자료구조(1) 시험대비 정리

2022-1, 박요한 교수님 (교재: C언어로 쉽게 풀어쓴 자료구조)

강의내용은 다른 교수님들과 진행순서가 달라서 다를 수 있음. 시험 출제방식은 중간고사 기준 단답식(과제나 PPT 내용의 코드를 가지고 빈칸 채우기나 해당 코드의 결과값 예측 등을 문제로 냄.)

자료구조와 알고리즘 / 1주-1

자료구조란?

- 자료구조는 말 그대로 Data Structure, 자료를 구조화 하는 것으로 특수 환경에서 자료의 정리를 하고자 하는 것임.
- 자료를 효율적으로 표현,저장 및 처리할 수 있도록 정리하는 것을 의미한다.

자료구조의 필요성

컴퓨터가 효율적으로 문제를 처리하기 위해선 문제를 정의하고 분석해서 그에 대한 최적의 프로그램 작성해야 함. (개념과 활용 능력이 필요하다)

> 일상생활에서 볼 수 있는 자료구조 형태
>
> - 그릇을 쌓아 보관하는 것: Stack(= 선입후출 방식)
> - 마트 계산대의 줄: Queue(= 선입선출 방식)
> - 버킷 리스트: List
> - 영어사전: 사전
> - 지도: Graph
> - 컴퓨터의 Directory 구조: Tree

프로그램은 자료구조+알고리즘으로 만들어진다.

알고리즘

컴퓨터로 문제를 풀기 위한 단계적인 절차

알고리즘의 조건

- 입력: 0개 이상 입력이 필요함

- 출력: 1개 이상의 출력이 필요함
- 명백성: 각 명령어의 의미는 모호하지 않고 명확해야 함
- 유한성: 한정된 수의 단계 후에는 반드시 종료되어야 함
- 유효성: 각 명령어들은 실행 가능한 연산이어야 함

알고리즘의 기술

- 영어/한국어 같은 자연어
- 흐름도(flow chart)
- 의사코드(pseudo-code)
- 프로그래밍 언어

과제 코드 (임의의 0~100 사이 정수 100개 만들어 배열 저장 후 100개의 점수 중 임의값 10개 선택해서 그중에 max값 찾기)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //난수 생성을 위해 시간 이용

#define MAX_ELEMENTS 100 //100개의 점수 받으므로 미리 정의
#define random_element 10 //10개 선별하므로 미리 정의

int scores[MAX_ELEMENT]; //= scores[100]
int random[random_element]; //= random[10]

void randomsc(){ //randomscore 부여
  int rd = 0;
  srand(time(NULL));
  for (rd = 0; rd < MAX_ELEMENT; rd++){ //100번 수행
    scores[rd] = rand() % 100; //scores 0부터 99까지 랜덤값 부여
  }
}

int main(void){
  randomsc(); //randomsc 함수 호출하여 랜덤값 부여한거 불러오기
  srand(time(NULL)); //현재시간을 이용한 난수생성
```

```
    int o;
    for (o = 0; o < random_element; o++){
      int rdmnum = rand() % 100; //scores에서 값 랜덤추출
      random[o] = scores[rdmnum]; // 10개의 값을 random에 저장
    }
    int i,  largest;
    largest =. random[0]; //일단 largest는 random[0]으로 둠
    for (i = 0; i < random_element; i++){
      if(random[i] = largest)
        //largest가 random[i]의 값보다 낮은 수를 가진 경우
        largest = random[i];
        //largest는 자신이 가진 수보다 더 컸던 random[i]를 취함.
    }

    printf("최댓값 = %d", largest);
    system("pause");
    return 0;
  }
```

자료구조의 개념

자료를 효율적으로 표현, 저장, 처리할 수 있도록 정리하는 것을 의미한다.

*data와 function을 묶어서 보되 둘을 각각 따로 확인하는 방식을 객체지향이라고 함. 객체지향 언어로는 JAVA, C# 등이 존재. 그리고 Function에 대해서 고민하는 것을 algorithm이라고 함.

추상데이터타입 (ADT) / 1주-2 연장선

abstract : A statement summarizing the important point of a text =텍스트의 중요부분을 강조, 정리하는 것 ex) abstract class, abstract function

추상화 시킬 때 중요한 것 -사용자 관점에서 보았을 때 중요한 것, 상위 level (not 구별 level.) ex)

```
//add function 만들 시
void ADD (int a, int b){
int c = a+b
}
```

자료형(Data Type)

자료형(data type): 데이터의 종류(data+연산). 정수나 실수, 문자열 등이 기초적인 자료형의 예. 데이터의 집합과 연산의 집합을 의미한다.

기초자료형에는 char(문자), int(정수형), float(실수), double(정수 및 소수부)이 존재하고 파생 자료형에는 배열(Array), 포인터가 존재함. 사용자 정의 자료형에는 구조체(Struct)와 공용체가 있다.

추상 데이터 타입(ADT)

데이터타입을 추상적(수학적)으로 정의한 것. 데이터나 연산이 무엇인가는 정의되나 어떻게 구현할 것인지는 정의되지 않는 것을 의미한다.

추상 데이터 타입의 정의

외부에서 내부 구조를 볼 수 없도록 디자인해야 프로그램이 안전해진다.

- 객체: 추상 데이터 타입에 속하는 객체가 정의된다.
- 연산: 이들 객체 사이의 연산이 정의된다. 해당 연산은 추상 데이터 타입과 외부를 연결 해 주는 인터페이스의 역할을 한다.

> Stack의 ADT

```
//객체: 0개 이상의 원소를 가지는 유한 선형 리스트
//연산:
▪ create(size) ::= 최대 크기가 size인 공백 스택을 생성한다. ▪ is_full(s)
::=
  if(스택의 원소수 == size) return TRUE; else return FALSE;
▪ is_empty(s) ::=
  if(스택의 원소수 == 0) return TRUE; else return FALSE;
▪ push(s, item) ::=
  if( is_full(s) ) return ERROR_STACKFULL; else 스택의 맨 위에 item
을 추가한다.
▪ pop(s) ::=
  if( is_empty(s) ) return ERROR_STACKEMPTY; else 스택의 맨 위의 원소
를 제거해서 반환한다.
▪ peek(s) ::=
  if( is_empty(s) ) return ERROR_STACKEMPTY;
else 스택의 맨 위의 원소를 제거하지 않고 반환한다.
```

순환(Recursion) 1차 / 2주-1

Recursion이란?

문제의 단순화를 시켜 해결하도록 하는 방식. 알고리즘이나 함수가 수행 도중 자기자신을 반복 호출해 문제를 해결하는 기법(수학적 귀납법), 가장 좋은 예시로 마트료시카 있음.

재귀호출(순환호출)

- 자기 자신을 호출하여 순환작업 수행하는 것
- 전체 문제를 한번에 해결하기 보다는 동일 유형의 하위 작업으로 분할, 작은 문제부터 해결하는 방법이 효율적일 경우에 사용.

recursion을 이용할 때 작업처리를 위해 존재하는 것이 두가지 있음.

- 하위 작업(Recursion step): 현재 수행 중인 작업의 하위단계. 즉 좀 더 작은 단위 작업을 의미.
- 베이스 케이스(Base case)

- 재귀호출의 과정 반복하다 보면 한 번에 해결할 수 있을 정도로 분할된 작업 단위가 충분히 작아지는 단계
- 더 이상 자신을 호출하여 문제의 난이도를 낮추지 않고도 해결할 수 있는 문제.

Recursion(=순환)과 Iteration(=반복) 비교

factorial 연산을 진행하는 코드를 작성하여 recursion과 iteration의 차이를 보고자 한다.

recursion으로 factorial 연산을 진행하는 코드를 작성하면

```
if(base case인지 확인)
  //맞을 시
  return basecase;
else //아니면
  return(문제를 간단히 만드는 작업) 후 recursion.
```

이런 구조로 코드가 돌아간다. 여기서 recursion 방식은 if else문을 의미하고 iteration은 for문을 의미한다.

과제는 factorial연산을 recursion과 iteration으로 각각 만들고 코드 진행시간을 측정하는 코드를 작성하는 것이다.

