데이터구조 레포트 2

학년	2학년	학번	20211379
이름	장원영	제출일	2024.04.29

일체의 부정 행위 없이 스스로의 노력으로 작성한 레포트임을 확인합니다.

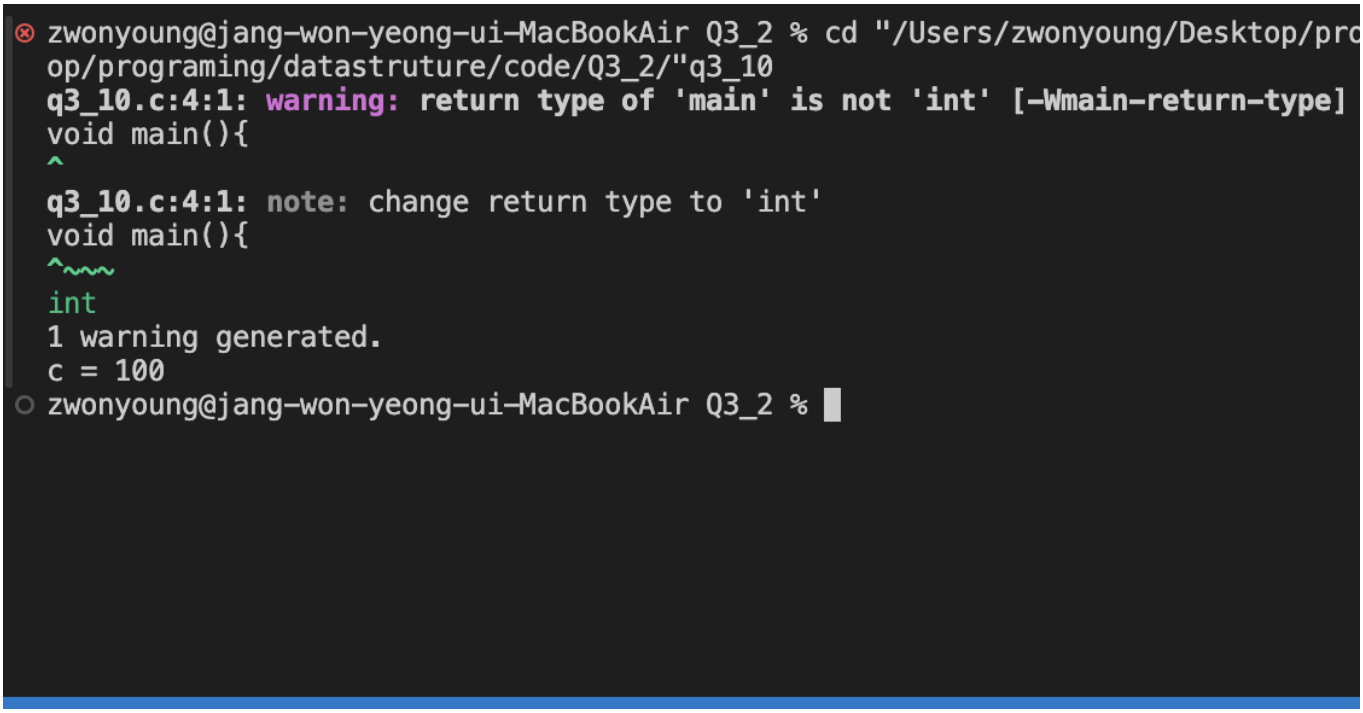
문제 3

문제 3.(10)

3.(10) 코드 및 주석

```
#include <stdio.h> // 표준 입출력 기능을 제공하는 헤더파일
#include <memory.h>// 메모리 관련 기능을 제공하는 헤더파일 하지만 malloc함수가 정의되어 있지 않음
#incldue <stdlib.h> // malloc함수를 사용하기 위한 헤더파일

void main(){
        char *pc; // 포인터 변수 pc 선언
        // 동적으로 1바이트 크기의 메모리를 힙 영역에 생성하고
        // malloc함수는 생성한 공간의 시작 주소인 void 포인터형 데이터를 반환하므로
        // char포인터형으로 타입캐스팅하여 준 후
        // 그 타입캐스팅 한 데이터를 char형 포인터 변수 pc에 넘겨 주었다.
        pc = (char*)malloc(1);

        // 포인터를 할당해 주었으므로 그 포인터를 통해 해당 주소 위치의 값을 100으로 변경
        *pc = 100;

        // 이후 printf를 통해 pc가 가리키는 주소의 값을 출력해 주었다.
        printf("c = %d\n", *pc);
        free(pc);
}
```

3.(10) 출력 화면 및 설명



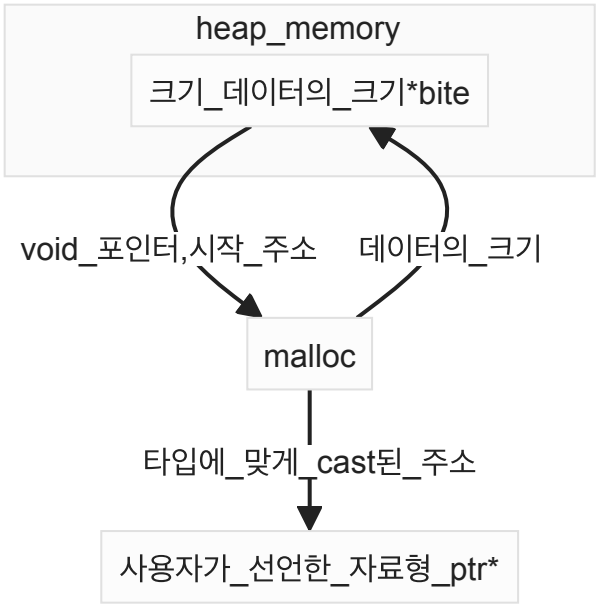
malloc 함수의 원형을 살펴보면 다음과 같습니다.

```
void *malloc(size_t size);
```

malloc 함수는 기본적으로 파라미터로 바이트 단위의 '크기'를 받습니다. 예를 들어 단순히 100을 넘겨주면 100바이트만큼의 메모리 크기를 '힙 영역에서' '프로그램 실행 도중'에 동적으로 할당합니다.

그리고 그 동적으로 할당한 메모리 공간의 처음 시작 주소를 return하는데 이때의 return하는 값은 void 포인터형입니다.

이때 void포인터형을 반환하는 이유는 이 void 포인터 형을 사용자가 사용하고자 하는 자료형에 맞게 변환하여 사용할 수 있도록 하기 위해서 void 포인터 형을 반환합니다.



이때 '크기데이터의크기' 'bite'는 heap 메모리 공간에서 malloc함수가 알아서 할당해 준다.

이 점을 숙지하고 다시 넘어가보면 동적으로 할당된 메모리의 처음 시작주소를 받을 char형 포인터 pc를 선언해주고,

이후 동적 메모리 할당을 하는데 malloc 함수에 인수로 1을 넘겨주고 그러면 이 malloc함수는 입력받은 1 *bite 크기만큼의 메모리를 heap메모리 공간에 임의로 만들어줍니다.

그리고 malloc함수는 그 메모리 공간의 시작 주소를 void포인터형으로 변환하는데 이때 void포인터형 그대로 사용할 수 없으므로 사용자가 사용하고자 하는 데이터형의 포인터로 바꾸어 주어야 합니다.

pc가 char형 포인터이므로 이에 맞게 malloc함수의 반환값인 포인터로 char형 포인터로 타입캐스팅 해 준 모습입니다.

이후 pc는 주소를 받은 포인터이므로 포인터 문법 *를 사용하여 동적으로 할당된 메모리를 자유롭게 다룰 수 있는데, 동적으로 할당된 메모리 공간의 값을 100으로 바꾸어주었고,

이후 printf문을 통해 pc가 가리키는 값을 정수형으로 읽어들여 출력하여 100을 성공적으로 출력한 모습입니다.

실제 '값' 자체는 이진수 1100100으로 저장되는데 printf를 통해 어떤 식으로 읽을지 정해 줄 수 있습니다.

예를 들면 지금 pc가 가리키는 '값' 자체는 %c로 해석한 아스키 코드로는 'd'가 되고 %d로 해석한 10진수 정수로는 100입니다.

동적으로 1바이트를 선언하였으므로 8비트의 공간만큼의 수를 표현할 수 있는데 256가지의 수를 표현이 가능합니다.

char형은 1바이트의 크기를 가지고 있습니다. 패리티 체크용 비트 1개 나머지 비트 7개 해서 총 128가지의 문자를 표현할 수 있고 그 각각의 숫자가 어떤 문자인지 할당이 되어 있습니다.

int형은 4바이트지만 여기서 동적으로 할당된 메모리의 크기는 1바이트이므로 부호 있는 10진수로 해석하는 %d로 해석이 되긴 합니다. 단순 비트이기 때문입니다. +127 -128까지 읽어들일 수는 있습니다.(맨 앞 비트는 패리티 체크 비트이므로 이 점을 인지하여 주의 깊게 보아야 할 필요는 있습니다.)

이 점들을 모두 고려하였을때 100이 정상적으로 출력되는 것을 알 수 있습니다.

또한 마지막으로 사용을 완료한 동적 메모리는 반드시 free 함수를 통해 시스템에 반환해 주어야 하는데, free 함수는 이 함수에 들어간 포인터가 가리키는 동적 메모리를 해제시켜 주는 작업을 진행합니다. 이렇게 해주지 않으면 실제로 프로그램상에서 사용되지 않는 동적 메모리지만 프로그램 실행 도중 계속 공간을 차지하게 됩니다. 이렇게 남아있는 쓰레기 메모리를 없애주지 않으면 프로그램 동작 중에 메모리를 모두 다 잡아먹어 시스템 성능 저하 및 오류로 이어질 수 있습니다.

따라서 반드시 사용하지 않는 동적 메모리는 꼭 바로바로 해제시켜 주는 것이 좋습니다.

ps) memory.h에는 malloc 함수가 정의되어 있지 않습니다. 따라서 필자는 stdlib.h를 대신 사용하였습니다...

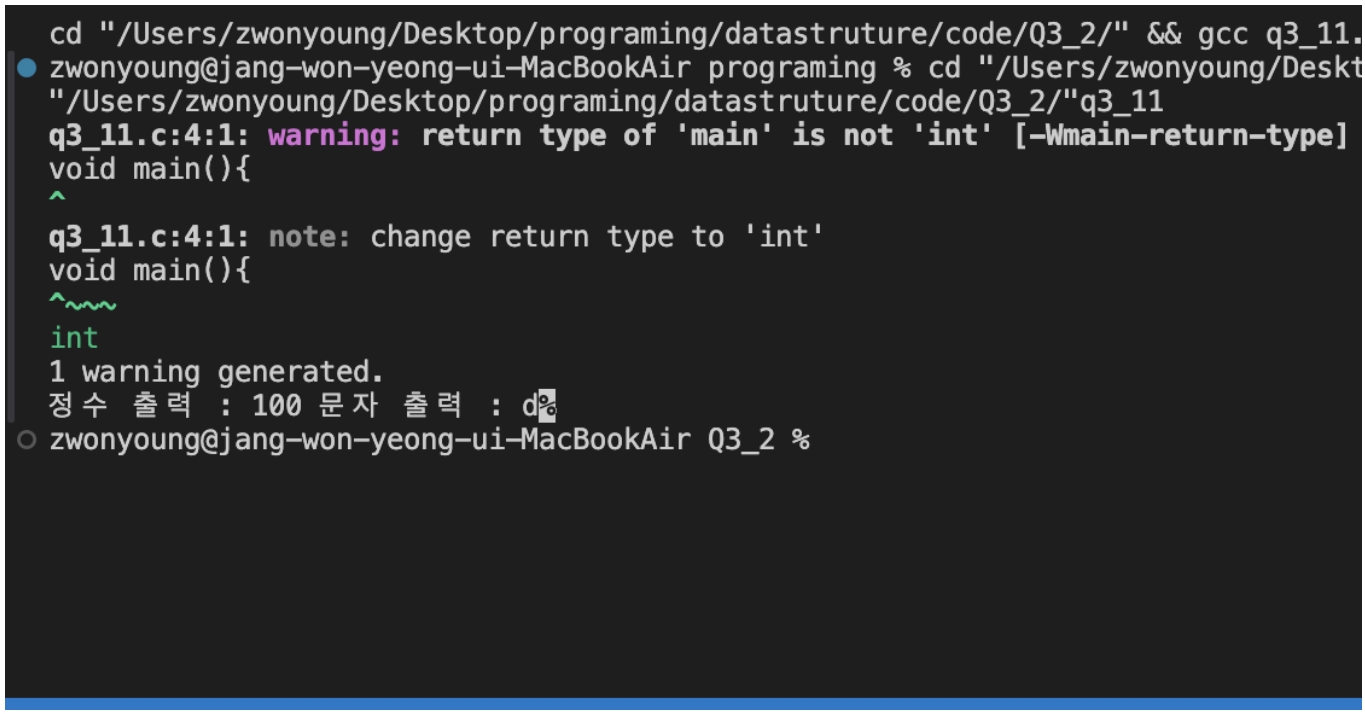
문제 3.(11)

