

자료구조와 알고리즘

○ **알고리즘** : 컴퓨터로 문제를 풀기 위한 단계적인 절차

- * **유한성** : 한정된 수의 단계 후에는 반드시 종료되어야 한다. // 알고리즘과 프로그램의 차이
- * `ArrayMax(A,n)` : 최댓값을 찾는 함수 // A: 배열, n : 개수

* 추상 데이터 타입(ADT : Abstract Data Type)

- 데이터 연산이 무엇인가는 정의하지만, 어떻게 구현될 것인지는 표현되지 않음.

- * `#include<time.h>` / `clock()` : 시간을 측정하는 함수
- * `CLOCKS_PER_SEC` : `clock` 함수의 결과를 초로 변경

* **시간복잡도 함수** : 연산의 수행횟수는 고정된 숫자가 아니라 입력의 개수 n 에 대한 함수

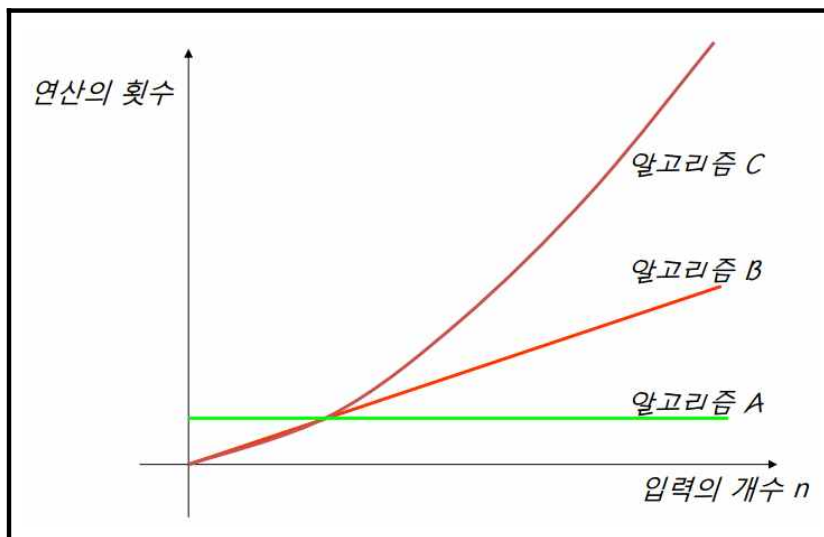
// ex) 연산의 수 = 8 \rightarrow $3n+2$

* **복잡도 분석의 예** (n 을 n 번 더하는 문제)

	알고리즘A	알고리즘B	알고리즘C
코드	<code>sum <- n*n;</code>	<code>sum <- 0;</code> <code>for i <- 1 to n do</code> <code>sum <- sum+n;</code>	<code>sum <- 0;</code> <code>for i <- 1 to n do</code> <code>for i <- 1 to n do</code> <code>sum <- sum+1;</code>
대입연산	1	$n + 1$	$n*n + 1$
덧셈연산		n	$n*n$
곱셈연산	1		
나눗셈연산			
전체연산수	2	$2n + 1$	$2n^2 + 1$

* 알고리즘 A가 제일 좋음

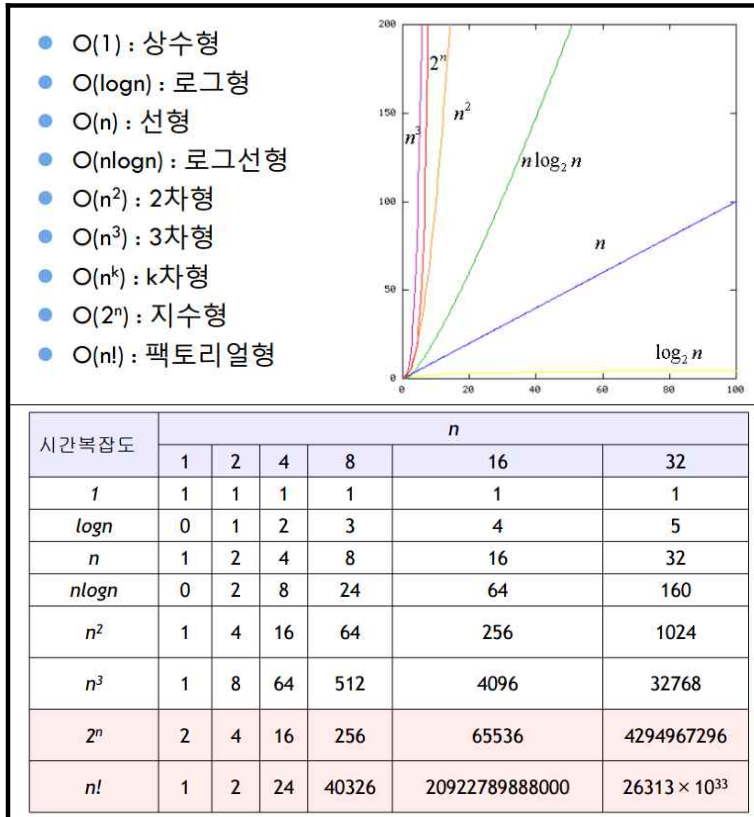
* **상수시간대** : 입력 개수와 연산의 횟수가 고정되어 있는 것



* **빅오 표기법** : 연산의 횟수를 대략적으로 표기한 것 / 데이터의 값에 따라 수행시간이 달라짐

- 두 개의 함수 $f(n)$ 과 $g(n)$ 이 주어졌을 때, 모든 $n \geq n_0$ 에 대하여 $|f(n)| \leq c|g(n)|$ 을 만족하는 2개의 상수 c 와 n_0 가 존재하면 $f(n)=O(g(n))$ 이다.
- 함수의 상한을 표시한다.(최고차항만 표시) // ex) $n \geq 5$ 이면 $2n+1 < 10n$ 이므로 $2n+1 = O(n)$
- 빅오의 데이터 이상의 시간이 걸리지 않는다. // ex) $O(n^2)$

[시험문제!!!!!!]- 빅오 표기법의 종류



* 빅오메가 표기법

- 모든 $n \geq n_0$ 에 대하여 $|f(n)| \geq c|g(n)|$ 을 만족하는 2개의 상수 c 와 n_0 가 존재하면 $f(n) = \Omega(g(n))$ 이다
- 빅오메가는 함수의 하한을 표시한다.
- ex) $n \geq 1$ 이면 $2n+1 > n$ 이므로 $n = \Omega(n)$

* 최선, 평균, 최악의 경우

- (예) 순차탐색
- **최선의 경우**: 찾고자 하는 숫자가 맨 앞에 있는 경우
 $\therefore O(1)$
- **최악의 경우**: 찾고자 하는 숫자가 맨 뒤에 있는 경우
 $\therefore O(n)$
- **평균적인 경우**: 각 요소들이 균일하게 탐색된다고 가정하면

$$(1+2+\dots+n)/n = (n+1)/2$$
 $\therefore O(n)$

인덱스 0에서 값 5 발견
숫자 비교 횟수 = 1

5	9	10	17	21	29	33	37	38	43
0	1	2	3	4	5	6	7	8	9

인덱스 9에서 값 43 발견
숫자 비교 횟수 = 10

5	9	10	17	21	29	33	37	38	43
0	1	2	3	4	5	6	7	8	9

인덱스 5에서 값 26 발견
숫자 비교 횟수 = 6

2	7	16	19	28	26	35	42	46	50
0	1	2	3	4	5	6	7	8	9

* 연습문제 2 : 코드의 각 명령문의 수행 횟수를 계산하여 시간 복잡도 함수를 계산하십시오

시간 복잡도 구하기	시간 복잡도 구하기2	시간 복잡도 구하기3
<pre> int test(int n){ int i; int total; total=1; // 1번 for(i=1; i<n; i++){ total *=n; // n-2번 } return n; // 1번 } </pre>	<pre> float sum(float list[], int n){ float tempsum; int i; tempsum=0; // 1번 for(i=0; i<n; i++){ tempsum+=list[i]; // n-1번 } tempsum += 100; // 1번 tempsum += 200; // 1번 return tempsum; // 1번 } </pre>	<pre> int test(int n){ int i,b; b=1; // 1번 i=1; // 1번 while(i <= n) i = i*b; // logn 번 } </pre>
$1+n-2+1 = n$ 시간 복잡도 : $O(n)$ 번	$1 + n-1 + 1 + 1 + 1 = n+3$ 시간 복잡도 : $O(n)$ 번	$1+1+\log n = \log n+2$ $O(\log n)$ 번

순환 (순환(재귀) 알고리즘(함수)) : 자기 자신을 호출하는 함수(매개변수만 달라짐)

* 팩토리얼 : 순환 알고리즘은 **멈추는 부분**과 **순환 호출하는 부분**으로 나뉘어진다.