```
//recursion 방식
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //코드 진행시간 측정을 위해 time.h 사용

#define CLOCKS_PER_SEC 1000 //CLOCKS_PER_SEC을 정의한 이유는 clock_t로 시간 측정 시 측정단위가 ms이기 때문에 초로 변환하기 위해서 1000을 곱하려고 미리 정의했다.

long factorial(int n);

int main(void){
  int a = 0, result;
  clock_t start, finish;
```

```
  double duration;//소숫점단위 초도 측정하려고 duration을 double형으로 만들었다.
  printf("정수 입력:");
  scanf_s("%d", &a); //factorial(a)를 진행할 것.

  start = clock(); //시간측정 시작

  result = factorial(a);
  printf("%d!은 %d이다", a, result);
  finish = clock(); //시간측정 종료
  duration = (double) (finish - start)/CLOCKS_PER_SEC;

  printf("걸린 시간: %lf", duration);

  system("pause");
  return 0;
}

long factorial(int n){
  printf("factorial(%d)\n", n);

  if(n <= 1) return 1;
  else return n*factorial(n-1);
}
```

```
//iteration 방식
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //코드 진행시간 측정을 위해 time.h 사용

#define CLOCKS_PER_SEC 1000 //CLOCKS_PER_SEC을 정의한 이유는 clock_t로 시간 측정 시 측정단위가 ms이기 때문에 초로 변환하기 위해서 1000을 곱하려고 미리 정의했다.

long factorial(int n);

int main(void){
```

```
  int a = 0, result;
  clock_t start, finish;

  double duration;//소숫점단위 초도 측정하려고 duration을 double형으로 만들었다.
  printf("정수 입력:");
  scanf_s("%d", &a); //factorial(a)를 진행할 것.

  start = clock(); //시간측정 시작

  result = factorial(a);
  printf("%d!은 %d이다", a, result);
  finish = clock(); //시간측정 종료
  duration = (double) (finish - start)/CLOCKS_PER_SEC;

  printf("걸린 시간: %lf", duration);

  system("pause");
  return 0;
}

long factorial(int n){
  int x;
  int ftr = 1;
  for (x = 1; x <= n; x++){
    ftr *= x; //ftr에 x를 계속 곱해주는 방식. 1*2*3*... 처럼 값을 점점 올리면서 곱셈
연산을 하여 팩토리얼 진행.
  }
  printf("factorial(%d)", n);
  return (ftr);
}
```

이렇게 코드를 작성해서 비교를 해 보면 recursion은 일정 값 이상을 넣으면 Stack Overflow가 발생하지만 iteration에서는 Stack Overflow가 발생하지 않는다.

그 이유는 반복문(for)과 재귀함수(if)의 차이점 때문인데, iteration은 코드를 반복적으로 실행하는 것 그 자체이고 recursion은 하나의 함수가 자신을 다시 호출해 작업을 수행하는 방식이기 때문이다.

스택 메모리의 초과가 일어나는 이유는 iteration의 경우 스택 메모리를 일절 사용하지 않지만 recursion은 함수의 호출 시 매개변수, 지역변수, 리턴값, 종료 후 돌아가는 위치까지를 스택메모리에 저장해두기에 인풋값이 커지면 콜 스택이 지속적으로 쌓이고 쌓이다 보면 스택 메모리의 초과가 일어나는 것이다.

그럼에도 recursion을 이용하는 이유는 일반 반복문보다 변수의 사용을 줄일 수 있기 때문이다. 구조의 단순화가 진행되니 복잡한 문제도 직관적으로 볼 수 있기에 이를 이용하는 것이다.

순환(Recursion) 2차 / 2주-2

저번 내용에서 이어져 문제를 해결할 때 recursion과 iteration 중 문제 구현이 편한 방법이 존재할 것인데, 우리는 이를 해결할 때 어떤 방식을 이용해서 해결을 할 것인지 잘 생각을 해볼 필요가 있다.

예를 들어 거듭제곱의 값을 구하는 방법은 순환(Recursion)이 더 효율적인 부분이 된다. 아래의 코드를 참고하면 이를 알 수 있는데

```
//Recursion방식
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //clock_t로 동작시간 측정

#define CLOCKS_PER_SEC // 측정단위시간인 ms를 초로 변경할 때 이용할 1000을 미리 정의

double power_recur(int x, int inp) {
  if (inp == 0) return 1;
  else if ((inp % 2) == 0) //짝수일 경우(2로 나눈 나머지 0)
    return power_recur(x * x, inp / 2);
  else return x * power_recur(x * x (inp - 1) / 2);
}

int main(void) {
  int x_input = 0;
  int inp_input = 0;

  clock_t start, finish;
  double duration;
  unsigned int result_recur;
```

```
  printf("밑 숫자 입력: ");
  scanf_s("%d", &x_input);
  printf("지수 입력: ");
  scanf_s("%d", &inp_input);

  start = clock();

  result_recur = power_recur(x_input, inp_input);
  printf("%d의 %d제곱은 %d입니다.\n", x_input, inp_input, result_recur);
  finish = clock();

  duration = (double) (finish - start) / CLOCKS_PER_SEC;
  printf("Recursion으로 걸린 시간: %lf", duration);

  system("pause");
  return 0;
}
```

```
//Iteration 방식
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //clock_t로 동작시간 측정

#define CLOCKS_PER_SEC // 측정단위시간인 ms를 초로 변경할 때 이용할 1000을 미리 정의

double power_iter (int x, int inp) {
  int pow = 0;
  double result = 1.0;
  for (pow = 0; pow < inp; pow++){
    result *= x;
  }
  return result;
}
```

```c
int main(void)
{
  int x_input = 0;
  int inp_input = 0;

  clock_t start, finish;
  double duration;
  unsigned int result_iter;

  printf("밑 숫자 입력: ");
  scanf_s("%d", &x_input);
  printf("지수 입력: ");
  scanf_s("%d", &inp_input);

  start = clock();

  result_iter = power_iter(x_input, inp_input);
  printf("%d의 %d제곱은 %d입니다.\n", x_input, inp_input, result_iter);
  finish = clock();

  duration = (double) (finish - start) / CLOCKS_PER_SEC;
  printf("Iteration으로 걸린 시간: %lf", duration);

  system("pause");
  return 0;
}
```

여기서 recursion의 코드와 iteration의 코드를 비교 해 보면, recursion의 경우 거듭제곱을 진행하기 위해 만약 2^10을 진행하고자 하면 10제곱을 2로 나누며 감산시켜 연산을 동시에 진행하기 최적화되도록 만들고 횟수도 획기적으로 줄인다. 그러나 iteration은 2^10을 시행하려면 2에 2를 9번이나 곱해야 하는 코드가 작성되는데 이를 보고 비교해 보면 결국 원하는 값을 도출하는 것은 맞으나, 그 값을 도출하는 과정의 간략화가 어떻게 되는지가 다르다.

배열(Array), 구조체(Structure), 포인터(Pointer) / 3주-1

자료구조 강의 진행 중 마지막 C언어 복습.

배열(Array)

- 같은 자료형을 가진 자료들을 나열하여 메모리에 연속으로 저장해 만든 자료들의 그룹
- 인덱스(index)
  - 배열의 요소를 간단히 구별하기 위해 사용하는 번호
  - C에서 인덱스는 0부터 시작한다(그래서 배열은 0번부터 시작하는 것)

> Array를 사용하는 이유
>
> : int a, int b, int c, int d 등을 독립사용하게 되면 불편한 점이 존재한다. (4개 int값의 합이나 전체 출력 등..) 이런 불편한점을 개선하기 위해 a,b,c,d라는 명칭을 주고 int A[4] (= 배열형태)를 만들어서 사용하는 것이다.

배열의 선언

(1)자료형 (2)배열이름 (3) [배열 요소의 개수]; 의 구조로 작성한다.

이 구조를 따라 작성하면 int arr[6];의 구조를 만든다는 말이다.

배열 요소는 모두 자료형이 같아야 하며 배열 요소의 자료형은 곧 배열의 자료형이 된다. 변수 이름과 같은 규칙으로 배열이름을 정해야 하고, 대괄호를 통해 요소의 개수를 표시하는데 요소의 개수는 배열의 크기를 의미한다. 배열을 선언하면 메모리에 배열에 대한 공간이 할당되고 그 크기는 '자료형에 대한 메모리 할당 크기 x 배열 요소의 개수'이다.

> 자료형이 같은 여러가지 자료들을 배열로 묶기
>
> Ex) int list1, list2, list3, list4, list5, list6 -> int list[6]
>
> list1~list6까지가 배열의 요소가 되었고 인덱스번호는 6개의 값이니 0부터 5까지로 할당된다. 배열이 가지는 주소의 경우 물리/논리적으로 붙어있다.

1차원 배열과 다차원 배열

배열은 1차원 배열부터 2차원 이상의 배열 둘 다 존재한다. 1차원 배열은 배열의 선언에서 보여준 형태가 1차원, 선형으로 표현되는 배열을 의미한다.

다차원 배열은 앞으로 쓰일 일이 많은데(Tree나 Graph 등..) 다차원 배열의 선언법은 이러하다.

- 배열의 차수만큼 [배열크기]항목을 추가
- 2차원 배열 선언방식 - 자료형 배열이름 [배열크기(행/세로)] [배열크기(열/가로)]
  - 이에 따르면 int list[3] [5]; 는 세로 3칸, 가로 5칸 총 데이터 15개의 사이즈를 가지는 배열이 만들어진다.
- 다차원 배열의 초기화(여기서 초기화는 0이 아니라 초기값 지정을 의미)
  - 초기값의 지정형태는 다차원 배열이 '배열의 배열'이라는 것을 생각하여 초기값을 구분하여 지정하거나, 1차원 배열처럼 초기값 리스트를 지정하여 순서대로 배열요소의 초기값으로 설정
  - 2차원 배열의 초기화 방식
    1. Int i [2] [3] = {{1,2,3}, {4,5,6}}; (열 맞춤 방식)
    2. Int i [2] [3] = {1,2,3,4,5,6} (이렇게 써도 다 채울 수 있다. 위와 같은 원리로 작동하기 때문.)

       작성은 편한대로 하면 되는데, 그 이유는 실무에 들어가서 쓸 일이 있을땐 다 0으로 채우기 때문.

포인터

직접 해보지 않고선 이해하기 힘들다. 어디서 무엇을 가리키고 데이터가 어떻게 이동하는 지 볼 줄 알아야 한다.

포인터의 개념

- 변수의 메모리 주소값
- 포인터 변수
  - 주소값을 저장하는 특별한 변수(= 주소값 담는 변수)
  - 포인터 변수가 어떤 변수의 주소를 저장하고 있다는 것은 곧 포인터 변수가 그 변수를 가리키고 있다(= 포인트하고 있다)는 의미이다.
  - 포인터 변수를 이용해 연결된 주소의 변수 영역을 액세스한다.(= 주소값을 통해서 그 위치에 있는 데이터를 쓰고자 하는 것.)

- 포인터변수는 간단히 말해 포인터라고 부른다.

  *동적 할당된 Matrix를 만들면 주소를 저장 해 두고 주소값을 통해 Matrix의 해당 위치의 값으로 algorithm을 사용한다. 나중에는 분명 이중포인터를 사용하는 경우도 있을 것이다.(ex. delete) 다만, 대체로 기본 포인터만 이용하곤 한다.

- 포인터 주소의 사이즈는 운영체제의 사이즈에 따라 갈린다.

포인터의 선언

- 선언형식: (1)자료형 *(2)포인터명

1. 포인터 자체의 자료형이 아닌 포인터에 저장할 주소에 있는 일반 변수의 자료형을 적어야 한다. (만약 integer을 가리키고자 하면 int를, character을 가리키고자 하면 char을 써야한다.)
2. 일반 변수 이름과 구분이 가능하도록 앞에 *을 붙여 포인터임을 나타낸다.

> 포인터의 사용원리
>
> ```
> char a = 'a';
> char *p;
> p = &a; //(& = ancher. &는 a의 주소를 넣는다는 의미이다.)
> ```
>
> 포인터(= 주소를 담을 수 있는 변수)에 값을 부여할 땐 '&(ancher)변수명' 으로 쓰는데 &변수명은 해당 변수명을 가진 데이터가 있는 주소를 넣는다는 말이고, 그 데이터의 주소는 Compiler 단위에서 자동지정한다.
>
> 가리키는 내용 변경
>
> *연산자를 사용해 변경한다.
>
> ```
> char a = 'a';
> char *p;
> p = &a; //(& = ancher. &는 a의 주소를 넣는다는 의미이다.)
> *p = '200';
> ```
>
> printf p를 하면 p에 담긴 주소가 나올 것이고 *p를 하면 포인터가 가리키는 주소의 데이터를 표시할 것이다. 지금 경우에는 *p = ‘200’을 해서 p가 가리키는 주소의 데이터을 200으로 바꾼 경우에 해당한다.

포인터 연산

포인터 연산자

- 주소 연산자: &
  - 변수의 주소를 얻기 위해서 사용한다.
  - 형식: 포인터 = &변수;

    예시)