3.(11) 코드 및 주석

```c
#include <stdio.h>
#include <stdlib.h>

void main(){
        // 반복문 횟수 저장용 i
        int i;
        // 동적으로 할당된 메모리의 주소를 저장하기 위한 char형 포인터 변수
        char *pc;

        // 동적 메모리 할당, 크기는 100바이트로 지정
        pc = (char*)malloc(100);

        // 100번 반복
        for(i = 0; i < 100; i++){
        // pc가 가리키는 동적 메모리에 100할당
                *pc = 100;
        }
        // 제대로 할당되었는지와 각각을 10진수 정수와,
        // 문자형으로 해석하면 어떻게 되는지 확인하기 위한 printf
        printf("정수 출력 : %d 문자 출력 : %c", *pc, *pc);
        // 동적으로 할당된 메모리 해제
```

```
        free(pc);
}
```

3.(11) 출력 화면 및 설명



반복문에 사용할 변수 i를 선언해 주었고 할당된 동적 메모리의 공간을 받을 char형 포인터 변수 pc를 선언하고, malloc함수로 100바이트 크기의 동적 메모리를 선언하고 반환된 void형 포인터를 char형 포인터로 타입 캐스팅하여 char형 포인터 pc에 주소를 넘겨주는 모습을 볼 수 있습니다.

이후 반복문을 통해 포인터 pc가 가리키는 위치의 값을 100으로 100번 설정해주는 모습을 볼 수 있습니다.

그리고 각자 %d, %c로 출력하면 정상적으로 나오는 것을 볼 수 있는데 값 자체는 정상적으로 할당되는 것을 볼 수 있습니다.

하지만, 같은 주소값에서 같은 값을 100번 할당하는 것은 사실 의미없는 짓입니다. 문제에서 동적 메모리의 크기를 100바이트로 지정해준 점을 미루어보아 char형 데이터를 100개만큼 저장할 수 있고, 반복문을 100번 돌리고 있으므로 필자는 이 100바이트의 공간을 전부 이용하라는 것으로 해석하였습니다.

```
// 100번 반복
for(i = 0; i < 100; i++){
        // pc가 가리키는 동적 메모리에 100할당
        pc[i] = 100;
}
// 제대로 할당되었는지와 각각을 10진수 정수와,
// 문자형으로 해석하면 어떻게 되는지 확인하기 위한 printf
for(i = 0; i < 100; i++){
```

```
        printf("정수 출력 : %d 문자 출력 : %c\n", pc[i], pc[i]);
}
```

// 100번반복 으로 주석친 부분부터 이런 형태로 배열의 인덱스로 접근하여 값을 집어넣는것이 문제의 의도에 맞는 활용 방법이리고 보아집니다.

결론적으로 코드가 작동은 하지만 문제의 의도대로 동작하지는 않을 것이라는것이 필자의 의견입니다.

문제 3.(12)

3.(12) 코드 및 주석

```
#include <stdio.h>
#include <stdlib.h>

int main(){
        char *pc; // char형 포인터 pc 선언
        int *pi; // int형 포인터 pi 선언
        float *pf; // float형 포인터 pf 선언
        double *pd; // double형 포인터 pd 선언

        // pc에 char데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pc = (char*)malloc(100 * sizeof(char));
        // pc에 int데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pi = (int*)malloc(100 * sizeof(int));
        // pc에 float데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pf = (float*)malloc(100 * sizeof(float));
        // pc에 double데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pd = (double*)malloc(100 * sizeof(double));

        printf("pc의 크기 : %d바이트, pi의 크기 : %d바이트, pf의 크기 : %d바이트, pd의
크기 : %d바이트\n",
        sizeof(char)*100, sizeof(int)*100, sizeof(float)*100,
sizeof(double)*100);

        for(int i = 0; i < 100; i++){ // 각 동적 메모리에 0부터 99번 인덱스까지 i 삽입
                pc[i] = i; // 숫자 i를 삽입
                pi[i] = i; // 숫자 i를 삽입
                pf[i] = i; // 숫자 i를 삽입
```

```
                pd[i] = i; // 숫자 i를 삽입
        }

        for(int i = 97; i < 107; i++){ // 의도적으로 배열의 크기를 넘어가는 107번 인덱스까지 출력
                printf("char형 동적할당 결과 : %c\n", pc[i]);
                printf("int형 동적할당 결과 : %d\n", pi[i]);
                printf("float형 동적할당 결과 : %f\n", pc[i]);
                printf("double형 동적할당 결과 : %lf\n\n", pc[i]);
        }
        // 99번째 인덱스까지는 정상적으로 출력되는지 확인하는 과정

        // 여기까지
        free(pc);
        free(pi);
        free(pf);
        free(pd);

        return 0;
}
```

3.(12) 출력 화면 및 설명

```
6 warnings generated.
pc의 크기 : 100바이트, pi의 크기 : 400바이트, pf의 크기 : 400바이트, pd의 크기 : 800바이트
char형 동적할당 결과 : a
int형 동적할당 결과 : 97
float형 동적할당 결과 : 0.000000
double형 동적할당 결과 : 0.000000

char형 동적할당 결과 : b
int형 동적할당 결과 : 98
float형 동적할당 결과 : 0.000000
double형 동적할당 결과 : 0.000000

char형 동적할당 결과 : c
int형 동적할당 결과 : 99
float형 동적할당 결과 : 0.000000
double형 동적할당 결과 : 0.000000

char형 동적할당 결과 :
int형 동적할당 결과 : 0
float형 동적할당 결과 : 0.000000
double형 동적할당 결과 : 0.000000

char형 동적할당 결과 :
int형 동적할당 결과 : 1065353216
float형 동적할당 결과 : 0.000000
double형 동적할당 결과 : 0.000000

char형 동적할당 결과 :
int형 동적할당 결과 : 1073741824
float형 동적할당 결과 : 0.000000
double형 동적할당 결과 : 0.000000

char형 동적할당 결과 :
int형 동적할당 결과 : 1077936128
float형 동적할당 결과 : 0.000000
```

pc부터 순서대로 각각, char형, int형, float형, double형으로 동적 메모리를 할당해주었고 각각의 데이터 형의 100개 저장할 수 있는 배열이 생성됩니다.

malloc 함수는 인자로 바이트의 크기를 숫자로 받는데 23번줄인

```
printf("pc의 크기 : %d바이트, pi의 크기 : %d바이트, pf의 크기 : %d바이트, pd의 크기 : %d바이트\n",
sizeof(char)*100, sizeof(int)*100, sizeof(float)*100, sizeof(double)*100);
```

로 출력해보면 각각 자료형의 크기와 곱해진 char(1바이트) *100 = 100, int(4바이트)* 100 = 400, float(4바이트) *100 = 400, double(8바이트)* 100 = 800 과 같이 매개변수를 받은 것을 볼 수 있고, 이 크기만큼 동적 메모리가 할당이 되는 것을 알 수 있습니다.

또한 97번 주소부터 일부러 배열의 크기를 넘어가는 107번 인덱스까지 출력해보면 출력은 되지만 쓰레기 값이 출력되는 것을 알 수 있습니다.

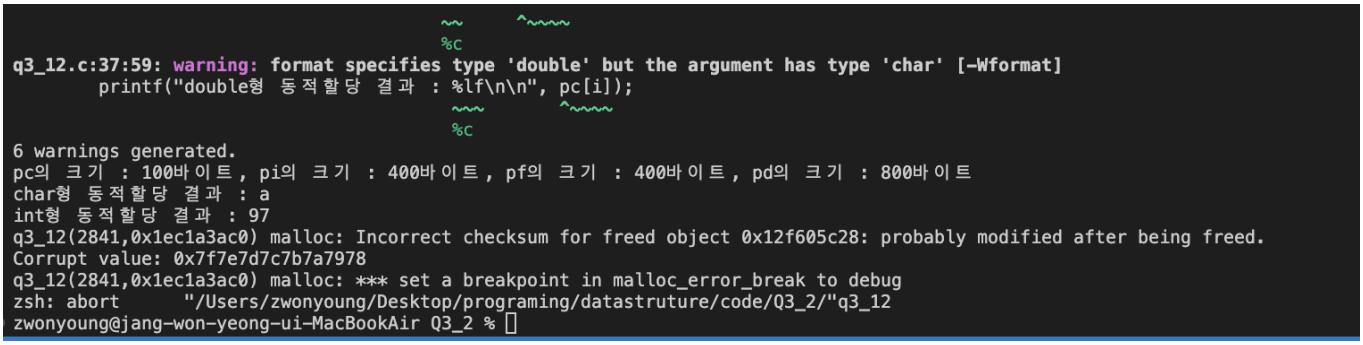
(char형은 아스키코드상 그 숫자랑 연결된 문자를 출력합니다...)

그렇다면 드는 의문은 동적으로 할당된 메모리의 크기를 넘어서서 메모리를 강제로 심을수 있을까? 싶을 것입니다.

인덱스에 값을 심는 부분의 인덱스를 할당 범위를 넘어선 200으로 잡았을때

```
for(int i = 0; i < 200; i++){ // 아까의 코드 100을 200으로 변경
        pc[i] = i; // 숫자 i를 삽입
        pi[i] = i; // 숫자 i를 삽입
        pf[i] = i; // 숫자 i를 삽입
        pd[i] = i; // 숫자 i를 삽입
}
```

다음과 같은 결과가 나오는 것을 볼 수 있습니다.



위와 같은 사례를 보았을때 강제로 동적으로 할당해준 범위를 넘어서 값을 집어넣으면 출력하면 할당되지 않은 메모리까지 값을 침범할 수 있으므로 굉장히 조심해서 써야합니다, 지금과 같은 상황을 보면 값이 할당하지 않은 값이 침범당했음을 인식하고 오류가 있음을 알려줍니다.

가끔씩 그냥 될때도 있긴 하지만 그 때가 더 위험합니다. 항상 동적 메모리를 사용할때는 포인터 침범을 주의하면서 사용해야 합니다.