팩토리얼 코드

```
int factorial(int n){
    if(n<=1) return 1; // 멈추는 부분
    else return (n*factorial(n-1)); // 순환 호출하는 부분
}
```

* **피보나치 수열** : 초기 값 : 1,1을 더해 그 다음 숫자를 구해냄

- 순환 알고리즘으로 풀면 더 느려짐
- 공식 : $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$

피보나치 수열 예제

1	2	5	13	34	89
1	3	8	21	55	144

* **하노이 탑** : 기둥 세 개에 원반 옮기기 (작은 것이 큰 것 밑에 오면 안됨) // $2^n - 1$ 번이 경우의 수

- 공식 : $\text{hanoi}(n) = 2 * \text{hanoi}(n-1) + 1$, if $n \geq 2$
- ex) $\text{hanoi}(10) = \text{hanoi}(9)*2 = (\text{hanoi}(8)*2)*2 = \dots$ 1이 될 때 까지

하노이탑 코드

```
#include <stdio.h>

void hanoi(int n, char from, char tmp, char to)
{
    if (n == 1)
        printf("원반 1을 %c 에서 %c으로 옮긴다.\n", from, to);
    else {
        hanoi(n - 1, from, to, tmp);
        printf("원반 %d을 %c에서 %c으로 옮긴다.\n", n, from, to);
        hanoi(n - 1, tmp, from, to);
    }
}

main()
{
    char from, to, tmp;
    int n;

    from = 'a'; to = 'c'; tmp = 'b';

    printf("원반 개수 입력");
    scanf("%d", &n);

    hanoi(n, from, tmp, to);
}
```

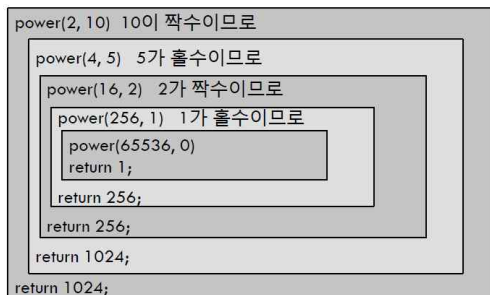
* **거듭제곱 값**

- ex1) $3^9 = 3*3^8 = 3*(3^2)^4 = 3*((3^2)^2)^2 = 3*((9)^2)^2 = 3*(81)^2 = 3*6561*6561^0 = 3*6561*1 = 19683$
- ex2) $3^8 = (3^2)^4 = (9^2)^2 = 81^2 = 6561$
- ex3) $2^{10} = (4)^5 = 4*(4)^4 = 4*(16)^2 = 4*(256)^1 = 4*256*1 = 1024$

ex2 거듭제곱 코드

```
power(x, n)
    if n=0 then return 1; // 멈추는 부분
    else if n이 짝수 then return power(x^2, n/2);
    else if n이 홀수 then return x*power(x^2, (n-1)/2);
```

2¹⁰을 계산하는 과정

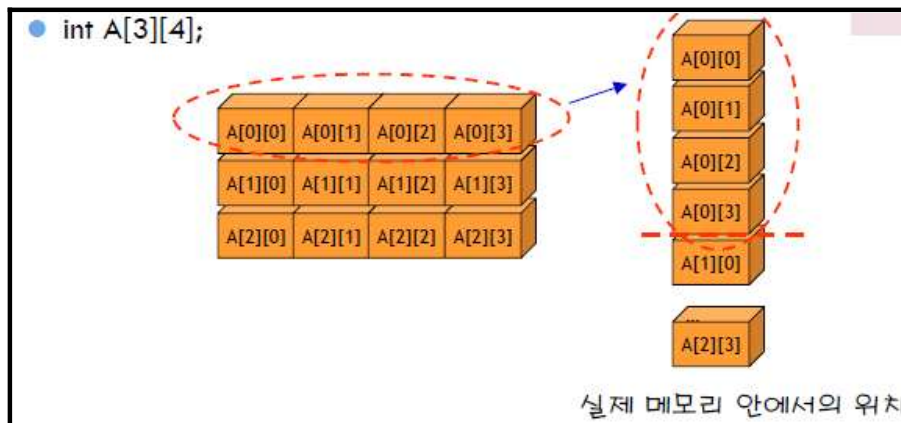


* **이진탐색** : 가운데 수를 기준으로 반씩 쪼개서 탐색함

배열, 구조체, 포인터

○ 배열

- * 배열 형태 : (자료형) 배열이름[] //ex) int A[3];
- * 메모리 주소 : 포인터(*)
 - int일 경우의 메모리크기는 4byte : [0] : 1200000 / [1] : 1200004 / [2] : 1200008
- * 2차원 배열 : int A[3][4]
 - 행(row)열(column)로 이루어짐
 - 2차원 배열시 주소 값 // ex) base+sizeof(int)*6
// base가 100일 경우 int는 4byte이므로 다음 배열은 124



- * 배열의 응용 - 다항식 : 2가지 방법
- * 모든 차수에 대한 계수 값을 배열로 저장한 방법 // A : $4x^{10}+3x^7+2x^5+6$
 - 장점 : 다항식의 각종 연산이 간단해짐
 - 단점 : 대부분의 항의 계수가 0이면 공간의 낭비가 심함

x 지수 (배열 수)	10	9	8	7	6	5	4	3	2	1	0
계수	4	0	0	3	0	2	0	0	0	0	6

- * 계수*지수를 따로 저장한 방법 (0이 아닌 항 저장) // B : $10x^{11}+15x^6+7x^3+5x$
 - 장점 : 메모리 공간의 활용이 효율적
 - 단점 : 다항식의 연산들이 복잡해짐

배열 수	1	2	3	4
계수(행)	10	15	7	5
x 지수(행)	11	6	3	1

배열의 응용 코드

```
#define MAX_TERMS 101

struct {
    float coef;
    int expon;
} terms[MAX_TERMS]= { {8,3}, {7,1}, {1,0}, {10,3}, {3,2}, {1,0} };

int avail=6;

// 두 개의 정수를 비교
char compare(int a, int b) {
    if( a>b ) return '>';
    else if( a==b ) return '=';
    else return '<';
}
```

* 다항식 연습문제 : $8x^6 + 7x^4 + 5x^3 + 3x$

- 첫 번째 방법

x 지수 (배열 수)	6	5	4	3	2	1	0
계수	8	0	7	5	0	3	0

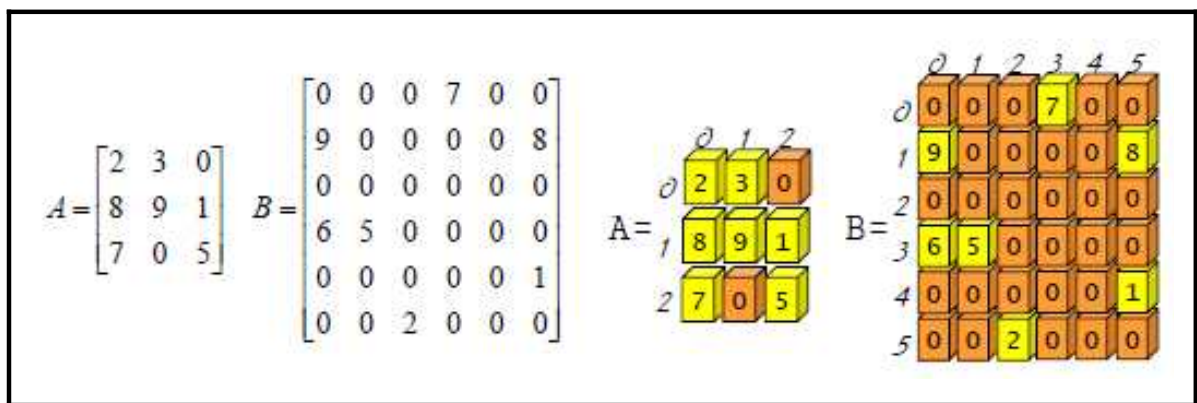
- 두 번째 방법

배열 수	1	2	3	4
계수(행)	8	7	5	3
x 지수(행)	6	4	3	1

* **희소행렬** : 대부분의 항들이 0인 배열

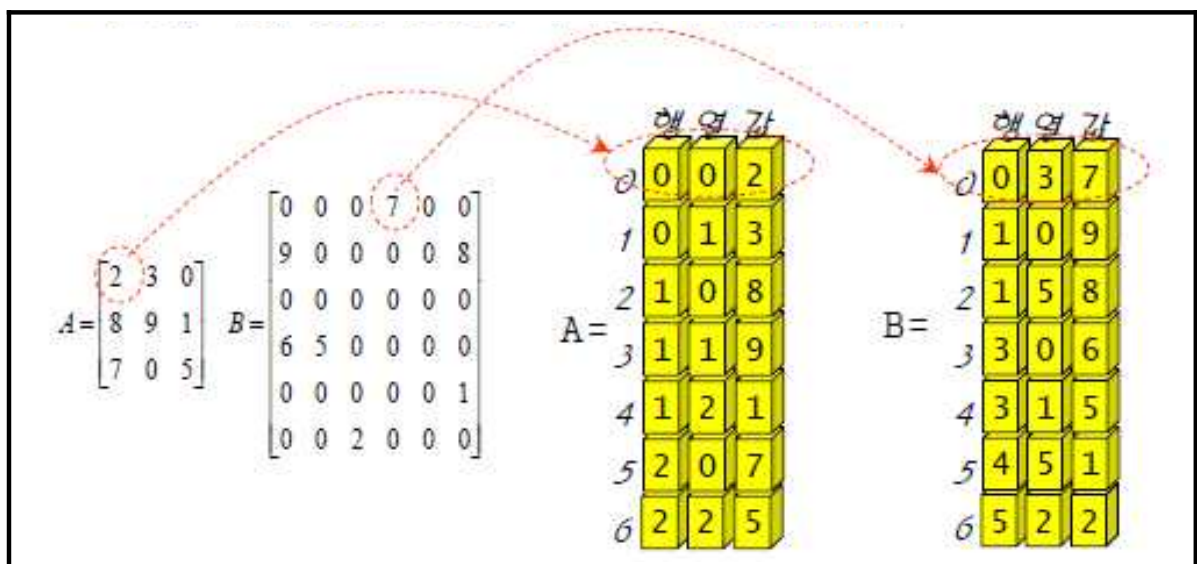
* 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법

