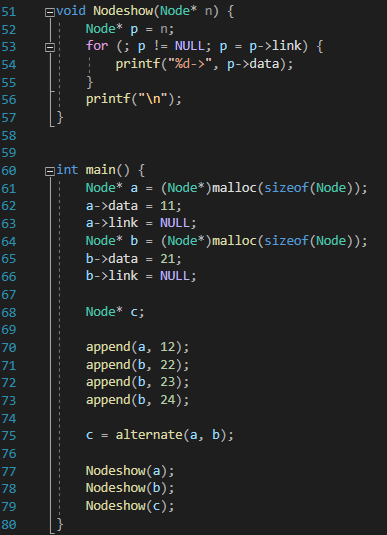
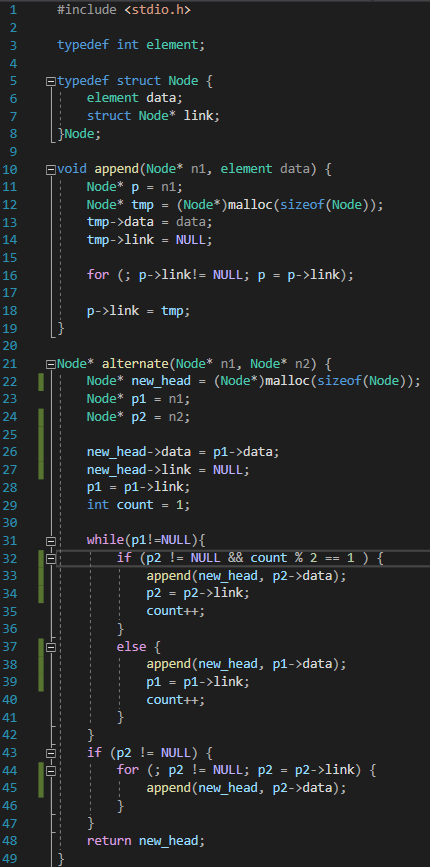
15.



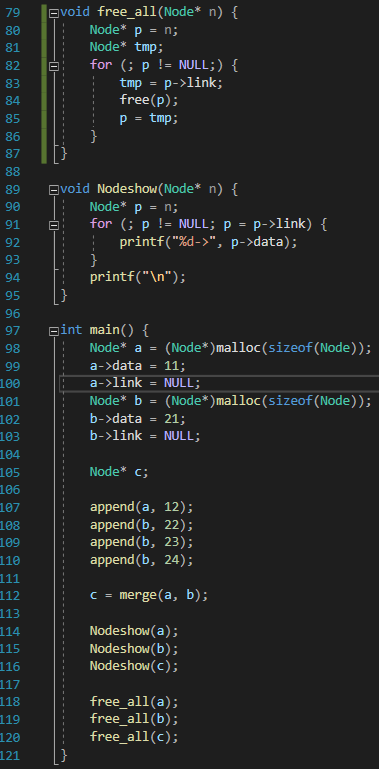
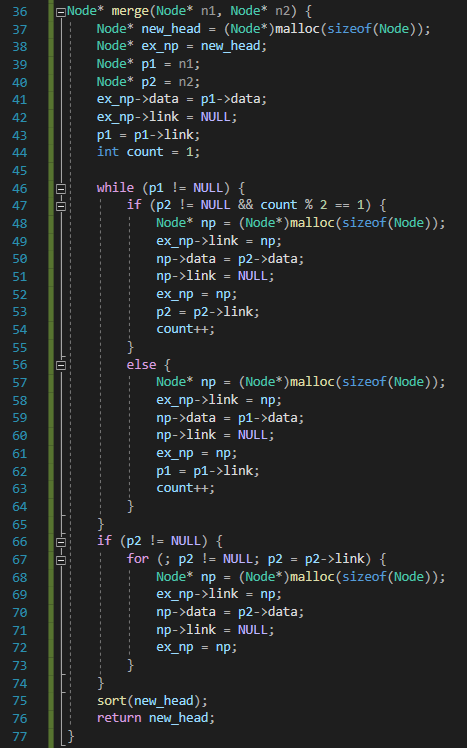
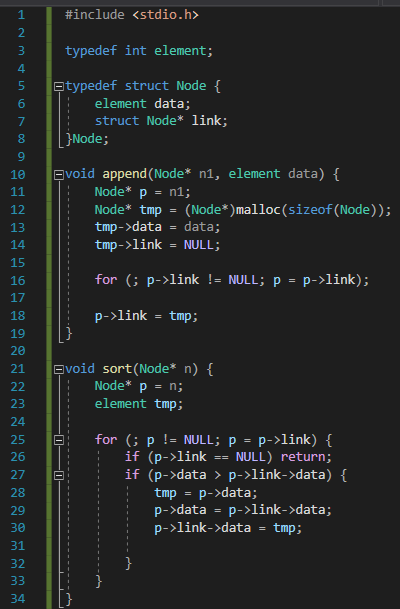
16.