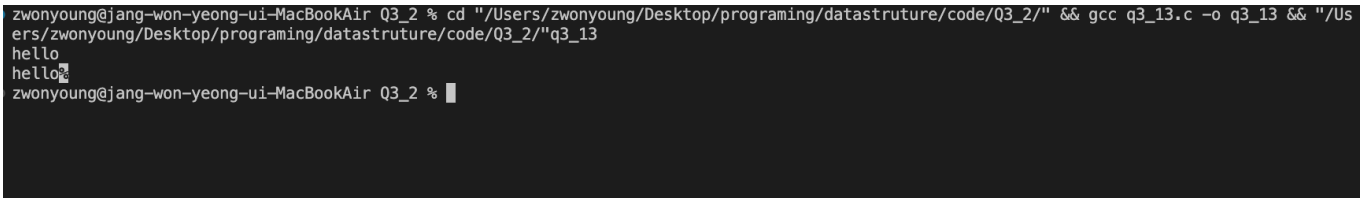
문제 3.(13)

3.(13) 코드 및 주석

```
#include <stdio.h> // 표준 입출력 헤더파일
#include <stdlib.h> // 동적 메모리 사용을 위한 헤더파일

char* alloctest(){ // 메모리를 동적 할당하고 동적 할당한 메모리에 값을 입력받고 그 주소 반환
char *pc; // char형 포인터 pc 선언

// pc에 char데이터형 크기 * 100 바이트 크기만큼 동적할당하고
// 배 동적할당 공간의 시작 주소 반환
pc = (char*)malloc(100 * sizeof(char));
scanf("%s", pc); // 사용자에게 문자열 입력받기
return pc; // 동적 할당된 공간의 포인터 반환하기
}
```

```c
int main(){
char *pc; // char형 포인터 pc 선언
pc = alloctest(); // 동적으로 메모리 생성하고 그 주소 반환하는 함수 호출
// 여기까지

printf("%s", pc); // pc(문자열) 출력
free(pc); // 동적 메모리 해제

return 0;
}
```

3.(13) 출력 화면 및 설명



위의 코드를 살펴보면 char형 포인터 pc를 선언하고 그 포인터에 동적으로 메모리를 할당하고 그 메모리에 값을 사용자로부터 입력받고 alloctest가 반환하는 주소값을 저장하고 그 주소값에 있는(문자열)을 출력하는 코드임을 알 수 있습니다!

여기서 들 수 있는 의문점으로는 함수 안에서 선언한 변수 등등은 함수의 스택이 해제되는 순간 반환한 값을 제외한 모든 함수 안에 저장한 메모리가 불가능한 메모리가 되는 것으로 알고 있는데 이 코드는 정상적으로 hello를 출력한 모습을 볼 수 있습니다.

이것은 동적으로 생성한 메모리는 사용자가 직접 해제시켜주지 않는 한 절대 해제되지 않기 때문입니다.

따라서 free를 해주지 않으면 프로그램 실행 중에는 그 동적으로 생성한 메모리가 계속 남아있는 것을 알 수 있고 그렇게 남아있는 메모리가 하나 둘씩 쌓이면 메모리를 계속 차지해 프로그램의 실행 속도 저하, 더 나아가 시스템을 다운시키는 사고를 일으킬 수 있습니다.

잘만 사용하면 위 사용법과 같이 사용이 가능합니다.

문제 3.(14)

3.(14) 코드 및 주석

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
```

```
        char *pc; // char형 포인터 pc 선언
        int *pi; // int형 포인터 pi 선언
        float *pf; // float형 포인터 pf 선언
        double *pd; // double형 포인터 pd 선언

        // pc에 char데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pc = (char*)malloc(100 * sizeof(char));
        // pc에 int데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pi = (int*)malloc(100 * sizeof(int));
        // pc에 float데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pf = (float*)malloc(100 * sizeof(float));
        // pc에 double데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pd = (double*)malloc(100 * sizeof(double));

        printf("pc = %lu\n", pc); // pc의 주소값 출력
        printf("pc+1 = %lu\n", pc+1); // pc에 +1한 주소값 출력
        printf("pc-1 = %lu\n", pc-1); // pc에 -1한 주소값 출력

        printf("pi = %lu\n", pi); // pi의 주소값 출력
        printf("pi+1 = %lu\n", pi+1); // pi에 +1한 주소값 출력
        printf("pi-1 = %lu\n", pi-1); // pi에 -1한 주소값 출력

        free(pc); // 동적 메모리 할당 해제
        free(pi); // 동적 메모리 할당 해제
        free(pf); // 동적 메모리 할당 해제
        free(pd); // 동적 메모리 할당 해제

        return 0;
}
```

3.(14) 출력 화면 및 설명

3-12 문제와 비슷하게 동적으로 메모리를 할당해주고 char형 포인터를 1씩 더하고 빼고 int형 포인터를 1씩 더하고 뺀 결과를 출력하고 있는 모습입니다.

char형 포인터는 포인터에 1씩 더하고 뺄때 1씩, int형 포인터는 1씩 더하고 뺄때 4씩 움직이는 것을 알 수 있는데 이는 주소 하나가 1바이트씩 가리킬 수 있고 int형은 4바이트씩 움직이므로 포인터도 그에 맞게 움직이

는 것을 알 수 있습니다.

즉, 포인터를 더하고 빼는 행위는 포인터의 자료형의 크기만큼의 주소만큼을 이동하는 것으로 보아도 무방합니다.

문제 3.(15)

3.(15) 코드 및 주석

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
        char *pc; // char형 포인터 pc 선언
        int *pi; // int형 포인터 pi 선언
        float *pf; // float형 포인터 pf 선언
        double *pd; // double형 포인터 pd 선언

        // pc에 char데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pc = (char*)malloc(100 * sizeof(char));
        // pc에 int데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pi = (int*)malloc(100 * sizeof(int));
        // pc에 float데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pf = (float*)malloc(100 * sizeof(float));
        // pc에 double데이터형 크기 * 100 바이트 크기만큼 동적할당하고
        // 배 동적할당 공간의 시작 주소 반환
        pd = (double*)malloc(100 * sizeof(double));

        printf("pf = %lu\n", pf); // pf의 주소값 출력
        printf("pf+1 = %lu\n", pf+1); // pf에 +1한 주소값 출력
        printf("pf-1 = %lu\n", pf-1); // pf에 -1한 주소값 출력

        printf("pd = %lu\n", pd); // pd의 주소값 출력
        printf("pd+1 = %lu\n", pd+1); // pd에 +1한 주소값 출력
        printf("pd-1 = %lu\n", pd-1); // pd에 -1한 주소값 출력

        free(pc); // 동적 메모리 할당 해제
        free(pi); // 동적 메모리 할당 해제
        free(pf); // 동적 메모리 할당 해제
        free(pd); // 동적 메모리 할당 해제
```

```
        return 0;
}
```

3.(15) 출력 화면 및 설명

```
6 warnings generated.
pf = 5366636832
pf+1 = 5366636836
pf-1 = 5366636828
pd = 5366637232
pd+1 = 5366637240
pd-1 = 5366637224
zwonyoung@jang-won-yeong-ui-MacBookAir Q3_2 %
```

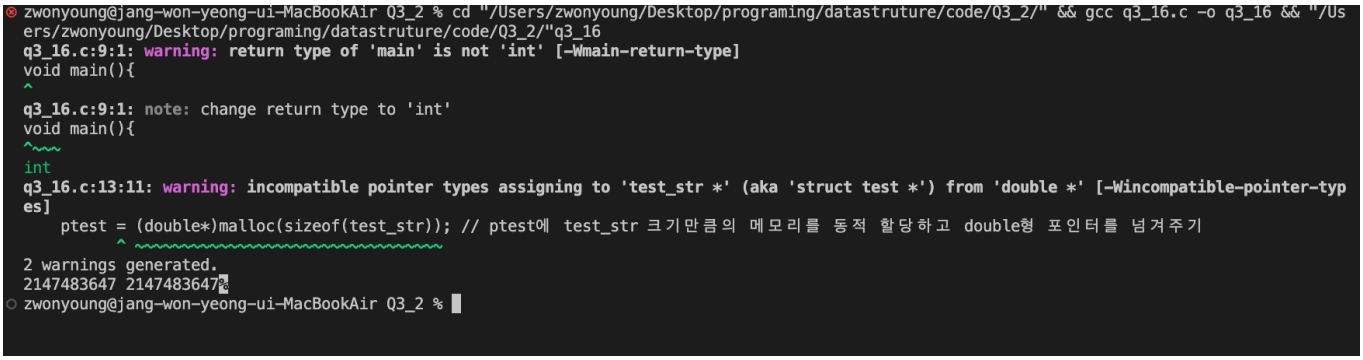
길게 설명할 필요도 없이 위의 3-14에서 설명했던 것처럼 float의 크기 4바이트 double의 크기 8바이트만큼 주소값이 1씩 변화할때마다 주소값은 각각 4, 8씩 이동하는 것을 알 수 있습니다.

문제 3.(16)

3.(16) 코드 및 주석

```
#include <stdio.h>
#include <stdlib.h>

typedef struct test { // 구조체를 test라는 이름으로 생성하고
        int data; // int형 데이터 선언
        struct test* link; // test를 가리킬 수 있는 포인터 link선언
} test_str; // 이 구조체를 test_str이라는 자료형 이름으로 쓸것을 명시

void main(){
        int i; // 인덱스를 지정할 반복문 변수 i
        test_str* ptest; // test_str형 구조체를 가리킬 수 있는 포인터 ptest

        ptest = (double*)malloc(sizeof(test_str)); // ptest에 test_str 크기만큼
의 메모리를 동적 할당하고 double형 포인터를 넘겨주기

        ptest->data = 2147483647; // int에서 할당 가능한 최대한의 데이터 크기 할당
        ptest->link = ptest; // ptest의 링크가 자기 자신을 가리키게 할것
        printf("%d %d", ptest->data, ptest->link->data);
        // 동적 할당한 구조체의 데이터와 그 링크가 자신을 가리켜서 데이터를 정상적으로 출력하는지
확인하는 코드
        free(ptest); // 동적 메모리 해제
}
```

3.(16) 출력 화면 및 설명



일단 구조체부터 간단하게 설명하자면 구조체를 선언하고 있으며 그 구조체의 이름을 test로 해 준 모습입니다.

그리고 그 구조체의 구성을 int형 값을 저장할수 있는 data, test형 즉 자기 자신과 같은 형태를 가진 데이터형을 가리킬 수 있는 link를 선언하였고 이 구조체를 test_str이라는 자료형으로 새로 정의하여 부를 것을 선언했습니다.

여기서 들 수 있는 의문으로는

1. 아직 정의되지 않은 test를 가리킬 수 있는 포인터를 선언할 수 있는가?
2. 만약에 된다고 하여도 왜 안쪽의 포인터는 test_str *로 선언된 것이 아닌 struct test* 로 지정을 해 주는가?
   일텐데 순서대로 답변하자면
3. C언어에서는 아직 선언되지 않은 구조체더라도 그 구조체가 생성될 것이 자명하다면 위 문법과 같이 사용이 가능합니다.
4. 그러나 test_str은 아직 타입으로 지정되기 이전이기 때문에 없는 내용으로 판단, 내부에서 지정할 때에는 test 이름을 기준으로 지정해 주어야 하기 때문입니다.
   따라서 위 구조체는 문제가 없는 것을 알 수 있으며 이 구조체는 test_str 변수명과 같이 끌어와서 사용이 가능한 것을 알 수 있습니다.

이후 그 구조체를 가리킬 수 있는 포인터 ptest를 선언하고 그 구조체의 크기만큼 동적 할당하여 double형 포인터로 돌려주는데, 여기서 문제가 발생합니다!

어떤 문제인가 하면, 위의 3-15 문제를 풀어 보았으면 알겠지만, double형 포인터는 데이터를 8바이트씩 끊어 읽으며 데이터 저장 구조도 위에서 선언한 구조와는 매우 다릅니다.

따라서 double형 포인터로 돌려주는 순간 ptest와는 호환되지 않는 포인터이기 때문에 절대로 값을 제대로 받을 수 없습니다.