- 장점 : 행렬의 연산들을 간단하게 구현할 수 있다.
- 단점 : 대부분의 항들이 0인 희소 행렬의 경우 많은 메모리 공간 낭비



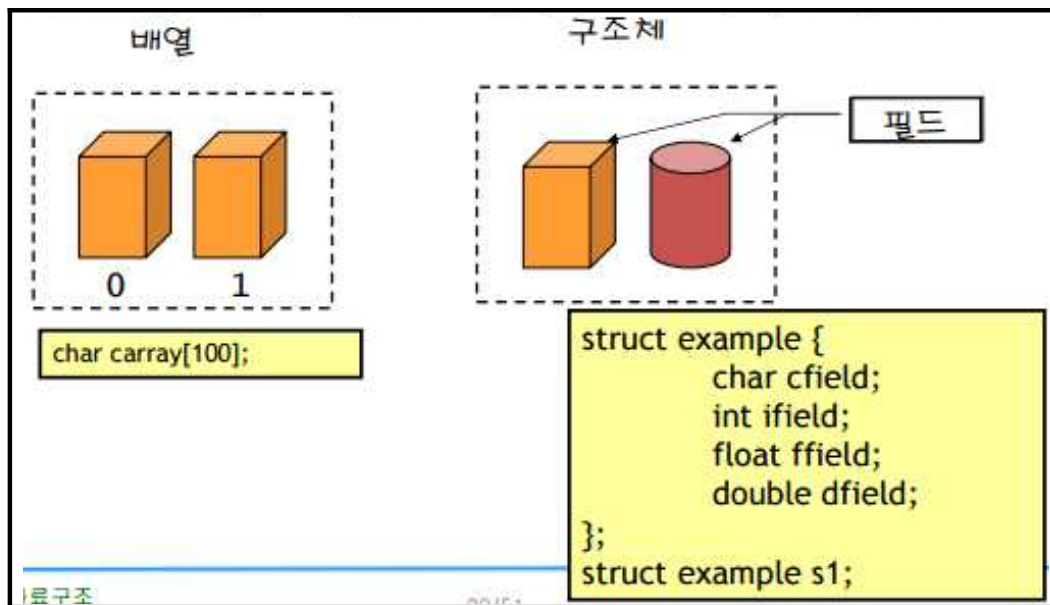
* 0이 아닌 요소들만 저장하는 방법

- 장점 : 희소 행렬의 경우, 메모리 공간의 절약
- 단점 : 각종 행렬 연산들의 구현이 복잡해진다.



○ 구조체(structure) : 타입이 다른 데이터를 하나로 묶는 방법

- * 배열 : 타입이 같은 데이터들을 하나로 묶는 방법
- * 항목 하나하나를 필드라고 한다.
- * 구조체 형태



- * 구조체 선언과 구조체 변수의 생성
 - . 연사자를 사용하여 변수를 불러올 수 있다. // ex) korea[10].age

● 구조체의 선언과 구조체 변수의 생성

```
struct person {
    char name[10]; // 문자배열로 된 이름
    int age;       // 나이를 나타내는 정수값
    float height;  // 키를 나타내는 실수값
};
struct person a; //구조체 변수 선언
struct person korea[100]; //구조체 배열
```

● typedef을 이용한 구조체의 선언과 구조체 변수의 생성

```
typedef struct person {
    char name[10]; // 문자배열로 된 이름
    int age;       // 나이를 나타내는 정수값
    float height;  // 키를 나타내는 실수값
} person;
person a; // 구조체 변수 선언
person korea[100]; //구조체 배열
```

- **typedef** : 자료형의 구조체를 정의해줌 // struct의 이름과 달라도 됨 (둘 중 아무거나 써도 상관없음)

구조체 typedef 예제 코드

```
typedef struct ListNode {
    int data;
    struct ListNode *link;
} Bic_ListNode;

ListNode *p; // 앞의 타입을 둘중 아무거나 써도 됨
Bic_ListNode *w; // 앞의 타입을 둘중 아무거나 써도 됨
```

- * 구조체 연습문제4) person이라는 구조체를 만들고, 문자배열로 된 이름, 사람의 나이를 나타내는 정수 값, 각 개인의 월급을 나타내는 float 값 등이 변수로 들어가게 만들고, 이중 struct를 사용

연습문제4 코드 - 첫 번째 방법

```
typedef struct person {  
    char name[10];  
    int age;  
    float pay;  
    struct {  
        int month;  
        int day;  
        int year;  
    };  
} company;  
  
void main(){  
    company p;  
    p.month = 12;  
}
```

연습문제4 코드 - 두 번째 방법

```
typedef struct person {  
    char name[10];  
    int age;  
    float pay;  
    struct dob{  
        int month;  
        int day;  
        int year;  
    }dob;  
} company;  
  
void main(){  
    company p;  
    p.dob.month = 12;  
}
```

연습문제4 코드 - 세 번째 방법

```
struct Dob {  
    int month;  
    int day;  
    int year;  
};  
  
typedef struct person {  
    char name[10];  
    int age;  
    float pay;  
    struct Dob dob  
} company;  
  
void main(){  
    company p;  
    p.dob.month = 12;  
}
```


- 과제) 구조체에 정의한 structure person을 사용하여 다음과 같은 프로그램을 작성해보시오
 - 몇 명의 사람 데이터를 입력 받을지 n의 값을 입력 받는다.
 - n명의 데이터를 저장할 공간을 동적으로 할당 받고 n명의 데이터를 입력 받아 저장한다.
 - 그 중 가장 나이가 많은 사람을 출력한다.
 - 평균 나이를 출력한다.

과제) 코드

```
#include <stdio.h>

typedef struct person {
    char name[10];
    int age;
    float pay;
    struct {
        int year;
        int month;
        int day;
    };
} company;

void main(){
    int n;
    int maxage, num=0;
    printf("사람 수 입력 >> ");
    scanf("%d", &n);
    company p[50];

    for (int i = 0; i < n; i++){
        printf("%i 번째 이름, 나이, 월급, 년, 월, 일을 순서대로 입력 >> \n", i+1);
        scanf("%s %d %f %d %d %d", &p[i].name, &p[i].age, &p[i].pay, &p[i].year,
        &p[i].month, &p[i].day);
        num += p[i].age;
    }

    for (int i = 0; i < n - 1; i++){
        for (int j = i + 1; j < n; j++){
            if (p[i].age > p[j].age){
                maxage = i;
            }
        }
    }
    printf("나이가 제일 많은 사람 >> %s %d세 %.1f원 %d년 %d월 %d일 >> \n",
    p[maxage].name, p[maxage].age, p[maxage].pay, p[maxage].year, p[maxage].month,
    p[maxage].day);
    printf("나이 평균 >> %d 세 \n", num / n);
}
```

```
C:\WINDOWS\system32\cmd.exe
1 번째 이름, 나이, 월급, 년, 월, 일을 순서대로 입력 >>
조광민 24 1000 1993 12 06
2 번째 이름, 나이, 월급, 년, 월, 일을 순서대로 입력 >>
김현철 20 500 1998 02 14
3 번째 이름, 나이, 월급, 년, 월, 일을 순서대로 입력 >>
최배성 21 10 1997 10 06
나이가 제일 많은 사람 >> 조광민 24세 1000.0원 1993년 12월 6일 >>
나이 평균 >> 21 세
계속하려면 아무 키나 누르십시오 . . .
```


○ 포인터(pointer) : 다른 변수의 주소를 가지고 있는 변수

*** 구조체 형태**

- `char *p` : 선언을 할 때는 *의 의미는 포인터 연산자
- `*p = 'B'` : 값을 바꿀 때는 *의 의미는 간접 참조 연산자

● 포인터: 다른 변수의 주소를 가지고 있는 변수

```
char a='A';
char *p;
p = &a;
```

● 포인터가 가리키는 내용의 변경: * 연산자 사용

```
*p = 'B';
```

주소	값	변수	기능
26	'B'	a	일반 변수
↖ (30)	26	p	포인터
↖	30	pp	이중 포인터

형태	값	설명
p	26	a의 주소
*p	'B'	a의 값
pp	30	p의 주소
*pp	26	p의 값

*** 포인터와 관련된 연산자**

- **& 연산자** : 변수의 주소를 추출
- *** 연산자** : 포인터가 가리키는 곳의 내용을 추출
- **** 연산자** : 포인터의 포인터(2중 포인터), 2차원 배열을 쓸 때 사용

● & 연산자: 변수의 주소를 추출

● * 연산자: 포인터가 가리키는 곳의 내용을 추출

```

p      // 포인터
*p     // 포인터가 가리키는 값
*p++  // 포인터가 가리키는 값을 가져온 다음, 포인터를 한칸 증가한다.
*p--  // 포인터가 가리키는 값을 가져온 다음, 포인터를 한칸 감소한다.
(*p)++ // 포인터가 가리키는 값을 증가시킨다.
    
```

```

int a; // 정수 변수 선언
int *p; // 정수 포인터 선언
int **pp; // 정수 포인터의 포인터 선언: 이중포인터
p = &a; // 변수 a와 포인터 p를 연결
pp = &p; // 포인터 p와 포인터의 포인터 pp를 연결
    
```

```
printf(" *p = %d, **pp = %d\n", *p, **pp );
```