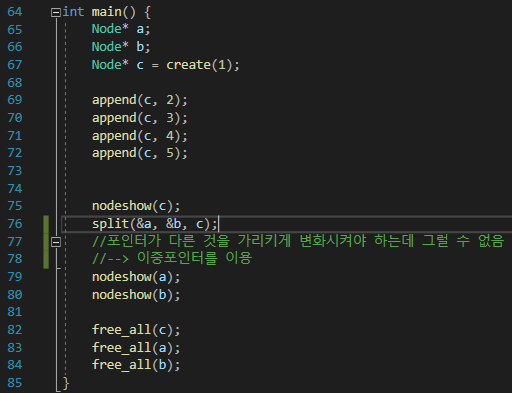
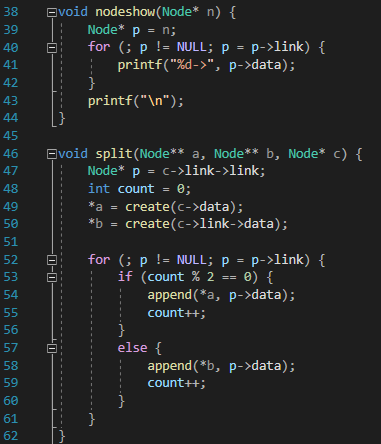
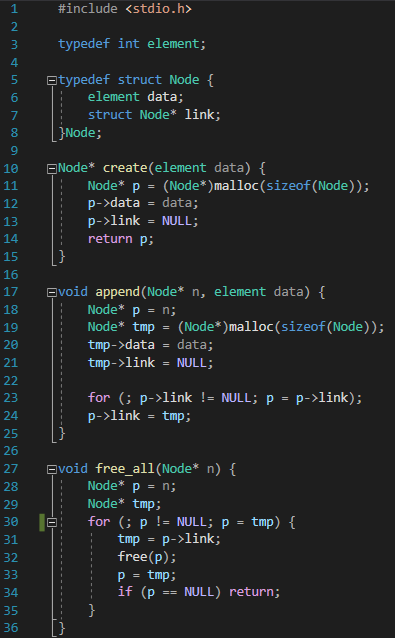
17.



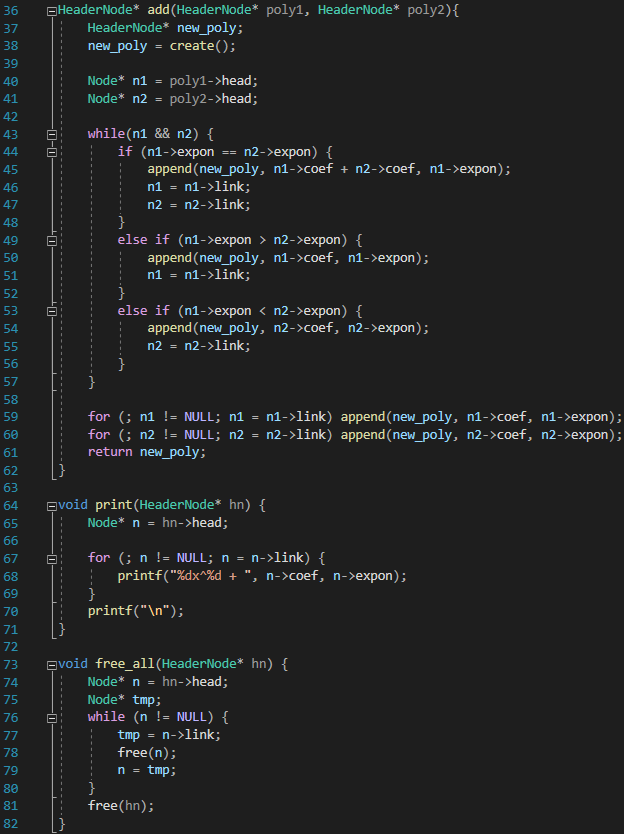
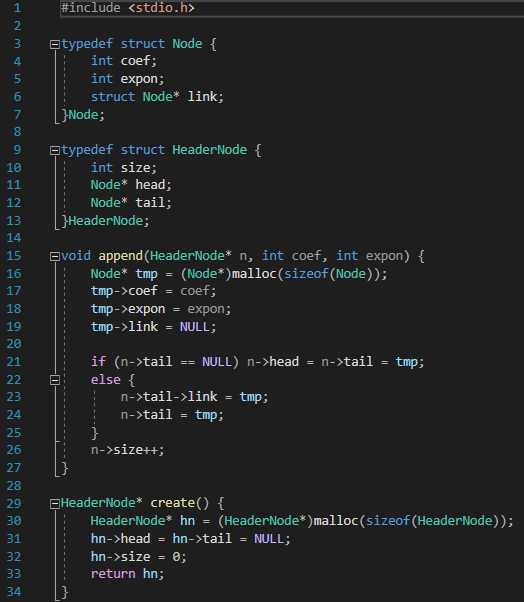
18.