따라서 위 코드를 정상적으로 작동시키려면
double *로 캐스팅하는것이 아닌 test_str* 로 캐스팅해야합니다.
아래는 제대로 바꿔주었을때의 코드와 실행 결과입니다.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
typedef struct test { // 구조체를 test라는 이름으로 생성하고
        int data; // int형 데이터 선언
        struct test* link; // test를 가리킬 수 있는 포인터 link선언
} test_str; // 이 구조체를 test_str이라는 자료형 이름으로 쓸것을 명시

void main(){
        int i; // 인덱스를 지정할 반복문 변수 i
        test_str* ptest; // test_str형 구조체를 가리킬 수 있는 포인터 ptest

        ptest = (test_str*)malloc(sizeof(test_str)); // ptest에 test_str 크기만큼의 메모리를 동적 할당하고 double형 포인터를 넘겨주기

        ptest->data = 2147483647; // int에서 할당 가능한 최대한의 데이터 크기 할당
        ptest->link = ptest; // ptest의 링크가 자기 자신을 가리키게 할것
        printf("%d %d", ptest->data, ptest->link->data);
        // 동적 할당한 구조체의 데이터와 그 링크가 자신을 가리켜서 데이터를 정상적으로 출력하는지 확인하는 코드
        free(ptest); // 동적 메모리 해제
}
```

```
zwonyoung@jang-won-yeong-ui-MacBookAir Q3_2 % cd "/Users/zwonyoung/Desktop/programing/datastruture/code/Q3_2/" && gcc q3_16.c -o q3_16 && "/Users/zwonyoung/Desktop/programing/datastruture/code/Q3_2/"q3_16
q3_16.c:9:1: warning: return type of 'main' is not 'int' [-Wmain-return-type]
void main(){
^
q3_16.c:9:1: note: change return type to 'int'
void main(){
^~~~
int
1 warning generated.
2147483647 2147483647
zwonyoung@jang-won-yeong-ui-MacBookAir Q3_2 %
```

위와 같이 정상적으로 포인터를 받고 출력하는 것을 알 수 있습니다.

ptest는 구조체가 저장된 공간을 가리키는 '포인터'이기 때문에 구조체의 요소에 접근하려면 포인터를 통해 접근해야 하는 것을 알 수 있고 구조체의 요소를 포인터로 접근하는 방법은

1. `printf("%d %d", ptest->data, ptest->link->data);`
2. `printf("%d %d\n", (*ptest).data, (*(*ptest).link).data);`
   (*(*ptest).link).data) 만 봐도 알 수 있듯이 -> 가 편리하기 때문에 -> 를 많이 사용합니다.

따라서 위 코드는 -> 문법을 통해 데이터를 2147483647로 링크를 ptest->link = ptest 와 같이 줌으로서 자기 자신을 가리키게 하였고
(ptest의 link는 test_str형 데이터를 가리킬 수 있는 포인터인데 ptest가 test_str형 자료이므로 자기 자신의 주소를 가리킬 수 있다.)
이후 printf를 통해 위 과정이 제대로 이루어졌는지 구조체의 요소별로 출력해준 모습입니다.

문제 3.(17)

3.(17) 코드 및 주석

```c
#include<stdio.h>
#include<stdlib.h>

// 추론
//sizeof()함수는 해당 자료형의 단순 크기를 바이트 크기로 반환한다
// int형은 4바이트, char형은 1바이트이므로 int 포인터로 읽어들이면
// 25개의 정수형 데이터를 받을수 있는 배열이 생성될 것이다.

// 실제로 그런지 확인해보자.

int main(){
        int *pi; // 원래 조건

        int sizepi; // 원래 조건의 할당된 크기를 받을 정수
        int sizerightpi; // 제대로 할당된 배열의 크기를 받을 정수

        sizepi = 100 * sizeof(char); // char형 크기에 100을 곱한 정수 저장
        sizerightpi = 100 * sizeof(int); // int형 크기에 100을 곱한 정수 저장

        pi = (int *)malloc(100 * sizeof(char)); // 문제에서 물어보는 조건 할당

        // 배열 인덱스 방식으로 값을 집어넣을때 값이 정상적으로 넣어지고 읽어지는지 확인
        for(int i = 0; i < 25; i++){
                pi[i] = i;
                printf("%d\n", pi[i]);
        }
        free(pi); // 동적 할당 메모리 해제
        free(rightpi); // 동적 할당 메모리 해제

        printf("char로 했을때의 크기 : %d int로 했을때의 크기 : %d", sizepi,
sizerightpi);

        return 0;
}
```

3.(17) 출력 화면 및 설명

일단은 코드부터 살펴보자면,

int형 포인터 pi를 선언하고 char형으로 크기 주었을때, int형으로 크기 주었을때 각각의 크기를 정수로 저장할 sizepi, sizerightpi를 선언하여 각각의 자료형의 크기 곱하기 100을 해 주었습니다.

이후 pi에 문제의 조건에 맞게 char형 크기 * 100의 크기만큼의 공간을 할당하고 그 공간의 주소를 int형 포인터로 타입캐스팅하여 pi에 넘겨주었고 0부터 25번까지의 인덱스에 정상적으로 값이 들어가는지 출력하고 테스트할 for문을 선언해 준 뒤,

동적 메모리를 해제하고

char형으로 선언했을때, int형으로 선언했을때의 크기를 출력해 주었습니다.

이 코드로 미루어 보아 알수 있는 것은 malloc('이곳') 즉 매개변수로 char형을 넣어준다 하더라도 크기만 다를 뿐 그 동적 할당된 메모리만 int 포인터형으로 제대로 타입캐스팅해서 돌려준다면 아무런 문제 없이 작동하는 것을 알 수 있습니다.

다만, akwlakr printf문을 보면 알 수 있듯이 각각의 크기가 100, 400으로 차이가 나므로, 실제로 동적 할당된 크기는 1/4 즉 25개의 정수를 저장할 수 있는 공간이 할당되었음을 알 수 있고, 포인터를 통해 값을 어떻게 저장할 수 있는지를 알 수 있습니다....

135p 17번 문제

17번 코드 및 주석

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h> // 기본 입출력을 위한 헤더
#include <stdlib.h> // 동적 메모리 할당, rand함수, NULL을 사용하기 위한 헤더
```

```c
#include <string.h> // strlen, strcpy, strcmp 사용을 위한 헤더
#include <stdbool.h> // boolean형 자료형, true, false를 표현하기 위한 헤더

typedef char DataType; // 코드 유지보수 증대를 위해 char를 DataType로 변경하여 해석

typedef struct Node { // 노드의 형태 정의
        struct Node* llink; // 왼쪽 요소를 가리키는 링크
        DataType* data; // 동적 할당된 문자열을 가리키는 data
        struct Node* rlink; // 오른쪽 요소를 가리키는 링크
}Node; // 명칭은 Node

typedef struct {// 노드를 가리키는 헤드노드의 정의
        Node* head; // 맨 처음 노드를 가리키는 포인터
        int count; // 링크드 리스트의 길이를 저장하는 변수
}Headnode; // 명칭은 Headnode

// 링크드 리스트 생성부
Headnode* mk_L_list(){ // 빈 링크드 리스트 생성, 파라미터 : 없음
        Headnode* H = (Headnode*)malloc(sizeof(Headnode)); // 헤드노드를 동적 생성하여 H에 할당
        if(!H){ // 할당 못하면(H == NULL 의 역이므로 참)
                printf("동적 할당 실패!\n"); // 경고 메세지 출력
                exit(1); // 오류코드 반환 후 프로그램 종료
        }
        H->count = 0; // 카운트를 0으로 초기화
        H->head = NULL; // 갱글링 포인터 방지를 위해 NULL을 할당
        return H; // 동적 할당한 헤드노드의 주소 반환
}

// 삽입
Node* mknode(DataType* data){ // 새로운 노드 생성, node_in함수에 종속적, 파라미터 : 복사받을 문자열의 주소
        Node* newnode = (Node*)malloc(sizeof(Node)); // 새로운 노드 생성
        if(!newnode){ // 할당 못하면(newnode == NULL 의 역이므로 참)
                printf("동적 할당 실패!(연결 리스트)\n"); // 경고 메세지 출력
                exit(1); // 오류코드 반환 후 프로그램 종료
        }

        char* cpydata = (DataType*)malloc(sizeof(DataType) * strlen(data) + 1); // 문자열 복사받을 공간 동적 생성
        if(!cpydata){ //할당 못하면(newnode == NULL 의 역이므로 참)
                printf("동적 할당 실패!(문자열)\n"); // 경고 메세지 출력
                exit(1);// 오류코드 반환 후 프로그램 종료
        }
        strcpy(cpydata, data); // 동적 생성한 문자열 공간에 파라미터로 받은 문자열 복사
```

```
        newnode->data = cpydata; // 동적할당된 cpydata를 newnode가 가리키게 함
        newnode->llink = NULL; // 갱글링 포인터 방지를 위해 NULL을 할당
        newnode->rlink = NULL; // 갱글링 포인터 방지를 위해 NULL을 할당

        return newnode; // 만들어진 노드의 주소 반환
}

void node_in_first(Headnode* H, Node* newnode, Node* prenode){ // 첫번째 노드로
만들기, node_in에 종속적, 파라미터 : 헤드노드 포인터, 새로운 노드의 주소, 이전 노드의 주소
        H->head = newnode; // 헤드 노드를 새로운 노드로 설정
        newnode->llink = newnode; // 자기 자신을 가리킴
        newnode->rlink = newnode; // 자기 자신을 가리킴
        H->count++; // 노드 수 증가
}
void node_in_normal(Headnode* H, Node* newnode, Node* prenode){ // 노드 끼워넣
기, node_in에 종속적, 파라미터 : 헤드노드 포인터, 새로운 노드의 주소, 이전 노드의 주소
        newnode->llink = prenode; // 새 노드의 llink를 이전 노드로 설정
        newnode->rlink = prenode->rlink; // 새 노드의 rlink를 이전 노드의 rlink로
설정
        prenode->rlink->llink = newnode; // 이전 노드의 rlink가 가리키는 노드의
llink를 새 노드로 설정
        prenode->rlink = newnode; // 이전 노드의 rlink를 새 노드로 설정
        H->count++; // 노드 수 증가
}
Node* node_in(Headnode* H, DataType* data, Node* prenode){ // 노드 생성 후 삽입,
종속 함수 2개 보유, 파라미터 : 헤드노드 포인터, 문자열 주소, 이전 노드의 주소
        Node* newnode = mknode(data); // 새로운 노드 할당

        if (!H->head) { // 리스트가 비어있을 경우
                node_in_first(H, newnode ,prenode); // 첫번째 노드로 만들기
        } else { // 리스트가 비어있지 않을 경우
                node_in_normal(H, newnode ,prenode); // 끼워넣기
        }
                return newnode; // 새로 생성한 노드의 주소 반환
}

// 삭제
void node_out_one(Headnode* H, Node* delnode){ // 빈 링크드 리스트로 만들기,
node_out에 종속적, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        free(delnode->data); // 지울 노드의 data가 가리키는 문자열의 동적 할당 해제
        free(delnode); // 생성했던 노드의 동적 할당 해제

        H->head = NULL; // head포인터가 NULL을 가리키게 하여 빈 링크드 리스트임을 표시
        H->count--; // 노드의 수 카운트 1개 감소
}
void node_out_normal(Headnode* H, Node* delnode){ // 지정한 노드 제거하기,
```

```
node_out, node_out_head에 종속적, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        delnode->llink->rlink = delnode->rlink; // 지우는 노드의 왼쪽 링크 즉, 이전 노드의 rlink가 지우는 노드의 다음 노드를 가리키게 한다.
        delnode->rlink->llink = delnode->llink; // 지우는 노드의 오른쪽 링크 즉, 다음 노드의 llink가 지우는 노드의 이전 노드를 가리키게 한다.

        free(delnode->data); // 지울 노드의 data가 가리키는 문자열의 동적 할당 해제
        free(delnode); // 생성했던 노드의 동적 할당 해제
        H->count--; // 노드의 수 카운트를 1개 감소시킨다.
}
void node_out_head(Headnode* H, Node* delnode){ // 헤드노드 옮기고 사이에 낀 노드 제거하기, node_out에 종속적, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        H->head = H->head->rlink; // 헤드포인터가 가리키는 노드의 오른쪽 노드를 헤드노드로 만든다.
        node_out_normal(H, delnode); // 지정한 노드 제거하기
}
void node_out(Headnode* H, Node* delnode){ // 지정한 노드 제거하기, 종속 함수 3개 보유, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        if(!H->head){ // 리스트가 비었으면
                printf("nodeout : 리스트가 비었습니다!\n"); // 메세지 출력
                return; // 함수 종료
        } else if(H->head == delnode && H->head == H->head->rlink){ //리스트에 원소가 하나이면
                printf("nodeout.one 지운 노드 : %s\n", delnode->data); // 메세지 출력
                node_out_one(H, delnode); // 빈 링크드 리스트로 만들기
                return; // 함수 종료
        } else if(H->head == delnode && !(H->head == H->head->rlink)){ // 지우는게 head 노드이면
                printf("nodeout.head 지운 노드 : %s\n", delnode->data); // 메세지 출력
                node_out_head(H, delnode); // 헤드 노드 옮기고 지정한 노드 제거하기
                return; // 함수 종료
        } else { // 그냥 제거해도 이상 없을때
                printf("nodeout.normal 지운 노드 : %s\n", delnode->data); // 메세지 출력
                node_out_normal(H, delnode); // 지정한 노드 제거하기
        return; // 함수 종료
        }
}

// 탐색
Node* node_search_int_L(Node* point){ // 왼쪽으로 한 칸 이동, node_serch에 종속, 파라미터 : 시작 노드의 주소
        point = point->llink; // point의 왼쪽을 point로 새로 지정
        return point; // point 반환
```

```
}
Node* node_search_int_R(Node* point){ // 오른쪽으로 한 칸 이동, node_serch에 종속, 파라미터 : 시작 노드의 주소
        point = point->rlink; // point의 왼쪽을 point로 새로 지정
        return point; // point 반환
}
Node* node_search(Headnode* H, Node* point, bool RL){ // 노드 지정한 횟수만큼 탐색 후 그 주소 반환, 종속 함수 2개 보유, 파라미터 : 헤드노드 포인터, 시작 노드의 주소, 회전옵션
        if(H->head == NULL){ // 노드가 비어있으면
                printf("nodeserch : 노드가 비었습니다!"); // 메세지 출력
                return NULL; // NULL 반환
        } else if(RL == false){ // 회전옵션이 false(오른쪽이면)
                return node_search_int_R(point); // 오른쪽으로 한칸 이동한 노드의 주소를 반환
        } else { // 회전옵션이 true(왼쪽이면)
                return node_search_int_L(point); // 왼쪽으로 한 칸 이동한 노드의 주소를 반환
        }
}

// 전체 노드 출력
void print_all_node(Headnode* H){ // 연결 리스트의 모든 노드 상황 출력
        if (H->head == NULL) { // 리스트가 비었으면
                printf("allnode : 노드가 비었습니다!\n"); // 메세지 출력
                return; // 함수 종료
        } else { // 리스트가 비어있지 않으면
                Node* cursor = H->head; // 노드를 가리킬 수 있는 포인터를 선언 및 헤드 가리키기
                for(;cursor != NULL;){ // 커서가 널값이 아니면(무한루프)
                        printf("%s -> ", cursor->data); // 커서가 가리키는 데이터 출력
                        cursor = cursor->rlink; // 커서의 오른쪽 노드를 커서에 저장
                        if (cursor == H->head){ // 커서가 head노드를 가리키면
                                break; // 반복문 종료
                        }
                }
        printf("\n"); // 구문 구분용 개행문자 출력
        }
}

// 메인 구현부
int main() {
        int N; // 건너뛸 횟수
        scanf("%d", &N); // 건너뛸 횟수 입력받기

        Headnode* H = mk_L_list(); // 새로운 링크드 리스트 생성
```

```
        Node* cursor = H->head; // 삽입 및 삭제용 커서 생성

        // 삽입부
        DataType name[50]; // 사용자가 입력하는 문자열의 크기
        for(;1;) { // (무한 루프)
                scanf("%s", name); // 이름,종료 입력받기
                if(!strcmp(name, "stop")) { // 입력값이 "stop"이면
                        break; // 반복문 종료
                } else { // 입력값이 "stop"이 아니면
                cursor = node_in(H, name, cursor); // 입력값을 노드에 삽입하고, 노드 주소 커서에 넘겨주기
                }
        }

        bool RL = false; // 방향 확인 및 랜덤 지정용 RL 선언

        cursor = H->head; //커서 일단 head노드가 가리키는 노드로 초기화
        // 랜덤 지정부
        int randnum; // 랜덤 반복 횟수 randnum 선언
        print_all_node(H); // 입력된 노드들을 출력
        srand(time(NULL)); // 시드값을 시간으로 주기
        randnum = rand() % H->count; // 랜덤값을 연결 리스트의 길이 범위 내에서 사용할 것을 명시
        for (int i = 0; i < randnum; i++){ // 랜덤값 만큼 반복
                cursor = node_search(H, cursor, RL); // 노드를 회전옵션 방향으로 1칸 이동
        }
        printf("랜덤으로 지정된 head는 : %s\n", cursor->data); // 커서가 가리키는 랜덤 헤드를 출력
        H->head = cursor; // head를 진짜 랜덤으로 지정한 cursor로 변경

        // 삭제부
        while (H->head != NULL) { // 리스트가 비어있지 않을때까지
                // N만큼 이동
                for (int i = 0; i < N; i++) {
                        cursor = node_search(H, cursor, RL); // 방향옵션쪽으로 1칸 이동
                        printf("커서 : %s\n", cursor->data); // 이동한 커서의 위치 출력
                }
                printf("지운 노드 : %s\n 현재 노드의 수 : %d\n", cursor->data, H->count); // 지울 노드와 현재 노드의 수 출력
                Node* next_cursor = node_search(H, cursor, RL); // 다음 노드를 가리키는 포인터를 저장

                node_out(H, cursor); // 현재 노드를 삭제
```

```
                print_all_node(H); // 현재 모든 노드 출력
                cursor = next_cursor; // 다음 노드로 이동
                printf("다음 커서 : %s\n", cursor->data); // 다음 커서가 가리키는 내
용 출력
                printf("\n"); // 구분용 개행문자
                RL = !RL; // 방향옵션 반전 주기
        }
        // 모든 노드 삭제 후 동적할당한 헤드노드 해제
        free(H);

        return 0;
}
```