이중포인터 pp 포인터 p 변수 a

```

void *p; // p는 아무것도 가리키지 않는 포인터
int *pi; // pi는 정수 변수를 가리키는 포인터
float *pf; // pf는 실수 변수를 가리키는 포인터
char *pc; // pc는 문자 변수를 가리키는 포인터
int **pp; // pp는 포인터를 가리키는 포인터
struct test *ps; // ps는 test 타입의 구조체를 가리키는 포인터
void (*f)(int) ; // f는 함수를 가리키는 포인터
    
```

* 포인터 연습문제1)

- 1번) int i = 10; int *p; p=&i, *p=8; // i값 : 8
- 2번) int i = 10; int *p; p=&i, (*p)--; // i값 : 9
- 3번) int a[10]; int *p; p=a, *p+=5; // 변경되는 배열의 요소 : a[0]=5, p=&a[1]; / 값을 넣고 배열증가
- 4번) int a[10]; int *p; p=a, ++p=5; // 변경되는 배열의 요소 : a[1]=5; p=&a[1]; / 배열증가, 값을 넣음
- 5번) int a[10]; int *p; p=a, (*p)++; // 변경되는 배열의 요소 : a[0]++;

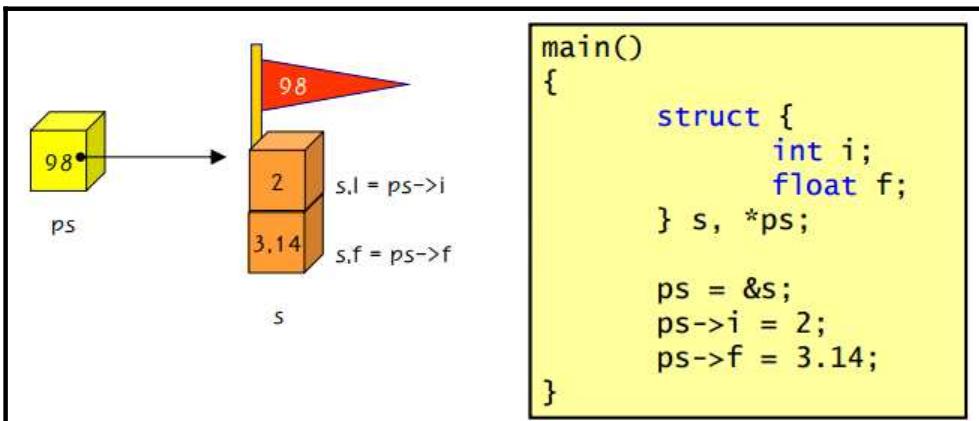
계산	p=5	p++	++p=5, (*p)++							
값	5	0	6	0	0	0	0	0	0	0

* 배열과 포인터 : 배열의 이름 : 사실상의 포인터와 같은 역할 // 맨 앞에 있는 주소를 넘김(시작 주소)

- 배열의 포인터가 ++가 되면 10이 11이 되지 않고 14가 됨(4byte씩 증가)

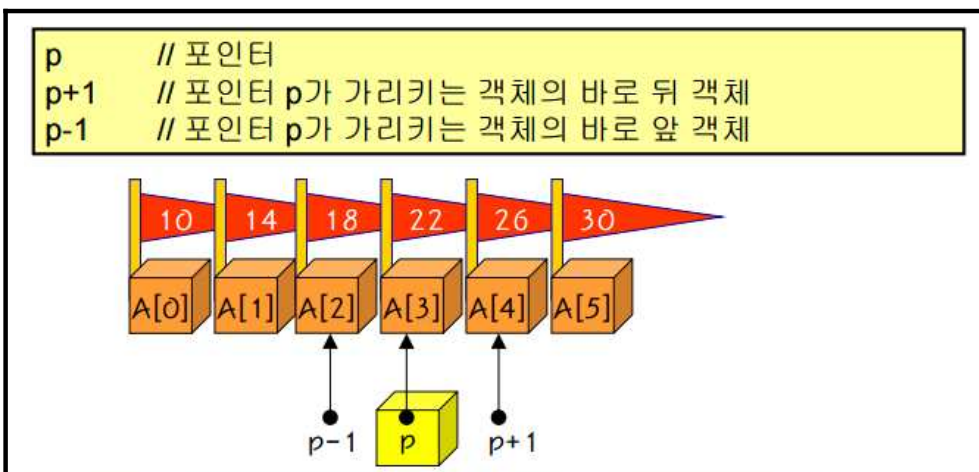
* 구조체의 포인터

- -> : 구조체의 요소에 접근하는 연산자(구조체의 포인터)
- (*ps).i => ps->i 로 간편하게 사용



* 포인터 연산

- 포인터에 대한 사칙연산 : 포인터가 가리키는 객체 단위로 계산



* 포인터 사용시 주의할 점

- 포인터가 아무것도 가리키고 있지 않을 때는 NULL로 설정
 - int *pi=NULL;
- 초기화가 안된 상태에서 사용 금지

```

main() {
    char *pc; // 포인터 pi는 초기화(바라보는 주소)가 안 되어 있음
    *pc = 'E'; // 포인터 타입간의 변환 시에는 명시적인 타입 변환 사용
}

```

- 포인터 타입 간의 변환 시에는 명시적인 타입 변환 사용

```

int *pi;
float *pf;
pf = (float *)pi; // float에 *를 붙이는 이유는 포인터의 값을 바꿔야하기 때문

```

○ 정적 메모리 할당

- * 메모리의 크기는 프로그램이 시작하기 전에 결정 // int형 변수 5개를 선언하면 실행 전에 20byte의 공간을 확보
- * 프로그램의 수행 도중에 그 크기가 변경될 수 는 없다 // ex) int buffer[100];

○ 동적 메모리 할당

- * 프로그램의 실행 도중에 메모리를 할당 받는 것
- * 필요한 만큼만 할당을 받고 또 필요할 때에 사용하고 반납
- * 메모리를 효율적으로 사용 가능

* 동적 메모리 할당 관련 라이브러리 함수

- malloc(int size) : 메모리 할당
- calloc : 메모리 할당 및 초기화 가능 (c++에 없음)
- realloc : 메모리를 할당받고 부족한 메모리를 재 할당 받음 (c++에 없음)
- size 바이트 만큼의 메모리 블록을 할당
- free(void ptr) : 메모리 할당 해제 // ptr : 변수
- sizeof(var) : 변수나 타입의 크기 반환(바이트 단위)

```
(char *)malloc(100); /* 100 바이트로 100개의 문자를 저장 */
(int *)malloc(sizeof(int)); /* 정수 1개를 저장할 메모리 확보 */
(struct Book *)malloc(sizeof(struct Book)); /* 하나의 구조체 생성 */
```

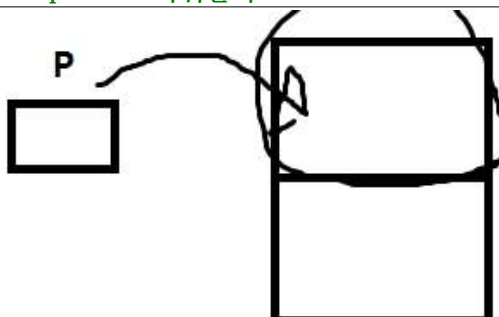
malloc과 sizeof에 대한 이해

```
struct Example{
    int number;
    char name[10];
};

void main()
{
    struct Example*p;
    p=(struct Example*)malloc(2*sizeof(struct Example));
    if(p==NULL){
        printf(stderr,"can't allocate memory\n");
        exit(1);
    }
    p->number=1;
    strcpy(p->name,"Park");
    (p+1)->number=2;
    strcpy((p+1)->name,"Kim");
    free(p);
}
```

- ex) p=(struct Example *)malloc(2*sizeof(struct Example)); //

//구조체를 저장/ 실제로 프로그램에서 sizeof해서 크기를 찍으면 크기가 다르기 때문에(더 크게 할당해줌) (struct Example)의 크기를 얻기 위해 sizeof를 썼는데 2개를 저장하고 싶어 2*를 하였음, 메모리를 할당을 받으면 void 타입이기 때문에 (struct Example *)를 써서 타입을 struct Example *로 바꿔준다



sizeof(struct Example)
(1개 선언시)

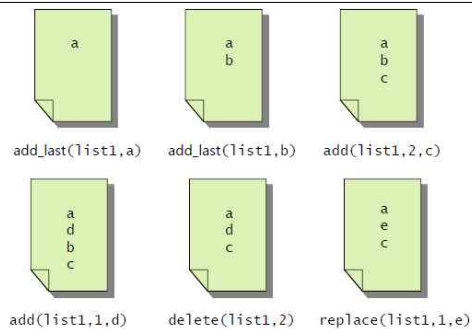
2*sizeof(struct Example)
(2개 선언시)

리스트

○ 리스트 연산 : 목록

※ 리스트 ADT

- 객체: n개의 element형으로 구성된 순서 있는 모임
- 연산:
 - 1) `add_last(list, item)` ::= 맨 끝에 요소를 추가한다.
 - 2) `add_first(list, item)` ::= 맨 처음에 요소를 추가한다.
 - 3) `add(list, pos, item)` ::= pos 위치에 요소를 추가한다.
 - 4) `delete(list, pos)` ::= pos 위치의 요소를 제거한다.
 - 5) `clear(list)` ::= 리스트의 모든 요소를 제거한다.
 - 6) `replace(list, pos, item)` ::= pos 위치의 요소를 item로 바꾼다.
 - 7) `is_in_list(list, item)` ::= item이 리스트 안에 있는지를 검사한다.
 - 8) `get_entry(list, pos)` ::= pos 위치의 요소를 반환한다.
 - 9) `get_length(list)` ::= 리스트의 길이를 구한다.
 - 10) `is_empty(list)` ::= 리스트가 비었는지를 검사한다.
 - 11) `is_full(list)` ::= 리스트가 꽉 찼는지를 검사한다.
 - 12) `display(list)` ::= 리스트의 모든 요소를 표시한다.