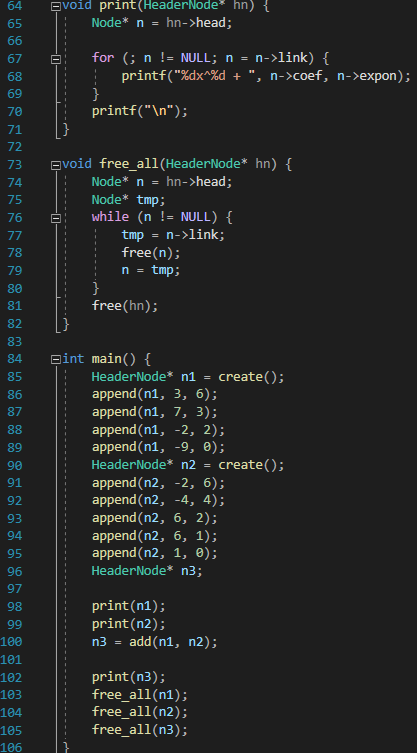


19.

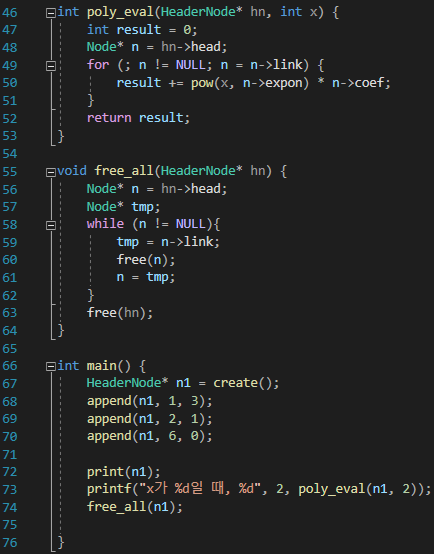
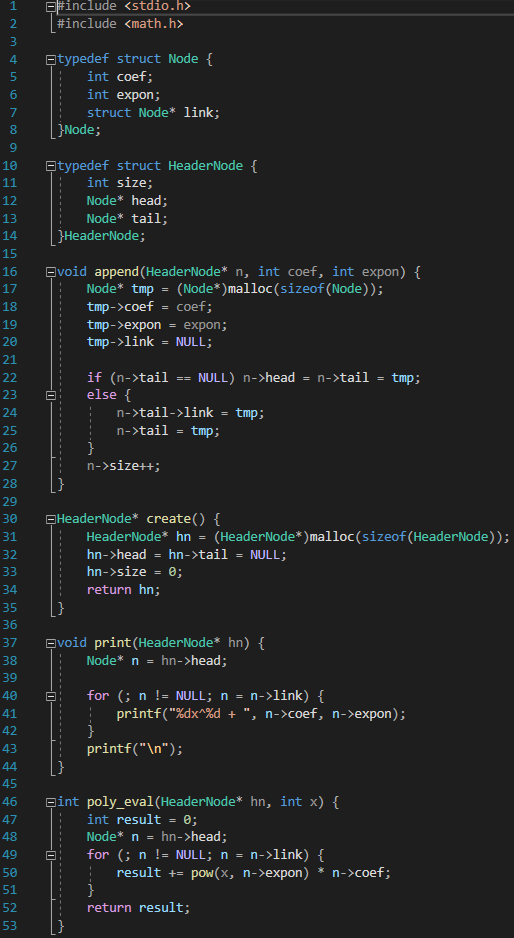


20.