랜덤 로직 미적용 버전 (main만 따로)

```
// 메인 구현부
int main() {
        int N; // 건너뛸 횟수
        scanf("%d", &N); // 건너뛸 횟수 입력받기

        Headnode* H = mk_L_list(); // 새로운 링크드 리스트 생성
        Node* cursor = H->head; // 삽입 및 삭제용 커서 생성

        // 삽입부
        DataType name[50]; // 사용자가 입력하는 문자열의 크기
        for(;1;) { // (무한 루프)
                scanf("%s", name); // 이름,종료 입력받기
                if(!strcmp(name, "stop")) { // 입력값이 "stop"이면
                        break; // 반복문 종료
                } else { // 입력값이 "stop"이 아니면
                        cursor = node_in(H, name, cursor); // 입력값을 노드에 삽
입하고, 노드 주소 커서에 넘겨주기
                }
        }
        print_all_node(H); // 입력된 노드들을 출력
        cursor = H->head; // 커서 초기화 랜덤 로직 미적용

        // 삭제부
        bool RL = false; // 방향 확인용 RL 선언
        while (H->head != NULL) { // 리스트가 비어있지 않을때까지
// N만큼 이동
                for (int i = 0; i < N; i++) {
                        cursor = node_search(H, cursor, RL); // 방향옵션쪽으로 1
칸 이동
                        printf("커서 : %s\n", cursor->data); // 이동한 커서의 위치
```

```
출력
                }
                printf("지운 노드 : %s\n 현재 노드의 수 : %d\n", cursor->data, H-
>count); // 지울 노드와 현재 노드의 수 출력
                Node* next_cursor = node_search(H, cursor, RL); // 다음 노드를
가리키는 포인터를 저장

                node_out(H, cursor); // 현재 노드를 삭제
                print_all_node(H); // 현재 모든 노드 출력
                cursor = next_cursor; // 다음 노드로 이동
                printf("다음 커서 : %s\n", cursor->data); // 다음 커서가 가리키는 내
용 출력
                printf("\n"); // 구분용 개행문자
                RL = !RL; // 방향옵션 반전 주기
        }
        // 모든 노드 삭제 후 동적할당한 헤드노드 해제
        free(H);

        return 0;
}
```

17번 출력 화면 및 설명

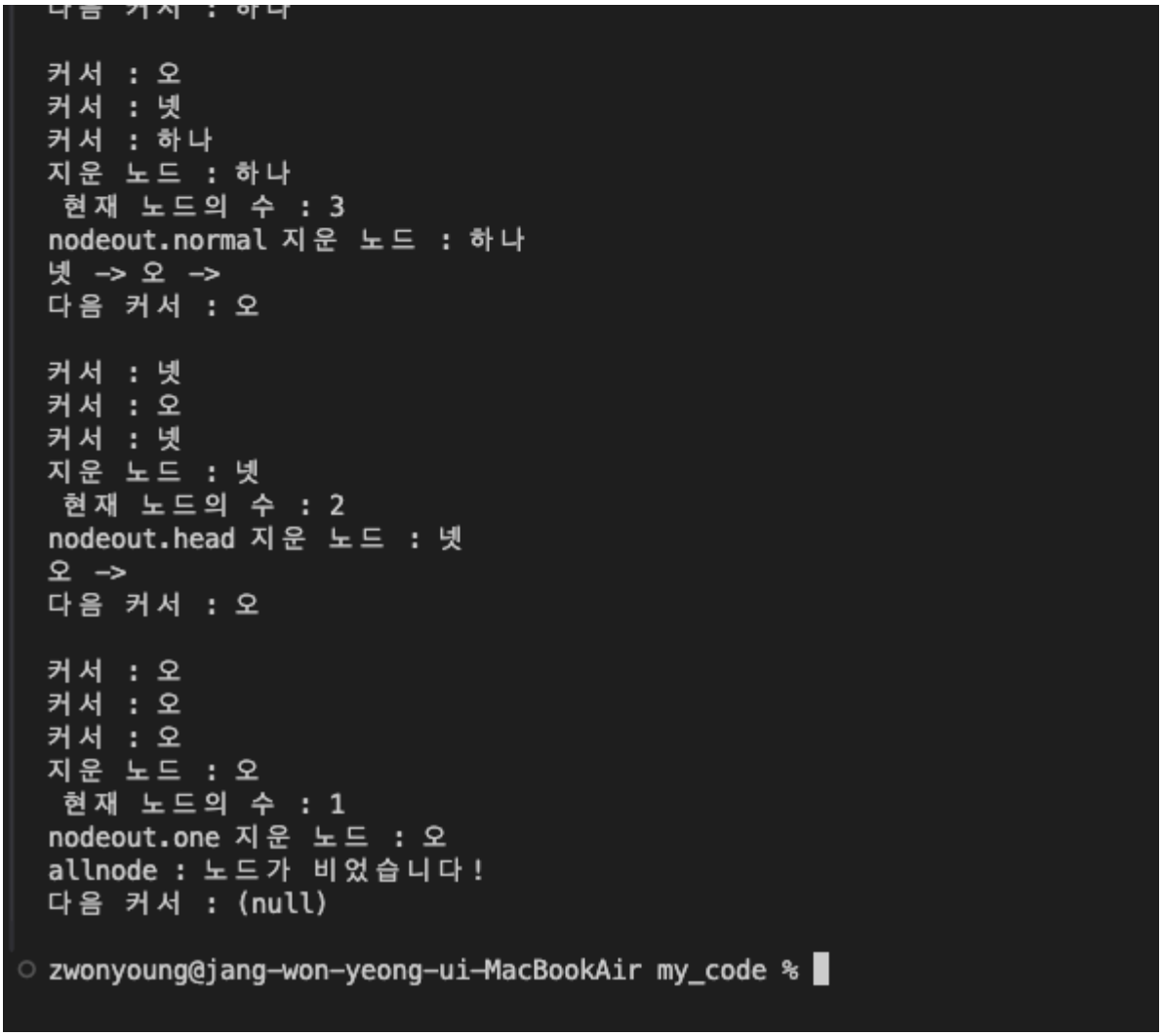
```
zwonyoung@jang-won-yeong-ui-MacBookAir my_code % cd "/Users/zwonyoung/D
ure/code/Q3_2/my_code/"mycode
3
하나
둘
삼
넷
오
육
칠
팔
stop
하나 -> 둘 -> 삼 -> 넷 -> 오 -> 육 -> 칠 -> 팔 ->
랜덤으로 지정된 head는 : 삼
커서 : 넷
커서 : 오
커서 : 육
지운 노드 : 육
 현재 노드의 수 : 8
nodeout.normal 지운 노드 : 육
삼 -> 넷 -> 오 -> 칠 -> 팔 -> 하나 -> 둘 ->
다음 커서 : 칠

커서 : 오
커서 : 넷
커서 : 삼
지운 노드 : 삼
 현재 노드의 수 : 7
nodeout.head 지운 노드 : 삼
넷 -> 오 -> 칠 -> 팔 -> 하나 -> 둘 ->
다음 커서 : 둘

커서 : 넷
커서 : 오
커서 : 칠
지운 노드 : 칠
 현재 노드의 수 : 6
nodeout.normal 지운 노드 : 칠
넷 -> 오 -> 팔 -> 하나 -> 둘 ->
다음 커서 : 팔

커서 : 오
커서 : 넷
커서 : 둘
지운 노드 : 둘
 현재 노드의 수 : 5
nodeout.normal 지운 노드 : 둘
넷 -> 오 -> 팔 -> 하나 ->
다음 커서 : 하나

커서 : 넷
커서 : 오
커서 : 팔
지운 노드 : 팔
 현재 노드의 수 : 4
nodeout.normal 지운 노드 : 팔
넷 -> 오 -> 하나 ->
```