```
int i = 10;
int *ptr;

ptr = &i //ptr이 int i의 주소를 가지게 한다(= i를 가리킨다)
```

과제: 포인터를 이용하여 reverse함수 구현(list가 받은 문자열을 반대로 뒤집어주는 함수 구현하기)

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

#define limit 100

void reverse(char *list, char *list2) {
  int i;
  int len = strlen(list); //len변수의 크기를 strlen(list)를 통해 list의 길이와 동일해지도록 변경
  for (i = 0; i < len; i++){
    *(list2 + i) = *(list + len - 1 - i); //list2의 i번째에 list의 맨 마지막 값 넣기. (-1이 들어간 이유는 문자열의 경우 맨 마지막엔 NULL이 들어가야 쓰레기값을 표시하지 않고 정상 작동하기 때문.)
  }
  return;
}

int main(void) {
```

```
  char s1[limit] = { 0 }, s2[limit] = { 0 }; //혹시 모를 오류에 대비하여 배열을 미리 0으로 초기화를 해 둔다.
  char *s1p;
  char *s2p;
  int i;

  printf("%X\n", s1); //이 줄은 array s1의 주소를 보고자 이용함.
  printf("문자열 입력: ");
  scanf_s("%s", s1, limit-1); //보안 문제 때문에 사용하는 scanf_s의 경우 배열의 사이즈를 지정해 주지 않으면 오류가 발생한다. 그리고 위에서 설명 한 대로 -1을 사용하지 않고 사이즈를 전부 끌어다 쓰면 쓰레기 값을 만날 수도 있다.

  s1p = s1;
  s2p = s2;

  reverse(s1p, s2p);
  printf("역순문자열: %s \n", s2);

  system("pause");
  return 0;
}
```

배열(Array), 구조체(Structure), 포인터(Pointer) 2차시 / 3주-2

포인터와 배열은 거의 비슷하게 동작하나 분명 다른 것이다. 항상 이 점은 유의해야 한다.

구조체(Structure)

타입이 다른 데이터를 하나로 묶는 방법. (배열은 타입이 같은 데이터들을 하나로 묶는 방법이었다. 타입이 다른 것도 묶을 수 있다는게 배열과 구조체의 차이점)

구조체의 선언

- 구조체 이름과 자료형, 데이터 항목으로 구성된다.
  - 구조체 이름: 구조체로 정의하는 새로운 자료형의 이름
  - 항목: 구조체를 구성하는 내부 변수들의 이름
    - 구조체의 항목은 배열의 각 배열요소에 해당한다.
    - 배열요소는 모두 같은 자료형으로 되있어서 배열요소에 대한 선언 없이도 사용이 가능하다.
    - 구조체에서는 각 항목이 다른 자료형을 가질 수도 있기에 항목별로 자료형과 항목명(= 변수명)을 선언해야 한다.

구조체형의 선언과 사용형식

```
//구조체형의 선언
struct 구조체형_이름 {
  자료형 항목(내부변수) 1;
  자료형 항목 2;
  자료형 항목 3;
  ...
  자료형 항목 n;
}

//구조체형의 사용 형식
struct 구조체_이름 구조체_변수_이름;
```

구조체의 사용 단계

1. 구조체형의 선언: 구조체의 내부 구조를 선언한다.
2. 구조체 변수 선언: 구조체형에 따른 변수를 선언한다.
3. 구조체 변수의 사용: 내부 항목에 데이터를 저장하고 사용한다.

> 구조체 변수의 선언방법
>
> ```
> //가장 일반적인 방법
> struct employee {
>   char name[10];
>   int year;
>   int pay;
> ```

```
};
struct employee Lee; //구조체형 선언 후 구조체변수 선언.

//간편하게 쓰는 경우
struct employee {
  char name[10];
  int year;
  int pay;
} Lee; //구조체형과 구조체 변수를 연결해 선언.
```

> 구조체는 논리적으로 붙어있을 뿐만 아니라 물리적으로도 위치가 붙어있다. 그리고 할당공간은 fix된 상태로 유지된다.
>
> 만일 남는 공간이 존재할 시 그냥 노는 공간으로 남는다.

구조체와 포인터

구조체는 포인터를 사용해서 접근을 할 수도 있다. 위 코드에서 연장하여 struct employee *Sptr = Lee 를 사용하면 아래처럼 이용이 가능하다.

```
//구조체 포인터를 이용한 데이터 항목의 지정법

//구조체 포인터의 화살표 연산자 사용
Sptr->name = "susan";
Sptr->year = 2014;
Sptr->pay = 4300;

//구조체 포인터의 참조 연산자 사용
(*Sptr).name = "susan";
(*Sptr).year = 2014;
(*Sptr).pay = 4300; //참조 연산자를 이용할 때는 꼭 괄호 써줘야 한다.

//1번 방식과 2번 방식은 동일한 결과를 도출한다.
```

구조체와 함수

구조체를 함수의 인수로 전달하는 경우

- 구조체의 사본이 함수로 전달되게 된다.(Parameter로 쓸 때는 원본이 이용되는 것이 아니라 사본이 이용되는 것이다. (Swallow copy, 얕은 복사))
- 만약 구조체의 크기가 크면 그만큼 시간이나 메모리가 소요된다.(= 단점은 메모리 효율성)

```
//함수를 인수로 전달하는 case 예시
int equal(struct student s1, struct student s2) {
  if( strcmp(s1.name, s2.name) == 0 )
    return 1;
  else
    return 0;
}
```

구조체의 포인터를 함수의 인수로 전달하는 경우

- 시간과 공간을 절약할 수 있다.
- 원본 훼손의 가능성이 있다.

  (포인터를 이용할 시 주소값을 이용해 값을 불러와 원본을 이용하게 되고, 이렇게 되면 시간과 데이터 공간의 절약이 가능하나 원본내용을 조작이 가능해지는 위험성이 존재한다.)

  포인터를 사용하는 매개 변수는 const를 사용한다. const를 쓰는 이유는 원본데이터를 수정할 수 없도록 하기 위해서이다. const를 사용하게 되면 원본데이터의 훼손을 막을 수 있고, 이렇게 써야 유지보수가 용이하다.