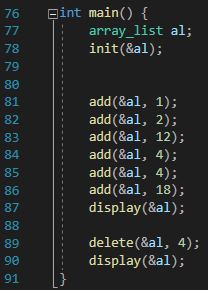
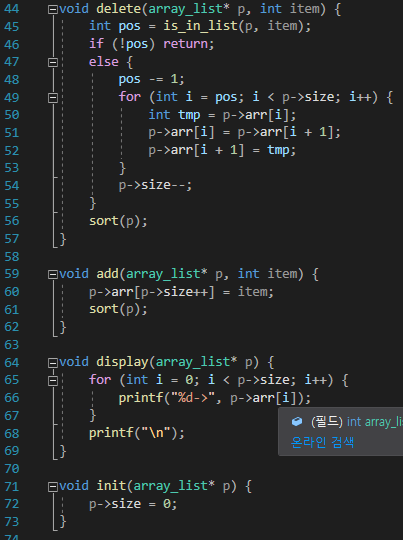
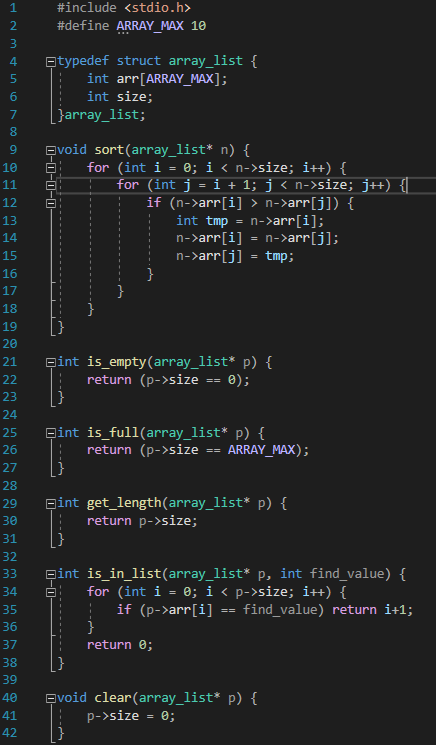




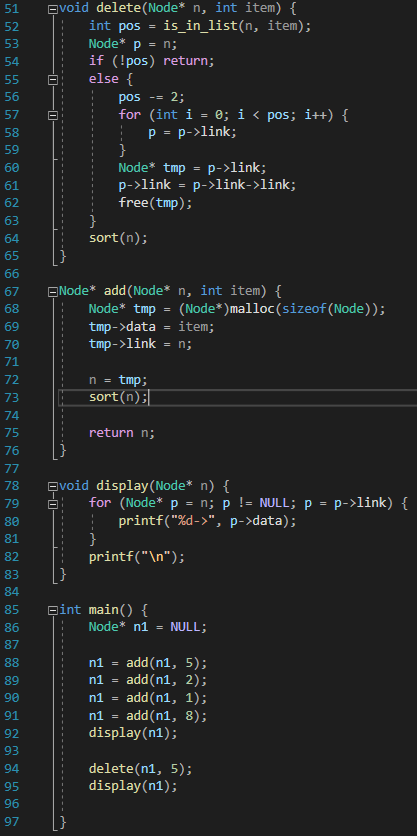
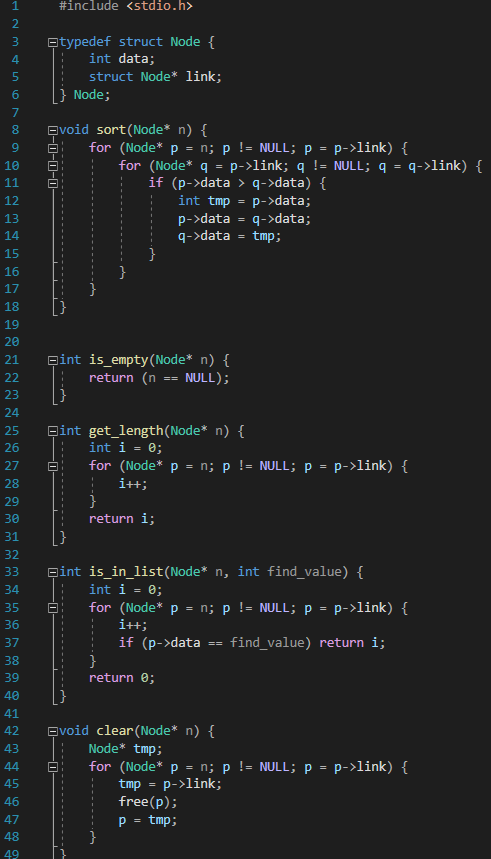
21.



22.



23.



24.