문제에서 요구하는 핵심 조건은 다음과 같습니다.

1. 사람의 입력은 정해지지 않음
2. 입력이 종료되면 입력한 N만큼 방향을 바꿔가면서 건너뛰고 해당 사람 지우기
3. 지운 다음 사람부터 다시 반복

문제에서 계속 방향을 바꾸므로 2중 연결 리스트로 구현하는것이 이점이 있다고 판단,
회전을 진행하므로 원형 리스트로 구현하는것이 이점이 있다고 판단,

즉 원형 이중 연결리스트를 구현하여 문제 해결을 하는 것으로 접근하였습니다.

이에 따라 문제 조건에 맞는 코드 구현을 위해 다음과 같은 기능으로 세분화해 함수화하여 나누어 개발하였습니다...

구조 명세서

---

1. 구조의 정의(구조체)
    1. 데이터의 정의

    2. 노드의 구조 정의
    3. 노드를 가리킬 헤드노드의 구조 정의
2. 동작의 정의(함수)
    1. 연결 리스트 생성
    2. 삽입
        1. 노드 생성
        2. 노드가 비었을때의 삽입
        3. 노드가 차있을때의 삽입
        4. 메인 삽입
    3. 삭제
        1. 연결 리스트에 노드가 하나 있을때의 삭제
        2. 지우려는 Node가 head노드일때의 삭제
        3. 보통 노드 삭제
        4. 메인 삭제
    4. 탐색
        1. 오른쪽 탐색
        2. 왼쪽 탐색
        3. 메인 탐색
    5. 모든 노드 출력
3. 메인 함수
    1. 입력받기
    2. 랜덤 노드 지정
    3. 노드 하나씩 제거하기

1. 구조의 정의

```
typedef char DataType; // 코드 유지보수 증대를 위해 char를 DataType로 변경하여 해석

typedef struct Node { // 노드의 형태 정의
        struct Node* llink; // 왼쪽 요소를 가리키는 링크
        DataType* data; // 동적 할당된 문자열을 가리키는 data
        struct Node* rlink; // 오른쪽 요소를 가리키는 링크
}Node; // 명칭은 Node

typedef struct {// 노드를 가리키는 헤드노드의 정의
        Node* head; // 맨 처음 노드를 가리키는 포인터
```

```
        int count; // 링크드 리스트의 길이를 저장하는 변수
}Headnode; // 명칭은 Headnode
```

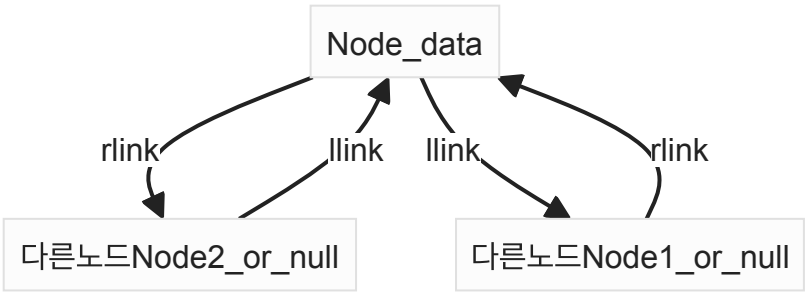
굳이 해 줄 필요는 없지만 코드의 유지보수성 증대를 위해 DataType이라는 이름으로 char형 데이터를 부를 것을 선언하였습니다.

이중 연결 리스트를 사용할 것이므로 노드는 이전 노드와 다음 노드를 가리킬 수 있는 Node 포인터형 llink와 rlink를 선언해 주었고, 필자는 문자열을 동적 할당할 것이기에 동적 할당한 문자열을 가리킬 수 있는 포인터 data를 선언해 주었습니다. 이게 하나의 노드입니다.

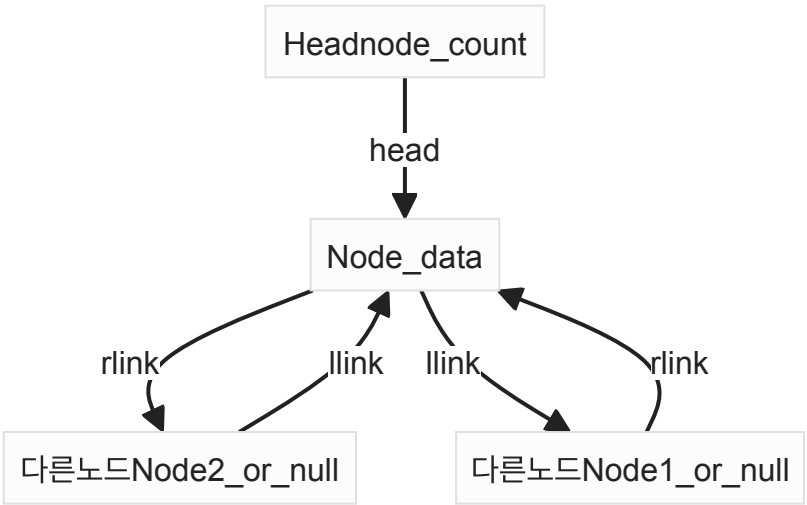
그리고 연결리스트 자체를 가리킬 head노드를 정의해주었는데 Headnode로 부를 것을 정의하고 각각의 요소로 Node포인터형 head를 선언하여 이 head를 통해 연결 리스트를 가리킬 수 있게 하였고,
해당 연결 리스트의 node가 몇개나 있는지 체크할 count 변수도 선언하였습니다.

각각의 구조를 시각화 한 결과는 다음과 같습니다.

Node



Headnode



이것 자체가 연결 리스트가 되는 것이라 보아도 무방합니다.

2. 동작의 정의

구조체는 정의하였으나 이 구조체를 어떻게 연결하고 동작시킬 것인지 정의하여야 합니다.

2.1 연결 리스트 생성

```
// 링크드 리스트 생성부
Headnode* mk_L_list(){ // 빈 링크드 리스트 생성, 파라미터 : 없음
        Headnode* H = (Headnode*)malloc(sizeof(Headnode)); // 헤드노드를 동적 생성하여 H에 할당
        if(!H){ // 할당 못하면(H == NULL 의 역이므로 참)
                printf("동적 할당 실패!\n"); // 경고 메세지 출력
                exit(1); // 오류코드 반환 후 프로그램 종료
        }
        H->count = 0; // 카운트를 0으로 초기화
        H->head = NULL; // 갱글링 포인터 방지를 위해 NULL을 할당
        return H; // 동적 할당한 헤드노드의 주소 반환
}
```

연결리스트를 생성하는 행위를 따로 정의하였습니다.

Headnode형 포인터 H를 선언하고 Headnode형의 크기만큼 동적 생성한 공간을 H에 할당해 주었습니다.
만약 할당하지 못한다면 malloc 함수는 null값을 반환하므로 이 널값의 부정을 통해 강제로 if조건식의 조건식을 참으로 만들어 exit(1); 을 통해 프로그램을 종료하도록 만들었습니다.

이후 성공적으로 할당에 성공했다면 헤드노의의 카운터를 초기화하고, head가 가리키는 값을 0으로 만들어 빈 연결리스트를 생성합니다.

2.2 삽입

삽입 파트는 1개의 삽입 메인 함수, 3개의 종속적인 함수로 구성되어 있습니다.
메인 함수에서 다른 종속적인 함수를 가지고와서 처리하는 방식입니다.

먼저 문제를 쪼개어 상황들을 떠올려 보면 기능을 다음과 같이 나누어 개발해야 하는 것을 알 수 있습니다.

1. 메인 노드 삽입 함수
2. 노드 생성
3. 노드가 비었을때의 삽입
    1. -> headnode가 단순히 새로 생성된 노드를 가리키게 한다.
4. 노드가 차있을때의 삽입

1. 새로운 노드가 양옆을 가리키고 이전 노드와 이전 노드의 다음 노드가 새로운 노드를 가리키게 한다.

5. 모든 로직 완료후 생성한 노드의 주소 반환

따라서

문제 조건에 따라 파트별 함수로 나누었고 그것을 기능에 따라 또 세분화 하였습니다.

2.2.1 노드 생성

```
Node* mknode(DataType* data){ // 새로운 노드 생성, node_in함수에 종속적, 파라미터 : 복사받을 문자열의 주소
        Node* newnode = (Node*)malloc(sizeof(Node)); // 새로운 노드 생성
        if(!newnode){ // 할당 못하면(newnode == NULL 의 역이므로 참)
                printf("동적 할당 실패!(연결 리스트)\n"); // 경고 메세지 출력
                exit(1); // 오류코드 반환 후 프로그램 종료
        }

        char* cpydata = (DataType*)malloc(sizeof(DataType) * strlen(data) + 1); // 문자열 복사받을 공간 동적 생성
        if(!cpydata){ //할당 못하면(newnode == NULL 의 역이므로 참)
                printf("동적 할당 실패!(문자열)\n"); // 경고 메세지 출력
                exit(1);// 오류코드 반환 후 프로그램 종료
        }
        strcpy(cpydata, data); // 동적 생성한 문자열 공간에 파라미터로 받은 문자열 복사

        newnode->data = cpydata; // 동적할당된 cpydata를 newnode가 가리키게 함
        newnode->llink = NULL; // 갱글링 포인터 방지를 위해 NULL을 할당
        newnode->rlink = NULL; // 갱글링 포인터 방지를 위해 NULL을 할당

        return newnode; // 만들어진 노드의 주소 반환
}
```

단순히 데이터를 받으면 새로운 노드를 동적할당을 통해 생성하고 데이터의 크기만큼을 동적 할당한 문자열 공간에 집어넣습니다.