※ 리스트 구현 방법

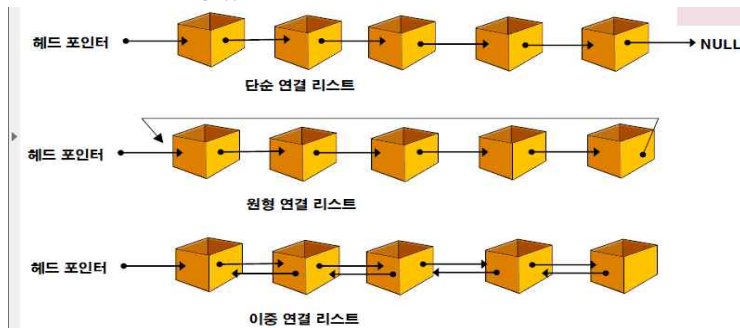
* 배열을 이용하는 방법

- 구현이 간단
- 삽입, 삭제 시 오버헤드 // 배열 중간에 데이터를 넣을 때 뒤에 있는 것을 모두 밀어야 함
- 항목의 개수 제한

* 연결리스트를 이용하는 방법

- 구현이 복잡
- 삽입, 삭제가 효율적
- 크기가 제한되지 않음

* 연결 리스트의 종류



* 노드 (node) : <항목, 주소> 쌍

- 리스트의 항목들을 노드라고 하는 곳에 분산하여 저장
- 구조체로 만들어서 저장 // <항목, 주소>를 받는 구조체
- 필요할 때마다 동적으로 생성
- 마지막 노드의 링크 값은 NULL (처음에는 링크 값이 NULL)

* 헤드 포인터(header pointer) : 리스트의 첫 번째 노드를 가리키는 변수 (중요) // L

※ ArrayListType

* 항목들의 타입은 element로 정의 (구조체)

- typedef int element; // element가 int타입으로 정의됨

ArrayListType - 선언

```
typedef int element;
typedef struct {
    element list[MAX_LIST_SIZE]; // 배열 정의
    int length; // 현재 배열에 저장된 항목들의 개수
} ArrayListType;
```

ArrayListType - 리스트 초기화

```
void init(ArrayListType *L)
{
    L->length = 0; // (*L).length = 0;
}
```

ArrayListType - 리스트 empty, full 연산

```
// 리스트가 비어 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_empty(ArrayListType *L) {
    return L->length == 0;
}

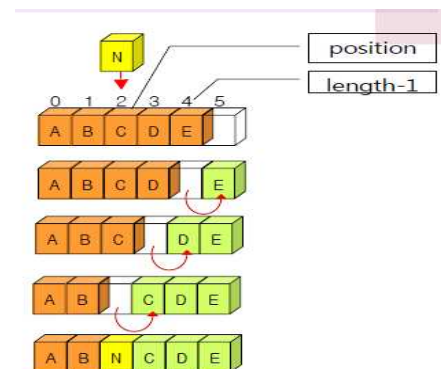
// 리스트가 가득 차 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_full(ArrayListType *L) {
    return L->length == MAX_LIST_SIZE;
}
```

ArrayListType - 리스트 초기화

```
void init(ArrayListType *L)
{
    L->length = 0; // (*L).length = 0;
}
```

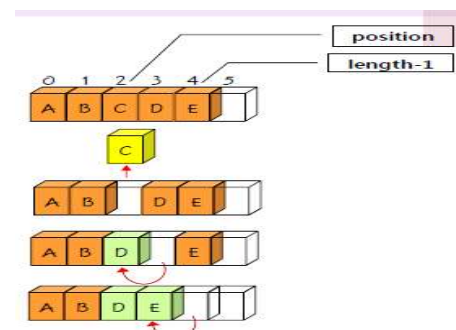
ArrayListType - 삽입 연산

```
// position: 삽입하고자 하는 위치
// item: 삽입하고자 하는 자료
void add(ArrayListType *L, int position, element item)
{
    if( !is_full(L) && (position >= 0) &&
        (position < L->length) )
    {
        int i;
        for(i=(L->length-1); i>=position;i--) {
            L->list[i+1] = L->list[i]; //오른쪽으로 이동
            L->list[position] = item;
            L->length++;
        }
    }
}
```



ArrayListType - 삭제 연산

```
// position: 삭제하고자 하는 위치
// 반환값: 삭제되는 자료
element delete(ArrayListType *L, int position) {
    int i;
    element item;
    if( position < 0 || position > L->length )
        error("위치 오류");
    item = L->list[position];
    for(i=position; i<(L->length-1);i++)
        L->list[i] = L->list[i+1]; //왼쪽으로 이동
    L->length--;
    return item;
}
```

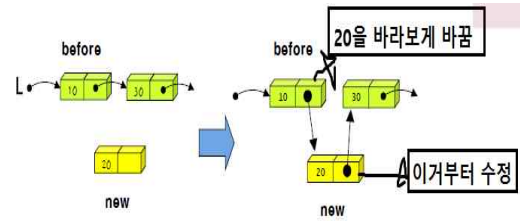


※ 단순 연결 리스트

- * 하나의 링크 필드를 이용하여 연결
- * 마지막 노드의 링크 값은 NULL
- 헤드 노드의 처음 값은 NULL이 아님

단순 연결 리스트 - 삽입

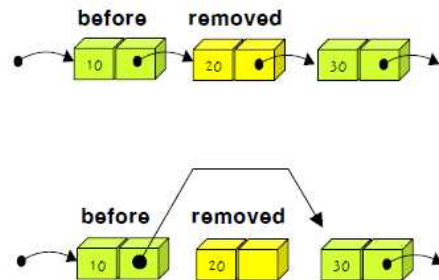
```
void insert_node(L, before, new){
    if (L==NULL){ // 노드를 사용하면 이게 필요 없음
        new->link=L; // 첫 노드의 값이 있기 때문
    } else {
        new->link = before->link;
        before->link = new->link;
    }
}
```



```
insert_node(before, new){
    new->link = before->link;
    before->link = new->link;
}
```

단순 연결 리스트 - 삭제

```
void remove_node(L, before, removed){
    if (L != NULL){
        before->link = removed->link;
        free(removed);
    }
}
```



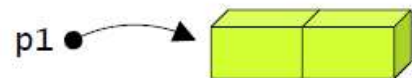
- * 데이터 필드 : 구조체로 정의

- * 링크 필드 : 포인터 사용

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

- * 노드의 생성 : 동적 메모리 생성 라이브러리 malloc 함수 이용

```
ListNode *p1;
p1 = (ListNode *)malloc(sizeof(ListNode));
```



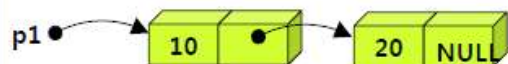
- * 데이터 필드와 링크 필드 설정

```
p1->data = 10;
p1->link = NULL;
```



- * 두 번째 노드 생성과 첫 번째 노드와의 연결

```
ListNode *p2;
p2 = (ListNode *)malloc(sizeof(ListNode));
p2->data = 20;
p2->link = NULL;
p1->link = p2;
```



* 단순연결 리스트 예제

- strcpy(tmp1->day, "월"); // c에서는 스트링이 없으므로 함수인 strcpy(스트링카피)를 사용

단순 연결 리스트 예제 - 요일 삽입

```
//연결리스트 pDays={월,화,수,목,금, 토, 일}
#include <stdio.h>
#include <stdlib.h>
#include <string>

typedef char element;
typedef struct ListNode {
    element day[3];
    struct ListNode *link;
} ListNode;

void printList( ListNode* p )
{
    ListNode *tmp = p;
    printf("연결리스트 출력:");
    while( tmp!= NULL ) {
        printf(" %s ", tmp->day );
        tmp = tmp->link;
    }
    printf("\n");
}

int main()
{
    ListNode *pDays = NULL;

    ListNode *tmp1, *tmp2, *tmp3, *tmp4;

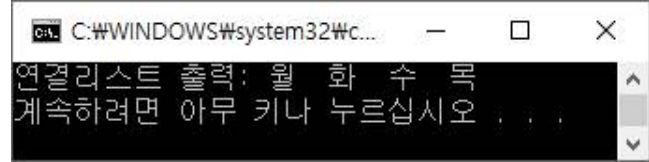
    // 메모리 할당
    tmp1 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp1->day, "월"); // c에서는 스트링이 없으므로 strcpy(스트링카피)를 사용
    pDays = tmp1;

    tmp2 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp2->day, "화");
    tmp2->link = NULL;
    tmp1->link = tmp2;

    tmp3 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp3->day, "수");
    tmp3->link = NULL;
    tmp2->link = tmp3;

    tmp4 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp4->day, "목");
    tmp4->link = NULL;
    tmp3->link = tmp4;

    printList( pDays );
    return 0;
}
```