동적 메모리 할당(Dynamic Memory Allocation)

- 프로그램의 실행 도중에 메모리를 할당 받는 것이다.
- 필요한 만큼만 할당을 받고 또 필요한 때에 사용 후 반납하는 방식이다.
- 메모리를 매우 효율적으로 이용할 수 있다.

> Static(정적) 과 Dynamic(동적)의 차이
>
> Static의 경우 Compile할때 Stack 에서 결정하고 Dynamic의 경우 Runtime(실 작동중일 때) 에 결정한다.

> Static의 예시
>
> - Int a: 사용하든 사용하지 않든 메모리를 할당받아 영역을 생성한다.
> - 구조체(Structure): 마찬가지로 사용하지 않아도 영역을 생성한다.
>
> Static 방식의 단점
>
> - 관리 문제(= 관리 이슈)

동적 메모리의 할당방식

```
int main(void) {
  int *pi;
  pi = (int *)malloc(sizeof(int)); //동적 메모리 할당
  //동적 메모리 사용
  free(pi); //사용 후 메모리 반납
}
```

malloc은 memory allocation의 줄임말이다.

Struct에 동적 메모리를 할당할 때는 int가 아니라 Struct의 이름을 넣는다. 그리고 동적메모리는 따로 이름이 없기 때문에 이름을 만들어야 한다.

과제: 구조체를 이용한 주소록 구현

```
#pragma warning(disable:4996) //VS에서는 _s를 붙여야 scanf가 사용 가능하도록 경고를 띄우지 않는데 이를 무시하는 코드 선입력.

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

int count = 0; //count를 여기저기서 이용해야 하여 전역변수로 생성.

#define namesize 10
```

```
struct addressBook { //주소록 구조체
  char name[namesize]; //이름을 받고 저장할 내부변수 name
  char phone[15]; //15자리의 전화번호를 받고 저장할 내부변수 phone
  char addr[50]; //50자리의 주소지를 받고 저장할 내부변수 addr
  char birthday[8]; //8자리의 생일 받고 저장할 내부변수 birthday
}aB_test[50]; //구조체 변수 선언하고 주소록의 저장 가능 총량을 50으로 함.

void list(void) //주소록 출력
{
  int i;
  if (count == 0) printf("등록된 주소록 목록이 존재하지 않습니다 \n\n"); //전역변수
count의 값이 0이면 주소록에 저장된 것이 없는 것이므로 printf 출력
  else{ //만약 주소록에 값이 존재할 경우
    printf("\n 이름  \t전화번호  \t주소  \t생일  \n");
    printf("============================\n");
    for (i = 0; i < count; i++){
      printf("%s\t. %s\t  %s\t  %s\t", aB_test[i].name,
aB_test[i].phone, aB_test[i].addr, aB_test[i].birthday);
    }// 주소록에 존재하는 전체 데이터를 양식에 맞추어 출력하도록 만들어준다.
  }
}

void append(void) //주소록에 입력
{
  count; //전역변수 count를 불러 와 주고
  printf("추가할 이름: "); scanf("%s", aB_test[count].name);
  printf("추가할 번호: "); scanf("%s", aB_test[count].phone);
  printf("추가할 주소: "); scanf("%s", aB_test[count].addr);
  printf("추가할 생일: "); scanf("%s", aB_test[count].birthday);
  count++; //값 추가가 끝난 뒤 전역변수에 1을 더해준다.
}

void search(void) //주소록에서 특정 데이터 찾기
{
  char name[namesize];
```

```
  int i;
  printf("검색할 이름: ");
  scanf("%s", name); //이름으로 데이터를 찾는 방식을 채택했다.

  for (i = 0; i < count; i++) {
    if (strcmp(name, aB_test[i].name) == 0) { //입력한 이름과 동일한 이름을 가
진 데이터가 존재하면
      printf("이름: %s\n", aB_test[i].name);
      printf("전화번호: %s\n", aB_test[i].phone);
      printf("주소지: %s\n", aB_test[i].addr);
      printf("생일: %s\n", aB_test[i].birthday);
    } //값을 출력 해 준다.
    else if (i == count)
      printf("%s에 해당하는 이름을 가진 정보는 존재하지 않습니다.\n\n", name);
  }
}

void del(void)
{
  char name[namesize];
  int i, o;
  printf("삭제할 이름: ");
  scanf("%s", name);

  for (i = 0; i < count; i++){
    if(strcmp(name, aB_test[i].name) == 0) {
      for (o = i; o+1 < 0; o++){
        strcpy(aB_test[o].name, aB_test[o+1].name);
        strcpy(aB_test[o].name, aB_test[o+1].name);
        strcpy(aB_test[o].name, aB_test[o+1].name);
        strcpy(aB_test[o].name, aB_test[o+1].name);
      }
      printf("삭제 완료. \n");
      count--;
    }
    else if (i == count)
```

```
        printf("%s에 해당하는 이름을 가진 정보는 존재하지 않습니다. \n\n");
    }
}

int main(void) {
    char choice;
    do {
        printf("\n\n========================\n");
        printf("---------- 주소록 메뉴 ----------\n");
        printf(" 1)목록보기 \t2)추가 \t3)검색 \t4)삭제 \t5)종료 \n");
        printf("========================\n");

        scanf("%c", &choice); getchar();
        switch (choice) {
        case '1':
            list(); break;
        case '2':
            append(); getchar(); break;
        case '3':
            search(); getchar(); break;
        case '4':
            del(); getchar(); break;
        case '5':
            printf("\n .\n"); return;
        default:
            printf(".\n");
        };
    }while (choice != '5')
      system("pause");
      return 0;
}
```

연결리스트(Linked List) / 4주-1

연결리스트의 종류로는 단방향 연결 리스트(Singly Linked List), 순환연결리스트(Circular Linked List), 양방향 연결 리스트(Doubly Linked List)가 있다.

List의 반대는 Set. (List: 순서가 있는 자료형 / Set: 순서가 없는 자료형)

Array vs List

Array의 장점

- 접근속도가 일정하다.(= index가 존재한다.)
- Locality(=집약성/Cache Memory에 데이터를 넣을 수 있다.)

Array의 단점

- 고정된 크기(데이터의 입/출력이 잦은 경우에는 매우 비효율적)
- Sequencial 구조를 유지하기 위해서 데이터의 추가 및 삭제 시 기존 데이터가 이동을 많이 하게 된다.

Linked List의 장점

- 데이터의 입력과 삭제가 유용하다.(중간 데이터를 삭제하면 그 전 데이터에 다음 데이터의 주소를 덮어씌운다.)
- 데이터의 이동과 삭제, 추가에 큰 제약이 없다.(= 유동성이 좋다)
- 작업의 수행 도중에 크기를 늘리거나 줄일 수가 있다.

Linked List의 단점

- 데이터를 읽어올 때 모든 노드를 읽어와야 원하는 데이터를 불러올 수 있다(Array와 다르게 접근성이 느리다.)
- Locality(논리적으로는 Sequencial 구조이지만 물리적으로 데이터는 서로 떨어져있어 Cache Memory에 담기지 않는다. 물론 이를 감안하여 별도의 기법도 존재하긴 한다.)
- Overhead(= 주소값/array의 주소는 뻔해서 따로 저장할 필요가 없지만(물리적으로도 붙어 있기 때문인 것으로 추정.), 이건 다음 노드의 주소를 꼭 알아야 해서 필요하다.)

두 가지는 각각 프로그램 기법으로 배열을 이용한 구현 / 포인터를 이용한 구현 으로 이용한다.

> 리스트 ADT(추상데이터타입/AbstractDataType)
>
> ```
> • 객체: n개의 element형으로 구성된 순서 있는 모임. 임의의 element type으로 만든다.
> • 연산:
> insert(list, pos, item) ::= pos 위치에 요소를 추가한다.
> insert_last(list, item) ::= 맨 끝에 요소를 추가한다.
> insert_first(list, item) ::= 맨 처음에 요소를 추가한다. delete(list, pos) ::= pos 위치의 요소를 제거한다. clear(list) ::= 리스트의 모든 요소를 제거한다. get_entry(list, pos) ::= pos 위치의 요소를 반환한다.
> get_length(list) ::= 리스트의 길이를 구한다. is_empty(list) ::= 리스트가 비었는지를 검사한다. is_full(list) ::= 리스트가 꽉 찼는지를 검사한다.
> print_list(list) ::= 리스트의 모든 요소를 표시한다.
> ```

배열로 구현된 리스트

배열을 이용하여 리스트를 구현하면 순차적인 메모리 공간(물리적)이 할당되므로, 이것을 리스트의 순차적 표현(Sequential Representation)이라고 한다.

선형 리스트에서 원소 삽입: 선형리스트 중간에 원소가 삽입되면 그 이후 원소들은 한 자리씩 자리를 뒤로 이동해 물리적 순서를 논리적 순서와 일치시킨다.