이후 동적 할당한 문자열을 동적 할당한 노드의 data가 가리키게 하고, llink와 rlink는 각각 널값을 가리키게 하여 혹시 모를 갱글링 포인터를 방지합니다.

그리고 이후 할당이 전부 제대로 이루어 졌으면 새 노드의 주소를 반환합니다.

2.2.2 노드가 비었을때의 삽입

```
void node_in_first(Headnode* H, Node* newnode, Node* prenode){ // 첫번째 노드로 만들기, node_in에 종속적, 파라미터 : 헤드노드 포인터, 새로운 노드의 주소, 이전 노드의 주소
        H->head = newnode; // 헤드 노드를 새로운 노드로 설정
```

```
        newnode->llink = newnode; // 자기 자신을 가리킴
        newnode->rlink = newnode; // 자기 자신을 가리킴
        H->count++; // 노드 수 증가
}
```

파라미터로 각각 헤드노드의 포인터, 새로운 노드의 주소, 이전 노드의 주소를 받아 작업합니다.

노드가 비었을때는 Head포인터가 아무것도 가리키지 않을때를 의미하기도 합니다.

따라서 새로 들어오는 노드는 새로운 헤드가 되게 됩니다.

위 코드에서는 newnode의 주소를 head가 가리키게 하였고
원형 연결 리스트를 구현해야 하므로 llink와 rlink가 자기 자신을 가리키게 하였습니다.
노드의 요소가 하나임에도 이렇게 해 주는 것에는 이점이 있는데,
이후에 설명할 2개, 3개이상일때의 삽입로직을 적용할때도 이렇게 해 놓는 편이 일관적인 로직을 적용할 수 있기 때문입니다. 자세한 사항은 1.1.3에서 자세히 설명드리겠습니다.

이후 헤드노드의 카운터를 1 증가시켜 노드의 요소가 1개 늘어났음을 표현하였습니다.

2.2.3 노드가 차있을때의 삽입

```
void node_in_normal(Headnode* H, Node* newnode, Node* prenode){ // 노드 끼워넣기, node_in에 종속적, 파라미터 : 헤드노드 포인터, 새로운 노드의 주소, 이전 노드의 주소
        newnode->llink = prenode; // 새 노드의 llink를 이전 노드로 설정
        newnode->rlink = prenode->rlink; // 새 노드의 rlink를 이전 노드의 rlink로 설정
        prenode->rlink->llink = newnode; // 이전 노드의 rlink가 가리키는 노드의 llink를 새 노드로 설정
        prenode->rlink = newnode; // 이전 노드의 rlink를 새 노드로 설정
        H->count++; // 노드 수 증가
}```

파라미터로 각각 헤드노드의 포인터, 새로운 노드의 주소, 이전 노드의 주소를 받아 작업합니다.
node에 요소가 있으므로 연결 리스트를 다룰 때의 안전 수칙을 잘 지켜서 삽입합니다.
1. 새로운 노드를 먼저 연결한다.
2. 연결이 완료되면 기존의 연결을 새로운 노드에 이어준다.
3. ~~안지키면 막 segmentation fault 뜨고 난리납니다~~
이를 구현한 순서는 다음과 같습니다.

newnode, 즉 집어넣을 노드의 llink가 prenode를 가리키게 하고,

집어넣을 노드의 rlink가 prenode의 rlink를 가리키게 하는데 이게 무슨 의미인지 파악해보면,

newnode를 넣기 전의 prenode는 prenode의 다음 노드와 서로를 가리키고 있기 때문에 prenode->rlink는 연결하기 전의 다음 노드가 되는 것이고 그것을 newnode의 rlink가 가리키게 합니다
```

```
그림으로 표현하면 다음과 같습니다.
``` mermaid
flowchart TD
        prenode -->|rlink| nextnode
        nextnode -->|llink| prenode

        newnode -->|rlink| nextnode
        newnode -->|llink| prenode
```

그림이 좀 이상하지만 newnode의 rlink, llink가 각각 pre와 next를 가리키고 있는 모습입니다.

이처럼 연결이 성공적으로 진행되었으면 pre와 next가 서로 가리키고 있던 포인터를 newnode를 가리키게 하면 됩니다.

2.2.4 메인 삽입

```
Node* node_in(Headnode* H, DataType* data, Node* prenode){ // 노드 생성 후 삽입, 종속 함수 2개 보유, 파라미터 : 헤드노드 포인터, 문자열 주소, 이전 노드의 주소
        Node* newnode = mknode(data); // 새로운 노드 할당
        if (!H->head) { // 리스트가 비어있을 경우
                node_in_first(H, newnode ,prenode); // 첫번째 노드로 만들기
        } else { // 리스트가 비어있지 않을 경우
                node_in_normal(H, newnode ,prenode); // 끼워넣기
        }
        return newnode;
}
```

위의 함수들을 활용하여 만든 최종 함수입니다. 파라미터로 헤드포인터의 주소, 문자열의 시작 주소, 이전 노드의 주소를 통해서 조작을 해 주려는 모습입니다.

mknode 함수를 이용해서 노드를 생성, 그 노드의 주소를 node형 포인터 newnode에 넘겨주고, H->head가 NULL을 가리킬때,
(if 조건문 안에 null이면 false가 되는데 ! 연산자로 반전시켜주어 참으로 만들어 줍니다.)
즉 노드가 비었을때 node_in_first 함수를 실행시키고
노드가 차있을때는 node_in_normal함수를 실행시킵니다.
그러고 난 뒤 새로 생성한 노드의 주소를 반환합니다.

2.3 삭제

삭제를 할 때의 문제를 쪼개어 생각하면 다음과 같습니다.
```

1. 연결 리스트에 노드가 하나 있을때의 삭제
2. 지우려는 Node가 head노드일때의 삭제
3. 보통 노드 삭제
4. 메인 삭제

먼저 문제를 쪼개어 상황들을 떠올려 보면 기능을 다음과 같이 나누어 개발해야 하는 것을 알 수 있습니다.

1. 메인 노드 삭제 함수
2. 노드가 비었을때의 삭제
    1. 비었을때의 삭제는 오류를 일으키므로 프로그램을 종료시키기
3. 지우려는 Node가 head노드일때의 삭제
    1. headnode를 다음 노드로 옮긴다
    2. 4. 보통 노드 삭제 를 진행한다
4. 보통 노드 삭제
    1. 지울 노드의 이전 노드가 지울 노드의 다음 노드를 가리키게 하고
    2. 지울 노드의 다음 노드가 지울 노드의 이전 노드를 가리키게 한뒤
    3. 문자열의 노드 삭제를

    문제 조건에 따라 파트별 함수로 나누었고 그것을 기능에 따라 또 세분화 하였습니다.

2.3.1 연결 리스트에 노드가 하나 있을때의 삭제

```
void node_out_one(Headnode* H, Node* delnode){ // 빈 링크드 리스트로 만들기, node_out에 종속적, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        free(delnode->data); // 지울 노드의 data가 가리키는 문자열의 동적 할당 해제
        free(delnode); // 생성했던 노드의 동적 할당 해제

        H->head = NULL; // head포인터가 NULL을 가리키게 하여 빈 링크드 리스트임을 표시
        H->count--; // 노드의 수 카운트 1개 감소
}
```

연결 리스트에 노드가 하나만 있을때에는 그 노드를 지우고 헤드노드의 head가 null을 가리키게 하면 되므로 제거하려는 노드의 데이터,

즉 동적 할당한 문자열을 동적할당 해제하고 그 다음에 동적 할당한 노드를 동적 할당 해제 한 후,

헤드노드의 head가 null을 가리키게하고 헤드노드의 count의 수를 1개 줄여 주었습니다.

2.3.2 지우려는 Node가 head노드일때의 삭제

```
void node_out_head(Headnode* H, Node* delnode){ // 헤드노드 옮기고 사이에 낀 노드 제거하기, node_out에 종속적, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        H->head = H->head->rlink; // 헤드포인터가 가리키는 노드의 오른쪽 노드를 헤드노드로
```

```
만든다.
        node_out_normal(H, delnode); // 지정한 노드 제거하기
}
```

지우려는 노드가 head노드일경우 링크드리스트 자체를 가리키지 못하는 대참사가 발생할 수 있습니다...

따라서 이에 따른 분기처리를 할 수 있는 함수를 제작하였습니다.

헤드노드의 head가 헤드노드의 head가 가리키는 노드의 rlink가 가리키는 노드,

즉 다음 노드를 가리키게 하고 일반 삭제 기능을 담당하는 함수node_out_normal을 자기가 받은 매개변수, 헤드노드의 주소와 지울 노드의 주소를 넘겨주어 삭제를 진행하였습니다.

2.3.3 보통 노드 삭제

```
void node_out_normal(Headnode* H, Node* delnode){ // 지정한 노드 제거하기, node_out, node_out_head에 종속적, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        delnode->llink->rlink = delnode->rlink; // 지우는 노드의 왼쪽 링크 즉, 이전 노드의 rlink가 지우는 노드의 다음 노드를 가리키게 한다.
        delnode->rlink->llink = delnode->llink; // 지우는 노드의 오른쪽 링크 즉, 다음 노드의 llink가 지우는 노드의 이전 노드를 가리키게 한다.

        free(delnode->data); // 지울 노드의 data가 가리키는 문자열의 동적 할당 해제
        free(delnode); // 생성했던 노드의 동적 할당 해제
        H->count--; // 노드의 수 카운트를 1개 감소시킨다.
}
```

위 코드는 위의 예외 처리들이 필요 없을때, 즉 나머지 상황일때의 삭제를 담당합니다. 2중 연결 리스트 노드의 일반적인 삭제는 다음과 같은 절차를 거쳐야 하는데

1. 지우려는 노드의 이전 노드의 rlink가 지우려는 노드의 다음 노드를 가리키게 하고

   delnode->llink->rlink = delnode->rlink;
   delnode의 llink는 (이전노드)입니다.
   즉, 그 노드의 rlink가 지우려는 노드의 rlink(다음 노드)를 가리키게 하고
2. 지우려는 노드의 다음 노드의 llink가 지우려는 노드의 이전 노드를 가리키게 하고

   delnode->rlink->llink = delnode->llink;
   delnode의 rlink는 (다음노드)입니다.
   즉, 그 노드의 llink가 지우려는 노드의 llink(이전 노드)를 가리키게 하고
3. 연결이 완료되면 지우는 노드의 데이터가 동적 할당된 문자열이므로 이 동적 할당을 해제
4. 이후 동적 할당된 지울 노드를 동적 할당 해제

위 과정을 마치면 노드의 제거가 안전하게 성공적으로 이루어집니다.

2.3.4 메인 삭제

```
void node_out(Headnode* H, Node* delnode){ // 지정한 노드 제거하기, 종속 함수 3개 보유, 파라미터 : 헤드노드 포인터, 지울 노드의 주소
        if(!H->head){ // 리스트가 비었으면
                printf("nodeout : 리스트가 비었습니다!\n"); // 메세지 출력
                return; // 함수 종료
        } else if(H->head == delnode && H->head == H->head->rlink){ //리스트에 원소가 하나이면
                printf("nodeout.one 지운 노드 : %s\n", delnode->data); // 메세지 출력
                node_out_one(H, delnode); // 빈 링크드 리스트로 만들기
                return; // 함수 종료
        } else if(H->head == delnode && !(H->head == H->head->rlink)){ // 지우는게 head 노드이면
                printf("nodeout.head 지운 노드 : %s\n", delnode->data); // 메세지 출력
                node_out_head(H, delnode); // 헤드 노드 옮기고 지정한 노드 제거하기
                return; // 함수 종료
        } else { // 그냥 제거해도 이상 없을때
                printf("nodeout.normal 지운 노드 : %s\n", delnode->data); // 메세지 출력
                node_out_normal(H, delnode); // 지정한 노드 제거하기
        return; // 함수 종료
        }
}
```

위 삭제를 모두 정의하고 메인 삭제 로직을 구성하였으며 맨 처음 if문은 빈 연결리스트에서 노드를 제거하려 하면 오류가 발생할 수 있으므로 그 예외 처리를 해 주었고

그다음 else if문은 리스트에 원소가 하나 있을때, 하나가 아니라면 지우려는 노드가 head노드일때, 그것도 아니라면 일반 노드 삭제를 진행하도록 로직을 작성하였으며, 각각 호출하는 함수에는 node_out이 받는 모든 매개변수를 넘겨준 상황입니다.