단순 연결 리스트 예제2 - 요일 삽입 (insert)

```
//연결리스트 pDays={월,화,수,목,금, 토, 일}
```

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

```
typedef char element;
typedef struct ListNode {
    element day[3];
    struct ListNode *link;
} ListNode;
```

```
void printList( ListNode* p )
{
    ListNode *tmp = p;
    printf("연결리스트 출력:");
    while( tmp!= NULL ) {
        printf(" %s ", tmp->day );
        tmp = tmp->link;
    }
    printf("\n");
}
```

```
// phead: 리스트의 헤드 포인터의 포인터
// before : 선행 노드
// new_node : 삽입될 노드
```

```
void insert_node(ListNode **phead, ListNode *before, ListNode *new_node)
{
    if (*phead == NULL){ // 공백리스트인 경우
        new_node->link = NULL;
        *phead = new_node;
    }
    else if (before == NULL){ // before가 NULL이면 첫번째 노드로 삽입
        new_node->link = *phead;
        *phead = new_node;
    }
    else { // before 다음에 삽입
        new_node->link = before->link;
        before->link = new_node;
    }
}
```

```
int main()
{
```

```
    ListNode *pDays = NULL;
```

```
    ListNode *tmp1, *tmp2, *tmp3, *tmp4;
```

```
    // 메모리 할당
```

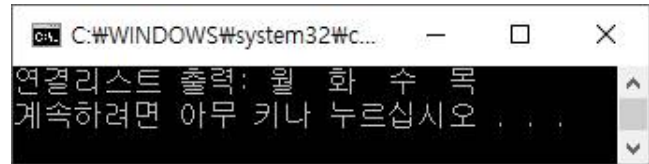
```
    tmp1 = (ListNode*)malloc(sizeof(ListNode)); // 첫 번째 방법
    strcpy_s(tmp1->day, "월"); // c에서는 스트링이 없으므로 strcpy(스트링카피)를 사용
    insert_node(&pDays, NULL, tmp1); // 두 번째 방법 (insert 함수 사용)
```

```
    tmp2 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp2->day, "화");
    tmp2->link = NULL;
    insert_node(&pDays, tmp1, tmp2);
```

```
    tmp3 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp3->day, "목");
    tmp3->link = NULL;
    insert_node(&pDays, tmp2, tmp3);
```

```
    tmp4 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp4->day, "수");
    tmp4->link = NULL;
    insert_node(&pDays, tmp2, tmp4);
```

```
    printList( pDays );
    return 0;
}
```



단순 연결 리스트 예제3 - 요일 탐색 (insert2 / search)

//연결리스트 pDays={월,화,수,목,금, 토, 일}

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

```
typedef char element;
typedef struct ListNode {
    element day[3];
    struct ListNode *link;
} ListNode;
```

```
void printList( ListNode* p )
{
```

```
    ListNode *tmp = p->link;
    //p가아닌 p->link부터 시작
    printf("연결리스트 출력:");
    while( tmp!= NULL ) {
        printf(" %s ", tmp->day );
        tmp = tmp->link;
    }
```

```
    printf("\n");
}
```

```
// phead: 리스트의 헤드 포인터의 포인터
// before : 선행 노드
// new_node : 삽입될 노드
```

```
void insert_node( ListNode *before, ListNode *new_node)
{
    // if문이 필요없어짐
    new_node->link = before->link;
    before->link = new_node;
} // 헤드포인트가 아닌 노드부터 시작하므로 insert_node의 매개변수가 2개만 있으면 된다.
```

```
ListNode *search(ListNode *head, char x[])
{
    ListNode *p = head->link;
    while (p != NULL){
        if (strcmp(p->day, x) == 0) return p; // 탐색 성공
        p = p->link;
    }
    return p; // 탐색 실패일 경우 NULL 반환
}
```

```
int main()
{
```

```
    ListNode *pDays;
```

```
    pDays = (ListNode*)malloc(sizeof(ListNode)); // pDays의 메모리를 할당한다.
    pDays->link = NULL; // pDays의 link를 NULL로 설정
```

```
    ListNode *new_node, *before;
```

```
    // 메모리 할당
```

```
    new_node = (ListNode*)malloc(sizeof(ListNode));
```

```
    strcpy_s(new_node->day, "월"); // c에서는 스트링이 없으므로 strcpy(스트링카피)를 사용
```

```
    new_node->link = NULL;
```

```
    insert_node( pDays, new_node);
```

```
    /*화*/
```

```
    new_node = (ListNode*)malloc(sizeof(ListNode));
```

```
    strcpy_s(new_node->day, "화");
```

```
    new_node->link = NULL;
```

```
    //search
```

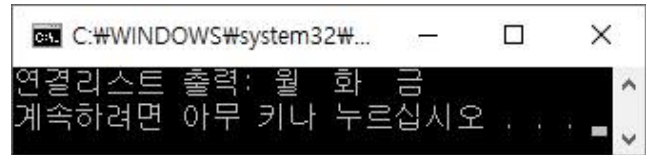
```
    before = search( pDays, "월"); // 월요일을 찾아서 before에 넣어줌
```

```
    insert_node( before, new_node);
```

```
    printList( pDays );
```

```
    return 0;
```

```
}
```



단순 연결 리스트 예제3 - 요일 삭제 (remove)

```
//연결리스트 pDays={월,화,수,목,금, 토, 일}
```

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

```
typedef char element;
typedef struct ListNode {
    element day[3];
    struct ListNode *link;
} ListNode;
```

```
void printList( ListNode* p )
```

```
{
    ListNode *tmp = p->link;
    // p부터가 아닌 p->link부터 시작해야한다.
    printf("연결리스트 출력:");
    while( tmp!= NULL ) {
        printf(" %s ", tmp->day );
        tmp = tmp->link;
    }
    printf("\n");
}
```

```
void insert_node( ListNode *before, ListNode *new_node)
```

```
{
    // if문이 필요없어진
    new_node->link = before->link;
    before->link = new_node;
}
```

```
ListNode *search(ListNode *head, char x[])
```

```
{
    ListNode *p = head->link;
    while (p != NULL){
        if (strcmp(p->day, x) == 0) return p; // 탐색 성공
        p = p->link;
    }
    return p; // 탐색 실패일 경우 NULL 반환
}
```

```
void remove_node(ListNode *before, ListNode *removed)
```

```
{
    before->link = removed->link;
    free(removed);
}
```

```
int main()
```

```
{
    ListNode *pDays;
```

```
    pDays = (ListNode*)malloc(sizeof(ListNode));
    pDays->link = NULL;
```

```
    ListNode *new_node, *before, *remove;
```

```
    // 메모리 할당
    /*월*/
```

```
    new_node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(new_node->day, "월"); // c에서는 스트링이 없으므로 strcpy(스트링카피)를 사용
    new_node->link = NULL;
    insert_node( pDays, new_node);
```

```
    /*금*/
```

```
    new_node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(new_node->day, "금");
    new_node->link = NULL;
```

```
    //search
```

```
    before = search( pDays, "월"); // 월요일을 찾아서 before에 넣어줌
    insert_node( before, new_node);
```

```
    /*화*/
```

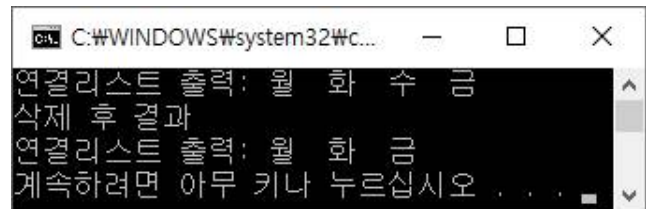
```
    new_node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(new_node->day, "화");
    new_node->link = NULL;
    before = search(pDays, "월");
    insert_node(before, new_node);
```

```
    new_node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(new_node->day, "수");
    new_node->link = NULL;
    before = search(pDays, "화");
    insert_node(before, new_node);
```

```
    printList(pDays);
```

```
    printf("삭제 후 결과\n");
    remove = search(pDays, "수");
    before = search(pDays, "화");
    remove_node(before, remove);
```

```
    printList( pDays );
    return 0;
}
```

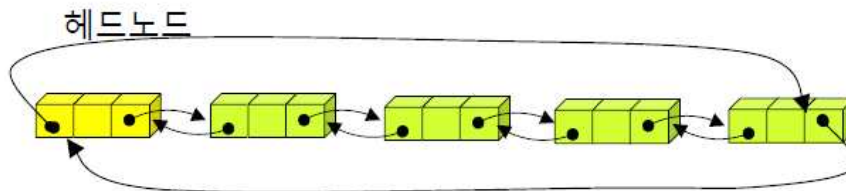


※ 원형 연결 리스트 (생략)

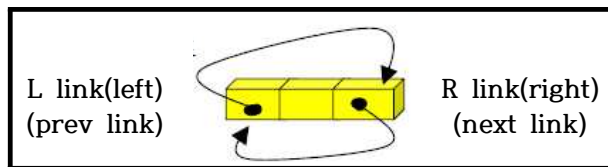
- * 맨 마지막의 new->link = head->link 로 넣어준다.