원소 삽입방법

- 원소를 삽입할 빈 자리 만들기
  - 삽입할 자리 이후의 원소들을 한 자리씩 뒤로 자리 이동
- 준비한 빈 자리에 원소 삽입하기

선형 리스트에서 원소 삭제: 선형리스트 중간에서 원소가 삭제되면 그 이후의 원소들은 한 자리씩 자리를 앞으로 이동하여 물리적 순서를 논리적 순서와 일치시킨다

원소 삭제방법

- 원소 삭제하기
- 삭제한 빈 자리를 채우기

- 삭제한 자리 이후의 원소들을 한 자리씩 앞으로 자리 이동

과제: 배열을 사용한 리스트 구현하기

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>

typedef struct {
   int no; // 현재 index 갯수
   int size; // 배열의 전체 크기
   int *array; // 배열
}ArrayListType; //ArrayListType 구조체 형식 선언

void create(ArrayListType *L) { //*array(array 배열)의 메모리를 동적할당 하는 함수
   printf("사용할 배열의 크기를 입력하시오 : ");
   scanf("%d", &L->size);
   L->array = (int *)malloc(sizeof(int) * L->size); // 사용자에게 입력받은 size를 곱하기 int만큼 메모리 할당
}

void init(ArrayListType* L) { // array 배열의 index 크기를 저장하는 no 변수를 초기화하는 함수
   L->no = 0;
}

int is_full(ArrayListType* L) { // 현재 배열의 최대 크기와 현재 index의 크기가 같은지 확인하는 함수
   return L->no == L->size;
}

void insert(ArrayListType* L, int pos, int item) // pos:값을 넣을 index 위치값 item:배열에 넣을 값을 받아 배열에 입력하는 함수
{
```

```
    if (!(is_full(L)) && (pos >= 0) && (pos <= L->no)) { // 배열이 꽉찬 상태
인지, pos 값이 잘못되지 않았는지 확인
        for (int i = (L->no - 1); i >= pos; i--)
            L->array[i + 1] = L->array[i]; // 값을 넣을 위치 pos 값을 기준으로 뒤
에있는 값들을 한칸씩 뒤로 이동
        L->array[pos] = item; // pos 위치에 item을 입력
        L->no++; // 배열의 index 개수인 no에 1 추가
    }
}

void error(char* message) { // 입력받은 meesage를 출력하고 프로그램을 종료하는 함수
    fprintf(stderr, "%s\n", message);
    exit(1);
}

void insert_last(ArrayListType *L, int item) // 현재 배열의 index 마지막에 값
을 입력하는 함수
{
    if (L->no >= L->size) { // index값이 배열의 최대크기 size값보다 큰지 확인
        error("리스트 오버플로우");
    }
    L->array[L->no++] = item; // 현재 마지막 index 값:no 에 원소 값 입력
}


void print_list(ArrayListType *L) // 현재 배열에 저장된 모든 값을 출력하는 함수
{
    for (int i = 0; i < L->no; i++) // 0 부터 최대 index값 n-1 까지 출력
        printf("%d -> ", L->array[i]);
    printf("\n");
}

void delete(ArrayListType* L, int pos) // pos index 값을 받아서 그 자리의 원소
를 삭제하는 함수
{
    int item;
```

```
    if (pos < 0 || pos >= L->no) // 오류: pos 값이 음수이거나 현재 마지막 index값
보다 클때
        error("위치 오류");
    item = L->array[pos];
    for (int i = pos; i < (L->no - 1); i++) // pos값을 기준으로 원소들을 한칸씩
덮어쓰기
        L->array[i] = L->array[i + 1];
    L->no--;
    return item;
}

int main(void) {
    ArrayListType list; // ArrayListType struct의 list를 선언

    init(&list);
    create(&list);
    insert(&list, 0, 10); print_list(&list);
    insert(&list, 0, 20); print_list(&list);
    insert(&list, 0, 30); print_list(&list);
    insert_last(&list, 40); print_list(&list);
    delete(&list, 0); print_list(&list);

    return 0;
}
```

단순연결리스트(Singly Linked List) / 4주-2

연결리스트는 자료구조에서 활용성이 높기 때문에 무조건적으로 이해를 할 필요가 있다.

리스트의 구현 방법

배열: 중간에 공간을 비워둘 수 없다(물리적으로도 붙어있는 array의 특성 때문)

연결리스트: 각 노드는 data부분과 link부분을 통해 data를 가지고 link로 다음 노드를 가리키기에 물리적으로는 데이터끼리 붙어있지 않다.

연결 리스트(LinkedList)

- 리스트의 항목들을 노드(node)라고 하는 곳에 분산저장함
- 노드는 데이터 필드와 링크 필드로 구성한다
  - 데이터 필드 - 리스트의 원소, 즉 데이터값을 저장하는 곳
  - 링크 필드 - 다른 노드의 주소값을 저장하는 장소 (= 포인터, 이걸로 다음 노드와 연결시킴 (singly))

헤드 포인터와 노드의 생성

LinkedList는 병렬적 수행을 실시하는데 1번은 2번의 주소를 가지고 2번은 3번의 주소를 가진다. 이 구조에서 1번의 주소를 가지는 것은 보이지가 않는데, 여기서 1번의 주소를 가지는 것을 head라고 한다. Linked list의 parameter은 head부터 시작한다(ADT의 시작은 head). 그 이유는 head가 1번 데이터 주소를 가지기 때문이다. 그리고 Linked List를 만들면 끝부분도 표현을 해 주어야 하는데 끝이 되는 맨 마지막 노드는 NULL을 가져야 한다. 이렇게 구성을 하면 Linked List가 완성된다.

단순연결리스트(Singly LinkedList)는 단방향으로만 읽어지는데 이전 데이터를 읽기 위해서는 Doubly LinkedList를 사용해야 그러한 작업이 가능해진다.

연결 리스트의 종류

단방향으로만 진행되는 Singly LinkedList(단순 연결 리스트), 마지막 노드가 첫 번째 노드를 가리키는 구조를 가지고 NULL을 가지지 않는 Circular LinkedList(원형 연결 리스트), 노드가 앞 뒤 노드의 주소를 가지고 있는 구조인 Doubled LinkedList(이중 연결 리스트)

노드의 정의 방법

```
typedef int element;

typedef struct ListNode { //노드 타입은 구조체로 정의한다
  element data; //type define으로 정의한 int형의 element인 data
  struct ListNode *link; //ListNode의 주소를 가지는 *link
}ListNode;
```

포인터의 경우 int *p는 포인터가 가리키는 datatype을 가리키기 위해 int를 붙이는 것이었고, 리스트의 경우 구조체인 ListNode의 주소를 가리키기 위해 struct ListNode라고 붙여놓는 것이다.

리스트/노드의 생성

```
//리스트 생성
//첫 번째
ListNode *head = NULL;

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

head->data = 10;
head->link = NULL; //뒤에 데이터가 없으니 링크에 NULL을 넣는다.

//두 번째
ListNode *p;
p = (ListNdoe *)malloc(sizeof(ListNode));

p->data = 20;
p->link = NULL;

//노드의 연결
head->link = p;
```

> Singly LinkedList의 ADT
>
> - 객체: n개의 element형으로 구성된 순서 있는 모임
> - 연산:
>
> insert(list, pos, item) ::= pos 위치에 요소를 추가한다.
> insert_last(list, item) ::= 맨 끝에 요소를 추가한다.
> insert_first(list, item) ::= 맨 처음에 요소를 추가한다.
> delete_first(list) ::= 맨 처음 요소를 제거한다. delete(list, pos) ::= pos 위치의 요소를 제거한다. is_empty(list) ::= 리스트가 비었는지를 검사한다. is_full(list) ::= 리스트가 꽉 찼는지를 검사한다. print_list(list) ::= 리스트의 모든 요소를 표시한다.

단순연결리스트의 다양한 연산

삽입연산(insert_first)

```
ListNode* insert_first(ListNode* head, int value) {
  ListNode* p = (ListNode*)malloc(sizeof(ListNode));
  p->data = value;
  p->link = head;
  head = p;
  return head;
}
```

출력연산(print_list)

```
void print_list(ListNode* head){
  for(ListNode*p = head; p != NULL; p = p->link)
    printf("%d -> ", p->data);
  printf("NULL\n");
}
//head에서 시작하여 1번 노드를 출력한 후 p를 p->link(2번째 노드)로 옮겨서 2번 노드의 값을 출력하는 방식으로 NULL을 만나기 전까지 계속 for문을 돌려 데이터 출력.
```

과제: Singly LinkedList의 insert_last 구현

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

typedef int element;

typedef struct ListNode {
  element data; //입력값을 저장할 부분(int형)
  struct ListNode *link //다음노드의 주소 저장하는 부분
```

```c
}ListNode; //ListNode의 명칭 정함

ListNode* insert_first(ListNode *head, int value) { //연결리스트 첫 번째에 신규값 넣기
  ListNode* p = (ListNode*)malloc(sizeof(ListNode));
  p->data = value;
  p->link = head;
  head = p;
  return head;
}

void print_List(ListNode* head){
  for (ListNode* p = head; p != NULL; p = p->link;)
    printf("%d -> ", p->data);
  printf("NULL\n");
}

ListNode* insert_last(ListNode *head, int value){
  ListNode* old_vl = (ListNode*)malloc(sizeof(ListNode));
  old_vl = head;

  while (old_Vl->link != NULL)
    old_vl = old_vl->link;
  ListNode* lt = (ListNode*)malloc(sizeof(ListNode));
  old_vl_link = lt;
  lt->data = value;
  lt->link = NULL;

  return head;
}

int main(void){
  ListNode* head = NULL;
  int i;

  for (i = 0; i < 5; i++){
```

```
    insert_first(head, i);
    print_list(head);
  }
  printf("insert_last 진행\n");
  insert_last(head, 10);
  print_list(head);

  free(head); //head의 동적 메모리 할당 해제
  system("pause");
  return 0;
}
```

단순연결리스트(Singly LinkedList)2 / 5주-1

지금까지 다룬 리스트 종류 1.배열 2.연결리스트가 있다. 저번시간은 insert 구현했고 이번에는 delete와 search를 구현한다. 이외에도 print, search_list가 있다. 연결리스트에 is_full 구현은 필요없으나 is_empty는 필요하다.