2.4 탐색

탐색 파트는 메인 함수 1개 그에 종속적인 함수 2개를 가지고 있습니다.

n만큼 건너뛰고 삭제후 다음 사람부터 시작하며 방향전환,
n만큼 건너뛰고 삭제후 다음 사람부터 시작하며 방향전환 과 같은 과정을 모든 리스트가 빌 때 까지 반복해야 하므로
먼저 문제를 쪼개어 상황들을 떠올려 보면 기능을 다음과 같이 나누어 개발해야 하는 것을 알 수 있습니다.

1. 메인 탐색 함수
2. 오른쪽으로 1칸 이동
3. 왼쪽으로 1칸 이동
   문제 조건에 따라 파트별 함수로 나누었고 그것을 기능에 따라 또 세분화 하였습니다.

이번 코드는 그리 길지 않으므로 한번에 보여드리겠습니다.

```
// 탐색
Node* node_search_int_L(Node* point){ // 왼쪽으로 한 칸 이동, node_serch에 종속, 파라미터 : 시작 노드의 주소
        point = point->llink; // point의 왼쪽을 point로 새로 지정
        return point; // point 반환
}
Node* node_search_int_R(Node* point){ // 오른쪽으로 한 칸 이동, node_serch에 종속, 파라미터 : 시작 노드의 주소
        point = point->rlink; // point의 왼쪽을 point로 새로 지정
        return point; // point 반환
}
Node* node_search(Headnode* H, Node* point, bool RL){ // 노드 지정한 횟수만큼 탐색 후 그 주소 반환, 종속 함수 2개 보유, 파라미터 : 헤드노드 포인터, 시작 노드의 주소, 회전옵션
        if(H->head == NULL){ // 노드가 비어있으면
                printf("nodeserch : 노드가 비었습니다!"); // 메세지 출력
                return NULL; // NULL 반환
        } else if(RL == false){ // 회전옵션이 false(오른쪽이면)
                return node_search_int_R(point); // 오른쪽으로 한칸 이동한 노드의 주소를 반환
        } else { // 회전옵션이 true(왼쪽이면)
                return node_search_int_L(point); // 왼쪽으로 한 칸 이동한 노드의 주소를 반환
        }
}
```

node_search 함수는 연결 리스트가 비어있는지 여부를 체크할 헤드노드의 포인터,
탐색 시작 위치 노드의 주소인 point, 어느 방향으로 탐색할지 결정하는 bool형 RL을 매개변수로 받았습니다.

1. 그리고 노드가 비어있을때는 노드가 비었습니다를 출력
2. 방향 옵션이 false일때는 node_search_int_R함수에 현재 위치를 넘겨주며 호출하고 그 함수의 반환값을 리턴
3. 방향 옵션이 true일때는 node_search_int_L함수에 현재 위치를 넘겨주며 호출하고 그 함수의 반환값을 리턴
   을 진행해 주었고 각각의 함수는 각각 point의 rlink를 새로운 point로 지정하고 그 값을 반환, 다른 함수는 llink를 같은 과정을 진행해 줌으로서 위 전체 로직은 회전 옵션에 따라 오른쪽이나 왼쪽으로 한칸씩 이동한 노드의 주소를 반환하는 함수임을 알 수 있습니다.

2.5 모든 노드 출력

```
// 전체 노드 출력
void print_all_node(Headnode* H){ // 연결 리스트의 모든 노드 상황 출력
        if (H->head == NULL) { // 리스트가 비었으면
                printf("allnode : 노드가 비었습니다!\n"); // 메세지 출력
                return; // 함수 종료
        } else { // 리스트가 비어있지 않으면
                Node* cursor = H->head; // 노드를 가리킬 수 있는 포인터를 선언 및 헤드 가리키기
                for(;cursor != NULL;){ // 커서가 널값이 아니면(무한루프)
                        printf("%s -> ", cursor->data); // 커서가 가리키는 데이터 출력
                        cursor = cursor->rlink; // 커서의 오른쪽 노드를 커서에 저장
                        if (cursor == H->head){ // 커서가 head노드를 가리키면
                                break; // 반복문 종료
                        }
                }
        printf("\n"); // 구문 구분용 개행문자 출력
        }
}
```

현재 연결 리스트에 존재하는 모든 노드를 출력하는 로직입니다.
헤드노드부터 순서대로 타고 넘어가면서 출력만 해주면 되므로 연결리스트의 헤드노드의 포인터만 매개변수로 받아 준 모습입니다.

만약 헤드노드의 head가 널포인터면 리스트가 빈 것이므로 노드가 비었음을 출력하고 함수를 종료하고

커서가 널값이 아니라면 (사실상 무한 루프)

head가 가지고 있는 주소값을 그대로 카피한 cursor(즉 head가 가리키고 있는 node의 주소 그 자체가 된다)Node형 포인터 cursor를 선언해주고
(이렇게 해주는 이유는 head를 직접 움직이면 head가 바뀌어 버리기 때문이다.)

모든 노드를 순차적으로 돌면서 그 노드의 데이터를 출력하고

cursor의 rlink가 head가 가지고 있는 주소값과 같다면 (다시 head를 만났다면)

반복문을 빠져나오고
구문 구분용 개행문자를 출력하고 함수를 종료합니다.

3. 메인 함수

```
// 메인 구현부
int main() {
	int N; // 건너뛸 횟수
	scanf("%d", &N); // 건너뛸 횟수 입력받기

	Headnode* H = mk_L_list(); // 새로운 링크드 리스트 생성
	Node* cursor = H->head; // 삽입 및 삭제용 커서 생성

	// 삽입부
	DataType name[50]; // 사용자가 입력하는 문자열의 크기
	for(;1;) { // (무한 루프)
		scanf("%s", name); // 이름,종료 입력받기
		if(!strcmp(name, "stop")) { // 입력값이 "stop"이면
			break; // 반복문 종료
		} else { // 입력값이 "stop"이 아니면
		cursor = node_in(H, name, cursor); // 입력값을 노드에 삽입하고, 노드 주소 커서에 넘겨주기
		}
	}

	bool RL = false; // 방향 확인 및 랜덤 지정용 RL 선언

	cursor = H->head; //커서 일단 head노드가 가리키는 노드로 초기화
	// 랜덤 지정부
	int randnum; // 랜덤 반복 횟수 randnum 선언
	print_all_node(H); // 입력된 노드들을 출력
	srand(time(NULL)); // 시드값을 시간으로 주기
	randnum = rand() % H->count; // 랜덤값을 연결 리스트의 길이 범위 내에서 사용할 것을 명시
	for (int i = 0; i < randnum; i++){ // 랜덤값 만큼 반복
		cursor = node_search(H, cursor, RL); // 노드를 회전옵션 방향으로 1칸 이동
	}
	printf("랜덤으로 지정된 head는 : %s\n", cursor->data); // 커서가 가리키는 랜덤 헤드를 출력
	H->head = cursor; // head를 진짜 랜덤으로 지정한 cursor로 변경

	// 삭제부
	while (H->head != NULL) { // 리스트가 비어있지 않을때까지
		// N만큼 이동
		for (int i = 0; i < N; i++) {
			cursor = node_search(H, cursor, RL); // 방향옵션쪽으로 1칸 이동
			printf("커서 : %s\n", cursor->data); // 이동한 커서의 위치
```

```
출력
                }
                printf("지운 노드 : %s\n 현재 노드의 수 : %d\n", cursor->data, H-
>count); // 지울 노드와 현재 노드의 수 출력
                Node* next_cursor = node_search(H, cursor, RL); // 다음 노드를
가리키는 포인터를 저장

                node_out(H, cursor); // 현재 노드를 삭제
                print_all_node(H); // 현재 모든 노드 출력
                cursor = next_cursor; // 다음 노드로 이동
                printf("다음 커서 : %s\n", cursor->data); // 다음 커서가 가리키는 내
용 출력
                printf("\n"); // 구분용 개행문자
                RL = !RL; // 방향옵션 반전 주기
        }
        // 모든 노드 삭제 후 동적할당한 헤드노드 해제
        free(H);

        return 0;
}
```

모든 것을 이 로직을 위해서였습니다. 각각을 순서대로 설명해 보겠습니다..

사용자 입력부

1. 사용자에게서 건너뛸 횟수 N을 입력받고
2. Headnode형 포인터 H에 링크드리스트 생성 함수 mk_L_list()를 통해 연결 리스트를 의미하는 헤드노드를 동적 생성하고
3. 삽입 및 삭제용 Node형 포인터 cursor를 선언하고
4. 사용자가 입력하는 문자열, 즉 이름의 최대 입력 크기를 50으로 지정해놓고 무한 루프 for문을 통해(가운데 조건식이 1이면 무조건 참) stop을 입력받으면 그대로 입력 받기를 종료하고

   아니라면 cursor = node_in(H, name, cursor);로직을 통해
   현재 커서와 헤드노드 실제 문자열 데이터인 name을 넘겨주고 동적으로 연결 리스트에 노드를 삽입한 후

   node_in은 삽입한 노드의 다음 주소를 반환하므로 그것을 다시 cursor에 넣어줌으로서 stop을 입력받기 전까지 노드를 순차적으로 이어가면서 생성하는 로직을 만들었으며

랜덤 생성부

1. 랜덤 반복 횟수를 저장할 randnum을 선언하고
2. 들어있는 노드를 순차적으로 출력하고

3. stdlib.h에 들어있는
   srand 함수(이 함수는 매개변수에 값을 넣으면 그 값을 바탕으로 rand함수를 실행하기전의 랜덤 시드 값을 결정합니다.)와
   time.h에 들어있는 time함수 (이 함수는 매개변수에 시간을 넣으면 그 시간을 출력하지만 NULL을 넣으면 현재 시간을 출력합니다.)를 통해 컴퓨터 시간에 따른 랜덤 시드를 결정하고
4. rand 함수를 통해 랜덤 값을 출력하고 연결 리스트의 길이 H->count로 모듈러 연산을 해 주어 0~연결 리스트의 길이 -1만큼의 값 내에서만 랜덤 값이 나오도록 설정해 주고
5. 랜덤 값만큼 오른쪽으로 노드를 순차적으로 가리키면서 그 노드에서부터 제거 로직을 실시합니다.

삭제부

1. 리스트가 비어잇지 않으면 계속 이 로직을 실행하는데
2. N만큼 건너뛰어야하므로 N만큼 노드를 1칸씩 이동하며 가리키는데
   node_search는 이동 완료한 노드의 주소를 반환하기 때문에 이 값을 cursor에 넣어가면서 계속 탐색하고
3. 한칸 더 넘어간 값을 next _cursor에 저장하고
4. 커서가 가리키는 노드를 node_out함수를 이용해 삭제한 후
5. 모든 노드를 출력한 뒤
6. next_cursor가 무엇인지 출력한 후
7. 방향 옵션 반전을 주고 위 로직을 반복한다.

와 같은 내용을 실시함으로서 문제 조건에 맞는 코드를 작성하였습니다...