※ 이중 연결 리스트

- * 단순 연결 리스트의 문제점 : 선행 노드를 찾기가 힘들다
 - 삽입이나 삭제 시에는 반드시 선행 노드가 필요
- * 이중연결 리스트 : 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
 - 링크가 양방향이므로 양방향으로 검색이 가능
 - 단점은 공간을 많이 차지하고 코드가 복잡
- * 실제 사용되는 이중 연결 리스트의 형태
 - 헤드노드 + 이중연결 리스트 + 원형연결 리스트

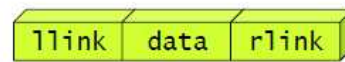


- * 헤드노드 : 데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드
 - 헤드 포인터와의 구별이 필요
 - 공백상태에서는 헤드 노드만 존재



* 이중연결리스트에서의 노드 구조

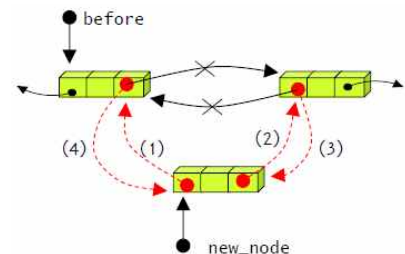
```
typedef int element;
typedef struct DlistNode {
    element data;
    struct DlistNode *llink;
    struct DlistNode *rlink;
} DlistNode;
```



* 삽입연산

이중 연결 리스트 - 삽입

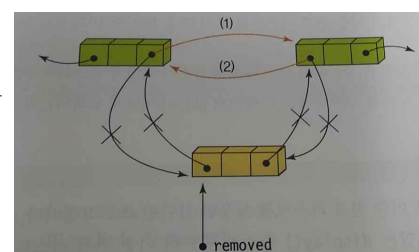
```
// 노드 new_node를 노드 before의 오른쪽에 삽입한다.
void dinsert_node(DlistNode *before, DlistNode
*new_node){
    new_node->llink = before;           // (1)
    new_node->rlink = before->rlink;    // (2)
    before->rlink->llink = new_node;    // (3)
    before->rlink = new->rlink;         // (4)
}
```



이중 연결 리스트 - 삭제

```
void remove_node(DlistNode *before, DlistNode *removed){
    before->rlink = removed->rlink;
    removed->rlink->llink = before;
}

// 노드 removed를 삭제한다.
void dremove_node(DlistNode *phead_node, DlistNode
*removed){
    if(removed == phead_node) return;
    removed->llink->rlink = removed->rlink; // (1)
    removed->rlink->llink = removed->llink; // (2)
}
```



이중 연결 리스트 예제

```
//연결리스트 pDays={월,화,수,목,금, 토, 일}
```

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

```
typedef char element;
typedef struct DlistNode {
    element day[3];
    struct DlistNode *prev;
    struct DlistNode *next;
} DlistNode;
```

```
void printList( // p는 headnode (pDays)
DlistNode* p )
{
    DlistNode *tmp = p->next;
    printf("\n연결리스트 출력:");
    while( tmp!= p ) {
        printf(" %s ", tmp->day );
        tmp = tmp->next;
    }
}
```

```
// before : 선행 노드
// new_node : 삽입될 노드
```

```
void dinsert_node(DlistNode *before, DlistNode *new_node)
{
    new_node->prev = before;
    new_node->next = before->next;
    before->next->prev = new_node;
    before->next = new_node;
}
```

```
// phead : 헤드 포인터에 대한 포인터
```

```
// removed: 삭제될 노드
```

```
void dremove_node(DlistNode *phead_node, DlistNode *removed)
{
    if( removed == phead_node ) return;
    removed->prev->next = removed->next;
    removed->next->prev = removed->prev;
    free(removed);
}
```

```
DlistNode *search(DlistNode *head, char x[])
```

```
{
    DlistNode *p = head->next;
    while( p != head ){
        if( strcmp(p->day, x) ==0) return p; // 탐색 성공
        p = p->next;
    }
    return p; // 탐색 실패일 경우 NULL 반환
}
```

```
void main()
```

```
{
    DlistNode *pDays, *new_node, *before, *remove;
```

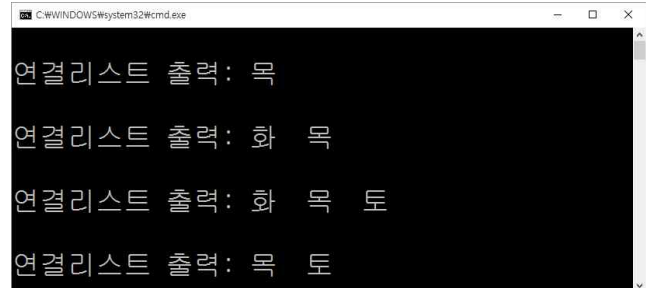
```
    pDays = (DlistNode*)malloc(sizeof(DlistNode));
    pDays->next = pDays;
    pDays->prev = pDays;
```

```
    new_node = (DlistNode*)malloc(sizeof(DlistNode));
    strcpy( new_node->day, "목");
    new_node->next = NULL;
    new_node->prev = NULL;
    dinsert_node( pDays, new_node );
    printList( pDays );
```

```
    new_node = (DlistNode*)malloc(sizeof(DlistNode));
    strcpy( new_node->day, "토");
    new_node->next = NULL;
    new_node->prev = NULL;
    before = search( pDays, "목");
    dinsert_node( before, new_node );
    printList( pDays );
```

```
    new_node = (DlistNode*)malloc(sizeof(DlistNode));
    strcpy( new_node->day, "화");
    new_node->next = NULL;
    new_node->prev = NULL;
    dinsert_node( pDays, new_node );
    printList( pDays );
```

```
    remove = search( pDays, "화");
    dremove_node( pDays, remove );
    printList( pDays );
}
```



```
C:\WINDOWS\system32\cmd.exe

연결리스트 출력: 목
연결리스트 출력: 화 목
연결리스트 출력: 화 목 토
연결리스트 출력: 목 토
```

이중 연결 리스트 연습 문제 - 라인에디터 // 23일 자정까지 작성

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

```
typedef char element;
typedef struct DListNode {
    int line;
    element day[200];
    struct DListNode *prev;
    struct DListNode *next;
} DListNode;
```

```
void printList(DListNode* p)
{ // p는 headnode (pDays)
    DListNode *tmp = p->next;
    printf("\n연결리스트 출력 >>> ");
    while (tmp != p) {
        printf("< %s >",
tmp->day);
        tmp = tmp->next;
    }
    printf("\n-----\n\n");
}
```

```
// before : 선행 노드
// new_node : 삽입될 노드
void dinsert_node(DListNode *before, DListNode *new_node)
{
    new_node->prev = before;
    new_node->next = before->next;
    before->next->prev = new_node;
    before->next = new_node;
}
```

```
// phead : 헤드 포인터에 대한 포인터
// removed: 삭제될 노드
void dremove_node(DListNode *phead_node, DListNode *removed)
{
    if (removed == phead_node) return;
    removed->prev->next = removed->next;
    removed->next->prev = removed->prev;
    free(removed);
}
```

```
DListNode *search(DListNode *head, char x[])
{
    DListNode *p = head->next;
    while (p != head){
        if (strcmp(p->day, x) == 0) return p; // 탐색 성공
        p = p->next;
    }
    return p; // 탐색 실패일 경우 NULL 반환
}
```

```
void main()
{
    DListNode *pDays, *new_node, *before, *remove;
```

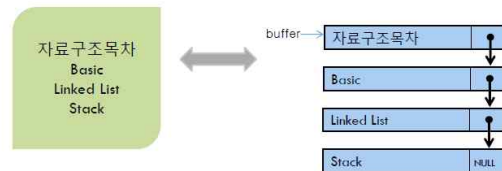
```
    char select;
    char input[200];
    char where[200];
```

```
    pDays = (DListNode*)malloc(sizeof(DListNode));
    pDays->next = pDays;
    pDays->prev = pDays;
```

```
    while (true){
        printf("'a' : Add \n");
        printf("'r' : Remove \n");
        printf("'e' : Exit \n");
        printf("Input Select >>> ");
        scanf("%c", &select);
        fflush(stdin);
        switch (select){
            case 'a':
                new_node = (DListNode *)malloc(sizeof(DListNode));
                printf("Input String >>> ");
                fgets(input, sizeof(input), stdin);
                input[strlen(input) - 1] = '\0';
                //scanf("%s", &input);
                strcpy(new_node->day, input);
                new_node->next = NULL;
```

도전 문제-라인에디터

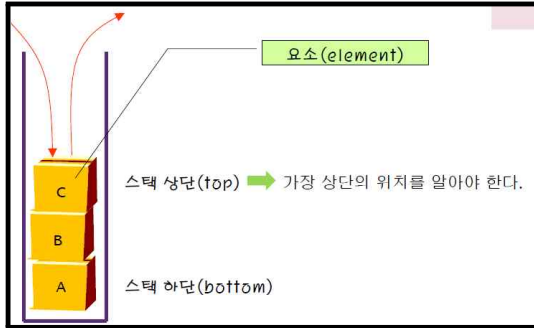
- 라인단위로 텍스트를 입력하거나 삭제할 수 있는 단순한 에디터
- 라인 단위로 이루어지므로 커서를 사용하지 않는다.
- 입력되는 라인의 개수가 정해지지 않았으므로 연결리스트 사용
- 삽입
 - 라인의 번호와 라인 내용을 입력 받아 삽입
- 삭제
 - 삭제하고자 하는 라인 번호를 입력 받아 삭제