이전시간 복습

```
//노드의 정의
typedef int element;

typedef struct ListNode{
  element data;
  struct ListNode *link; //링크는 주소값을 담는 포인터를 의미한다
} ListNode;

//insert_first 함수
ListNode* insert_first(ListNode* head, int value){
  ListNode* p = (ListNode*)malloc(sizeof(ListNode)); //첫 번째 순서로 들어갈
노드를 만들 ListNode* p
  p->data = value; //p의 data는 value
  p->link = head; //p의 link는 head가 가리키는 1번째 노드
  head = p; //head가 이제 p를 가리키도록 만듬
```

```
  return head;
}

//print_list 함수
void print_list(ListNode* head){
  for (ListNode* p = head; p != NULL; p = p->link)
    printf("%d-> ", p->data);
  printf("NULL\n");
}

//head 사용
int main(void)
{
  ListNode* head = NULL; //헤더포인터 생성
  for (int i = 0; i < 3; i++){
    head = insert_first(head, i);
    print_list(head);
  }
  insert_last(head, 5);
  print_list(head);

  free(head);
  return 0;
}
```

까지 4주-2에서 진행한 내용이고 5주-1에서 배울 함수는 아래와 같다

삭제연산(delete_first)

```
ListNode* delete_first(ListNode* head){
  ListNode* del_f;
  if (del_f == NULL){
    printf("노드가 존재하지 않으므로 삭제가 불가합니다.");
    exit(0);
  }
  del_f = head;
  head = del_f->link;
  free(del_f);

  return head;
}
```

탐색연산(search_list)

```
ListNode* search_list(ListNode* head, element x){
  ListNode* find = head;

  while(find != NULL){ //만약 find가 NULL을 만나지 않을 때까지
    if(find->data == x){ //find의 data가 x와 같다면
      printf("%d는 연결리스트에 존재합니다.\n", x); //출력하고
      return p; //p를 return해준다
    }
    p = p->link; //data가 x가 아니면 다음 노드로 넘어가고
  }
  return NULL; //NULL을 만나기 전까지도 존재하지 않으면 NULL return
}
```

과제: 이를 기반으로 하여 Singly LinkedList의 reverse함수 만들기

```
#pragma warning(disable:4996)

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

```c
typedef int element;

typedef struct ListNode{
  element data;
  struct ListNode* link;
}ListNode;

ListNode* insert_first(ListNode* head, int value){
  ListNode* p = (ListNode*)malloc(sizeof(ListNode));
  p->data = value;
  p->link = head;
  head = p;

  return head;
}

ListNode* insert_last(ListNode* head, int value){
  ListNode* old_vl = head;

  while(old_vl->link != NULL){
    old_vl = old_vl->link;
  }
  ListNode* lt(ListNode*)malloc(sizeof(ListNode));
  old_vl->link = lt;
  lt->link = NULL;
  lt->data = value;

  return head;
}

void print_list(ListNode* head){
  for(ListNode* p = head; p != NULL; p = p->link){
    printf("%d->", p->data);
  }printf("NULL\n");
}
```

```c
ListNode* delete_last(ListNode* head){
  ListNode* remove_a = head;

  if(head == NULL)
    return NULL;
  while(remove_a->link->link != NULL)
    remove_a = remove_a->link;
  ListNode* remove_b = remove_a->link;
  remove_a->link = NULL;
  free(remove_b);

  return head;
}

ListNode* search_list(ListNode* head, int x){
  ListNode* p = head;
  while(p != NULL){
    if(p->data == x){
      printf("%d가 리스트에 존재합니다.\n", x);
      return p;
    }
    p = p->link;
  }printf("%d는 리스트에 존재하지 않습니다.\n", x);
  return NULL;
}

ListNode* reverse(ListNode* head, ListNode* head2){
  while (head != NULL) {
    head = insert_first(head2, head->data);
    if(head->link == NULL){
      break;
    }
    head = head->link;
  }
  return head2;
}
```

```
int main(void {
  ListNode* head = NULL;
  ListNode* head2 = NULL;

  for (int i = 0; i < 5; i++){
    head = insert_first(head, i);
    print_list(head);
  }
  head = insert_last(head, 10);
  print_list(head);

  search_list(head, 0);

  head = delete_last(head);
  print_list(head);

  head2 = reverse(head, head2);
  print_list(head2);

  free(head);
  free(head2);

  system("pause");
  return 0;
}
```

단순연결리스트 3 / 5주 -2

n번째 노드에 데이터 넣기

고려해야할 사항

- 첫번째 위치에 값을 입력하는 경우 head를 담을 임시노드가 필요
- 마지막 위치에 값을 입력하는 경우 마지막 노드를 담을 임시노드가 필요
- 2번째 위치(중간)에 값을 입력하는 경우 임시 노드가 두개 필요

Insert_position(= n번째 노드에 데이터 넣기) 코드

```
//n번째 노드에 데이터를 넣는 방법 구현
ListNode* insert_position(ListNode* h, int pos, element value)
{ //insert_first와 last를 여기서 진행 가능한데 따로 빼는게 더 좋다
  int k = 1;
  ListNode* p = NULL, *q = NULL; //pos 위치 잡기위한 임시노드
  ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
  newNode->data = value;
  p = h;

  if (pos == 1) { //=insert_first
    newNode->link = p;//newNode->link가 p 가리킨다
    h = newNode; //head는 newNode 가리킨다
  }
  else { //그 외 경우
    while (p != NULL && (k<pos)) { //pos 위치찾기
      k++ //k가 몇번째인지 세어준다
      q = p; //q가 p가 있던 위치 가리키고
      p = p->link; //p는 그 다음 노드로 넘어간다
    }
    if(p == NULL) { //=insert_last
      q->link = newNode; //기존 마지막노드가 newNode를 가리킨다
      newNode->link = NULL; //newNode가 마지막이니 NULL 가리킨다
    }
    else{ //insert_position
      q->link = newNode; //들어갈 자리 이전노드가 newNode 가리키고
      newNode->link = p; //newNode는 들어가는 이후 노드를 가리킨다
    }
  }
  return head;
}
```

리스트의 전체 삭제

자료에 존재하는 delete_first를 지속 반복하여 모든 노드를 삭제해준 후 free(head)로 헤드포인터도 동적할당 해제를 해 주어야 한다. 헤드포인터만 동적할당 해제를 한다고 연결리스트가 사라지는 것이 아니기 때문이다.

과제: insert_position을 기반으로 delete_position 구현하기

```
#pragma warning(disable:4996)

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

typedef int element;
typedef struct ListNode {
  element data;
  struct ListNode* link;
}ListNode;

ListNode* insert_first(ListNode* head, int value) {
  ListNode* p = (ListNode*)malloc(sizeof(ListNode));
  p->data = value;
  p->link = head;

  return head;
}

void print_list(ListNode* head) {
  ListNode* p = head;
  while(p != NULL){
    printf("%d", p->data);
    p = p->link
  }
  printf("NULL\n");
}

ListNode* insert_last(ListNode* head, element value) {
```

```
  ListNode* old_vl = head;
  ListNode* lt = (ListNode*)malloc(sizeof(ListNode));
  lt->data = value;
  lt->link = NULL;

  if (old_vl = NULL){
    while(old_vl->link != NULL) {
      old_vl = old_vl->link
    }
    old_vl->link = lt;
  }
  else {
    head = lt;
  }
return head;
}

ListNode* delete_last(ListNode* head) {
  ListNode* remove_a = head;

  if(head == NULL)
    return NULL;
  else {
    while(remove_a->link->link != NULL) {
      remove_a = remove_a->link;
    }
  }
  ListNode* remove_b = remove_a->link;
  free(remove_b);

  return head;
}

ListNode* search_list(ListNode* head, int x) {
  ListNode* p = head;
  while(p != NULL){
```

```
    if (p->data == x){
      printf("%d은/는 리스트에 존재합니다.", x);
      return p;
    }
    p = p->link;
  }
  return NULL;
}

ListNode* delete_position(ListNode* h, int pos) {
  int k = 1;
  ListNode* delnode = NULL, *auxilnode = NULL;
  delnode = h;

  if (pos == 1) {
    h = delnode->link;
    free(delnode);
  }
  else {
    while (delnode != NULL && (k < pos)) {
      k++;
      auxilnode = delnode;
      delnode = delnode->link;
      if (k == pos) {
        auxilnode->link = delnode->link;
        free(delnode);

        return h;
      }
    }
  }
  return h;
}

int main(void) {
  ListNode* head = NULL;
```

```
    for (int i = 0; i < 5; i++) {
      head = insert_first(head, i);
      print_list(head);
    }
    head = insert_last(head, 10);
    print_list(head);

    search_list(head, 2);

    head = delete_last(head);
    print_list(head);

    delete_position(head, 3);
    print_list(head);

    system("pause");
    return 0;
}
```

원형연결리스트(Circular LinkedList) / 6주 - 1

이전까지 진행한 것은 단방향 연결 리스트(Singly LinkedList)이다. 이번에는 마치 원형처럼 마지막 노드가 첫 번째 노드를 가리키는 원형 연결 리스트(Circular LinkedList)에 대해서 알아볼 것이다.

스택을 구현하는 방식 3개 비교

Singly LinkedList / Array / Dynamic Array

Singly LinkedList의 경우 데이터를 찾는 indexing이 두 가지 경우에 비해 느린 특징을 보인다. 물리적으로 데이터가 붙어있지 않아서 그렇다. Array의 경우 비교적 Singly LinkedList보다 indexing은 빠르지만 크기가 고정되어있어 불편한 점이 존재한다. 이를 보완하고자 Dynamic Array가 존재하고 이는 Array의 단점을 어느정도 보완하였다.

원형 연결 리스트의 ADT

원형 연결 리스트 데이터 넣기(insert_first와 insert_last)

원형연결리스트는 마지막 노드 링크가 첫 번째로 연결되는 것을 제외하고는 단순 연결 리스트와 동일한 구조를 가진다.

```
//circular linkedlist의 insert_first
#pragma warning(disable:4996)

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

typedef int element;
typedef struct ListNode {
  element data;
  struct ListNode* link;
}ListNode;

ListNode* insert_first(ListNode* head, element value) {
  ListNode* new = (ListNode*)malloc(sizeof(ListNode));
  new->data = value; //data부분에는 미리 value 삽입
  if (head == NULL) { //노드가 아무것도 없으면
    head = new; //head가 new노드 가리키게 하고
    new->link = head; //new의 link도 new를 가리키게 함(원형구조)
  }
  else { //만약 기존값이 존재하면
    new->link = head->link; //신규노드 링크값은 기존 첫번째 노드
    head->link = new; //head가 가리키는 중인 마지막 노드는 신규노드 가리킴
  }
  return head;
}

//circular의 insert_last
ListNode* insert_last(ListNode* head, element value) {
  ListNode* new = (ListNode*)malloc(sizeof(ListNode));
  new->data = value;
```

```
  if (head == NULL){ //기존 노드가 없으면 first에 구현한 부분과 동일한 기능 함.
    head = new;
    new->link = head;
  }
  else { //만약 기존값 존재시
    new->link = head->link; //원형이므로 신규노드 링크는 첫 번째 노드 가리킴
    head->link = new; //기존 마지막 노드의 링크값은 신규노드를 가리킴
    head = new; //head는 신규노드를 가리킴
  }
  return head;
}
```

과제: Circular LinkedList의 insert_first / insert_last를 기반으로 delete_first / delete_last 구현

```
#pragma warning(disable:4996)

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

typedef int element;
typedef struct ListNode {
  element data;
  struct ListNode* link;
}

ListNode* insert_first(ListNode* head, element value) {
  ListNode* new = (ListNode*)malloc(sizeof(ListNode));
  new->data = value;

  if (head == NULL) {
    head = new;
    new->link = head;
  }
  else {
    new->link = head->link;
```

```
    head->link = new;
  }
  return head;
}

ListNode* insert_last(ListNode* head, elememt value) {
  ListNode* new = (ListNode*)malloc(sizeof(ListNode));
  new->data = value;

  if (head == NULL) {
    head = new;
    new->link = head;
  }
  else {
    new->link = head->link;
    head->link = new;
    head = new;
  }
}

void print_list(ListNode* head) {
    ListNode* p;

    if (head == NULL) return;
    p = head->link;
    do {
      printf("%d->", p->data);
      p = p->link;
    }while (p != head);
    printf("%d\n", p->data);
  }
}

ListNode* delete_first(ListNode* head) {
  ListNode* del_f;
  ListNode* prevnode;
```

```
  prevnode = head;
  del_f = prevnode->link;
  free(del_f);

  return head;
}

ListNode* delete_last(ListNode* head) {
  ListNode* del_lt;
  ListNode* prevnode;

  prevnode = head;
  while (prevnode->link != head) {
    del_lt = prevnode->link;
  }
  prevnode->link = head->link;
  free(head);

  return 0;
}

int main(void) {
  ListNode* head = NULL;

  for (int i = 0; i < 5; i++) {
    head = insert_first(head, i);
    print_list(head);
  }

  head = insert_last(head, 10);
  print_list(head);

  head = delete_first(head);
  print_list(head);
```

```
    head = delete_last(head);
    print_list(head);

    system("pause");
    return 0;
}
```

양방향(이중) 연결 리스트(Doubly LinkedList)1 / 7주-1,2

*6주-2의 경우 쪽지시험으로 대체됨. 시험내용은 Struct와 Singly LinkedList를 복합하여 사용하는 문제 하나와 Singly LinkedList에서 두 LinkedList를 하나로 합치는 add함수와 LinkedList 두 개를 합치는 데 번갈아가며 배치하는 mix함수를 구현해야 한다.

이번 내용은 연결리스트 중 원형이 아니고 이전 노드와 다음 노드의 주소값을 같이 가지는 양방향(이중) 연결 리스트에 대해서 배운다.

이중 연결 리스트란

이중(양방향) 연결 리스트란 단순 연결 리스트의 문제점인 선행 노드를 찾아내기 힘든 부분이 해소되도록 각 노드가 자신의 이전 노드의 주소도 가지는 형태를 갖추어 자신의 이전 노드와 다음 노드의 링크값을 다 가지는 연결 리스트를 의미한다. 단순(단방향) 연결 리스트의 문제점인 이전 노드 주소를 모르는 문제점은 해결했지만 그때문에 데이터가 공간을 많이 차지하게 되고 코드가 길어지며 복잡한 구조를 가지게 된다.

> 이전 노드를 가리키는 형식
>
> Llink(이전) / Rink(다음) 또는 prevlink / nextlink 를 이용하여 struct DListNode에 구현해야 한다.

```
//doubly LinkedList의 노드 구조
typedef int element;

typedef struct DListNode {
  struct DListNode* Llink; //앞 노드를 가리키는 Llink
  int data;
  struct DListNode* Rlink; //뒷 노드를 가리키는 Rlink
}DListNode;
```

이중 연결 리스트의 구현방법

이중 연결 리스트의 경우 헤드포인터가 아니라 헤더 노드를 만들어야 한다. 교재 기준으로 헤더노드를 만들고 헤더노드가 무조건 제일 앞에 오며 헤더 노드는 다음에 올 첫 번째 노드에서 헤더 노드를 Llink가 가리키게 하고 헤더노드의 Rlink가 첫 번째 노드를 가리키게 하며 헤더노드의 Llink는 리스트의 마지막 노드를 가리키며 리스트의 마지막 노드가 Rlink로 헤더노드를 가리키는 구조를 채택하였다. 이는 Circular방식에 Doubly를 섞었다고 볼 수도 있을 것이다.

> 헤드노드란?
>
> 헤드노드(헤더노드)는 데이터를 갖지 않고 단지 삽입과 삭제 코드를 단순화 하기 위한 목적으로 만드는 노드이다.
>
> 헤드 포인터와는 다른 개념이며 공백 상태에서는 헤더 노드의 양쪽 링크가 자기 자신을 가리키는 Circular와 같은 형태를 보인다.

이중 연결 리스트의 insert function 구조(삽입)

헤더(첫 번째)에 삽입하는 방법

```
//doubly의 insert_first 구조(헤더노드 이용)
void dinsert(DListNode* before, element value) {
  DListNode* new = (DListNode*)malloc(sizeof(DListNode));
  new->data = value; //일단 먼저 data부분에 value값 넣고
  new->Llink = before; //새 노드 Llink는 헤더를 가리키고
  new->Rlink = before->Rlink; //새 노드 Rlink는 헤더가 가리키던 기존 첫 번째 노드를 가리키고
  before->Rlink->Llink = new; //헤더가 가리키는 기존 첫 번째 노드의 Llink는 새 노드를 가리키게 한다
  before->Rlink = new; //그리고 헤더의 Rlink가 새 노드를 가리키도록 설계한다
}
```

중간에 삽입하는 방법

```
void dinsert(DListNode* before, element value) {
  //여기서 특징은 before을 특정하지 않으면 무조건 header에 들어간다는 점이다.
  DListNode* new = (DListNode*)malloc(sizeof(DListNode));
  new->data = value; //일단 새 노드의 data에는 value를 넣음

  new->Llink = before; //새 노드의 Llink에는 before의 주소 넣음
  new->Rlink = before->Rlink; //새 노드 Rlink에는 Before의 다음 노드 주소를 넣음
  before->Rlink->Llink = new; //before의 다음 노드가 가지는 Llink에는 새 노드 주
소를 넣음
  before->Rlink = new; //마지막으로 before의 Rlink에 새 노드 주소를 넣음
}
```

헤더 노드를 쓰지 않고 헤드포인터를 이용하는 방식도 있다(이중포인터 이용)

```
void DLLinsert(DListNode** h, int pos, int value) {
  int k = 1; //중간에 넣을 때 이용
  DListNode *new, *temp_a, *temp_b;
  new = (DListNode*)malloc(sizeof(DListNode));
  new->data = value; //앞에서와 마찬가지로 data에 값 먼저 입력
  if(pos == 1){ //첫번째에 넣는 경우
    if((*h) == NULL){ //값이 존재하지 않을 때
      new->Llink = NULL; //새 노드 Llink에는 NULL(헤드포인터므로)
      new->Rlink = *h; //새 노드 Rlink에도 NULL(원래 h는 NULL 가리키고 있었음 =
값이 없었음)
      *h = new; //그러고 헤드포인터가 새 노드 가리킴
    }
    else { //기존 값이 있을 때
      new->Llink = NULL;
      new->Rlink = *h;
      *h->Llink = new; //이 부분 추가. 기존 첫 번째 노드 Llink가 새 노드 가리킴
      *h = new;
    }
    return;
  }
  temp_a = *h; //두 개의 temp 필요해서 일단 a와 b를 붙임.
```

```
  while ((k < pos - 1) && temp_a->Rlink != NULL) {
    //k를 pos-1까지만 이동시키고 동시에 NULL을 만나지 않는다는 조건을 달아준다.
    temp_a = temp_a->Rlink;
    k++
  }
  if (temp_a->Rlink == NULL) { //마지막에 노드 삽입시 작동
    new->Rlink = temp_a->Rlink; //새 노드 Rlink는 temp_a의 Rlink를 가리키게 한
다
    new->Llink = temp_a; //새 노드 Llink는 temp_a를 가리키게 한다
    temp_a->Rlink = new; //temp_a->Rlink는 새 노드를 가리킨다
  }
  else { //마지막이 아닌 중간일 시
    new->Rlink = temp_a->Rlink; //새 노드의 Rlink는 temp_a의 Rlink를 가리킨다
    new->Llink = temp_a; //새 노드 Llink는 temp_a를 가리킨다
    temp_b = temp_a->Rlink; //temp_b를 temp_a의 RLlink로 보낸다
    temp_a->Rlink = new; //temp_a의 Rlink는 새 노드를 가리킨다
    temp_b->Llink = new; //b의 Llink도 새 노드를 가리킨다
  }
  return;
}
```

7주 1과 2를 묶어놔서 일단 7주 1의 과제를 먼저 정리한 후 7주 2를 정리할 것.

```
//7주-1 과제: Doubly LinkedList의 delete_first 구현(헤더노드 이용하는 코드구현법)

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

typedef int element;
typedef struct DListNode {
  DListNode* Llink;
  DListNode* Rlink;
  element data;
```

```
}DListNode; //doubly linkedlist는 다른 연결리스트와 다르게 이전 노드와 다음 노드의
주소값을 둘 다 가진다.

void init(DListNode* phead) {
  phead->Llink = phead;
  phead->Rlink = phead;
}

void print_list(DListNode* phead) {
  DListNode* p;
  for(p = phead->Rlink; p != phead; p = p->Rlink)
    printf("<-||%d||->", p->data);
  printf("\n");
}

void dinsert_first(DListNode* phead, element value) {
  DListNode* p = (DListNode*)malloc(sizeof(DListNode));
  p->data = value;
  p->Rlink = phead->Rlink;
  p->Llink = phead;
  phead->Rlink->Llink = p;
  phead->Rlink = p;
}

void ddelete_first(DListNode* phead, DListNode* del) {
  if (phead->Rlink == NULL) return;
  del->Llink->Rlink = del->Rlink;
  del->Rlink->Llink = del->Llink;
  free(del);
}

int main(void)
{
  DListNode* head = (DListNode*)malloc(sizeof(DListNode));

  init(head);
```

```
  for(int i = 0; i < 5; i++){
    dinsert_first(head, i);
    print_list(head);
  }
  ddelete_first(head, head->Rlink);
  print_list(head);

  free(head);
  system("pause");
  return 0;
}
```

```
//7주-2 과제: doubly linked list의 position을 사용한 삭제 함수를 구현(헤더노드가 아닌 헤드포인터를 이용하는 코드)

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

typedef int element;
typedef struct DListNode {
  DListNode* Llink;
  DListNode* Rlink;
  element data;
}DListNode;

void DLLinsert(DListNode** h, int pos, int value) {
  int k = 1; //중간에 넣을 때 이용
  DListNode *new, *temp_a, *temp_b;
  new = (DListNode*)malloc(sizeof(DListNode));
  new->data = value; //앞에서와 마찬가지로 data에 값 먼저 입력
  if(pos == 1){ //첫번째에 넣는 경우
    if((*h) == NULL){ //값이 존재하지 않을 때
      new->Llink = NULL; //새 노드 Llink에는 NULL(헤드포인터므로)
```

```
      new->Rlink = *h; //새 노드 Rlink에도 NULL(원래 h는 NULL 가리키고 있었음 = 값이 없었음)
      *h = new; //그러고 헤드포인터가 새 노드 가리킴
    }
    else { //기존 값이 있을 때
      new->Llink = NULL;
      new->Rlink = *h;
      *h->Llink = new; //이 부분 추가. 기존 첫 번째 노드 Llink가 새 노드 가리킴
      *h = new;
    }
    return;
  }
  temp_a = *h; //두 개의 temp 필요해서 일단 a와 b를 붙임.
  while ((k < pos - 1) && temp_a->Rlink != NULL) {
    //k를 pos-1까지만 이동시키고 동시에 NULL을 만나지 않는다는 조건을 달아준다.
    temp_a = temp_a->Rlink;
    k++
  }
  if (temp_a->Rlink == NULL) { //마지막에 노드 삽입시 작동
    new->Rlink = temp_a->Rlink; //새 노드 Rlink는 temp_a의 Rlink를 가리키게 한다
    new->Llink = temp_a; //새 노드 Llink는 temp_a를 가리키게 한다
    temp_a->Rlink = new; //temp_a->Rlink는 새 노드를 가리킨다
  }
  else { //마지막이 아닌 중간일 시
    new->Rlink = temp_a->Rlink; //새 노드의 Rlink는 temp_a의 Rlink를 가리킨다
    new->Llink = temp_a; //새 노드 Llink는 temp_a를 가리킨다
    temp_b = temp_a->Rlink; //temp_b를 temp_a의 RLlink로 보낸다
    temp_a->Rlink = new; //temp_a의 Rlink는 새 노드를 가리킨다
    temp_b->Llink = new; //b의 Llink도 새 노드를 가리킨다
  }
  return;
}

void delete_pos(DListNode** h, int pos) { //=pos가 가리키는 위치의 노드 삭제
  DListNode* del,* Ldel,* Rdel;
```

```
  del = *h; //del을 헤드포인터가 가리키는 것을 가리키도록 설정
  printf("---delete_pos---\n");
  if(pos == 1) { //첫 번째를 삭제할 때
    Rdel = del->Rlink; //Rdel을 del의 Rlink로 보냄
    Rdel->Llink = NULL; //Rdel의 Llink는 Null을 가리키게 함
    *h = Rdel; //헤드포인터가 Rdel을 가리키게 함
  }
  else { //만약 첫 번째가 아니라면
    for (int n = 1; n < pos; n++){ //int n을 1로 선언 후 n을 입력받은 pos 이전
까지 옮겨줌
      del = del->Rlink //n의 움직임에 따라 del을 옮겨줌
    }
    if (del->Rlink == NULL) { //del->Rlink가 Null을 만나면(delete_last)
      Ldel = del->Llink;
      Ldel->Rlink = NULL;
    }
    else {
      Rdel = del->Rlink;
      Ldel = del->Llink;
      Rdel->Llink = Ldel;
      Ldel->Rlink = Rdel;
    }
  }
  free(del);
  return;
}

void print_list(DListNode** h) { // list를 출력하는 함수
   DListNode* p;
   p = *h;
   for (p = *h; p != NULL; p = p->rlink) {
      printf("<- %d ->", p->data);
   }
   printf("\n");
   return;
}
```

```c
int main(void)
{
  DListNode* head = NULL;

  for (int i = 0; i < 5; i++){
    DLLinsert(&head, 1, i);
    print_list(&head);
  }
  DLLinsert(&head, 2, 5); print_list(&head);
  DLLinsert(&head, 4, 10); print_list(&head);

  delete_pos(&head, 1); print_list(&head);
  delete_pos(&head, 3); print_list(&head);
  delete_pos(&head, 5); print_list(&head);

  system("pause");
  return 0;
}
```