스택

○ 스택 정의

- * 스택이란? 쌓는거
- * 후입선출 : 가장 최근에 들어온 데이터가 가장 먼저 나감
- * 스택의 구조 : 삽입/삭제 할 때 top의 값만 바뀜

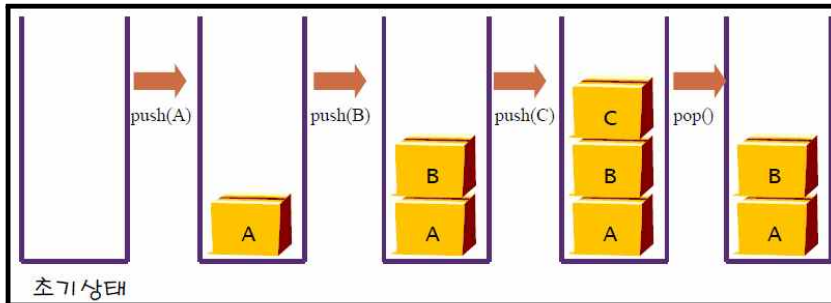


* 스택 추상데이터타입 (ADT)

- 객체: n개의 element형의 요소들의 선형 리스트
- 연산:
 - create() ::= 스택을 생성한다.
 - is_empty(s) ::= 스택이 비어있는지를 검사한다.
 - is_full(s) ::= 스택이 가득 찼는가를 검사한다.
 - push(s, e) ::= 스택의 맨 위에 요소 e를 추가한다.
 - pop(s) ::= 스택의 맨 위에 있는 요소를 삭제한다.
 - peek(s) ::= 스택의 맨 위에 있는 요소를 삭제하지 않고 반환한다.

※ 스택의 연산

- * Push() : 스택에 데이터를 추가
- * Pop() : 스택에서 데이터를 삭제

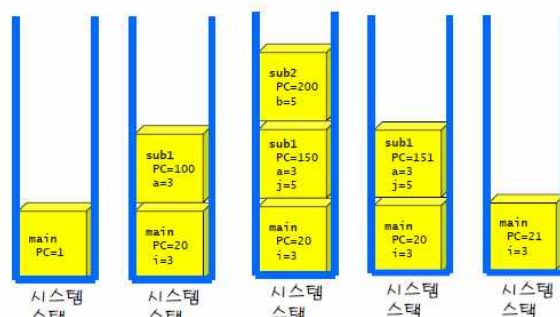


- * is_empty(s) : 스택이 공백상태인지 검사
- * is_full(s) : 스택이 포화상태인지 검사
- * create() : 스택을 생성
- * peek(s) : 요소를 스택에서 삭제하지 않고 보기만 하는 연산
// (참고) pop 연산은 요소를 스택에서 완전히 삭제하면서 가져온다.

※ 스택의 용도

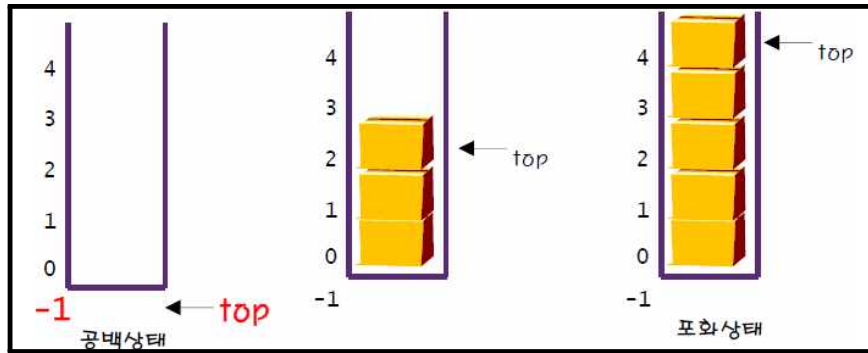
- * 입력과 역순의 출력이 필요한 경우
 - 에디터에서 되돌리기(undo) 기능
 - 함수 호출에서 복귀주소 기억
 - 웹 브라우저의 뒤로 버튼

```
1  int main() {
    int i=3;
20  sub1(i);
    ...
    }
100 int sub1(int a) {
    int j=5;
150  sub2(j);
    ...
    }
200 void sub2(int b) {
    ...
    }
```

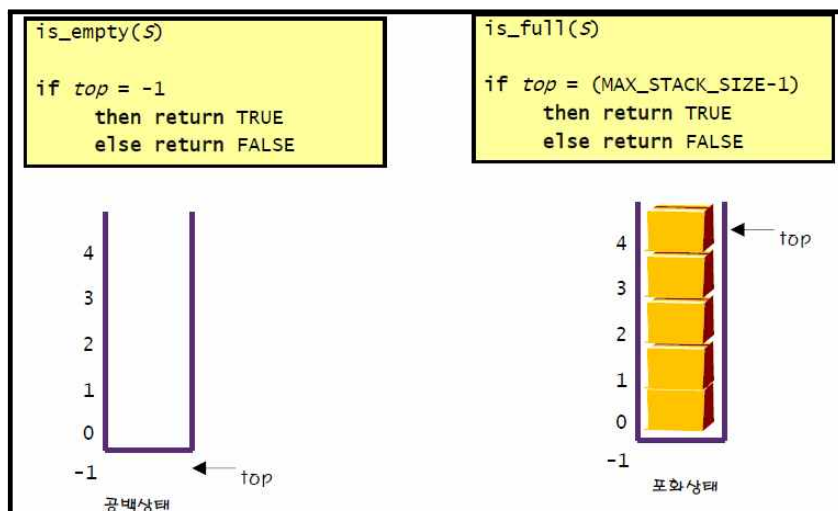


○ 배열을 이용한 스택의 구현

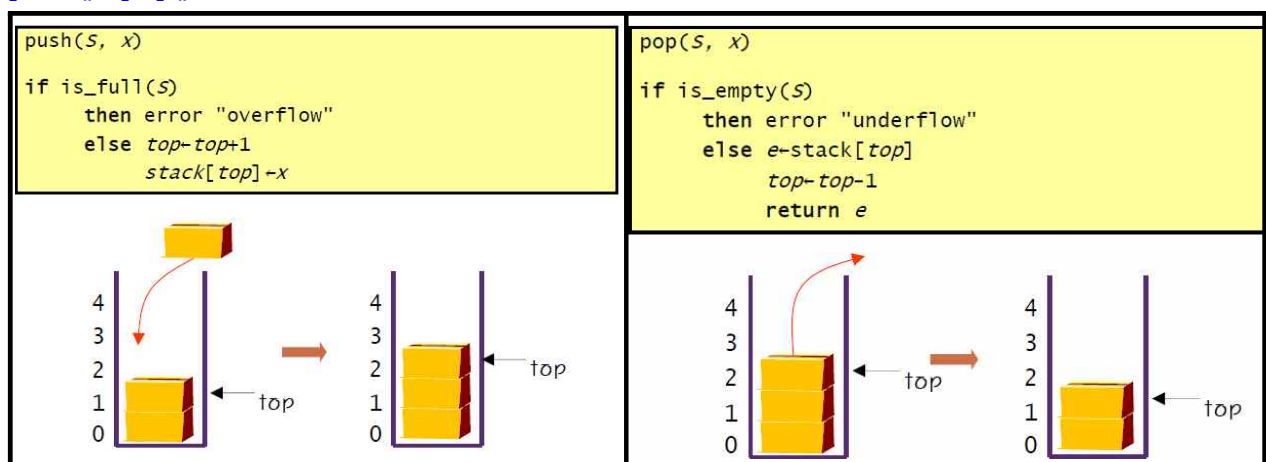
- * 1차원 배열 `stack[]`
- * 스택에서 가장 최근에 입력되었던 자료를 가리키는 **top** 변수
- * 가장 먼저 들어온 요소는 `stack[0]`에, 가장 최근에 들어온 요소는 `stack[top]`에 저장
- * 스택이 공백상태이면 `top`은 -1



※ `is_empty()`, `is_full()`

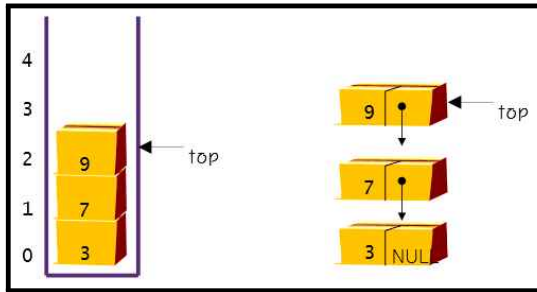


※ `push()`, `pop()`



○ 연결된 스택 : 연결리스트를 이용하여 구현한 스택

- * 장점 : 크기가 제한되지 않음
- * 단점 : 구현이 복잡하고 삽입, 삭제 시간이 오래 걸림



※ 연결된 스택 정의

```
typedef int element;
typedef struct StackNode {
    element item;
    struct StackNode *link;
} StackNode;
typedef struct {
    StackNode *top;
} LinkedStackType;
```

요소의 타입

노드의 타입

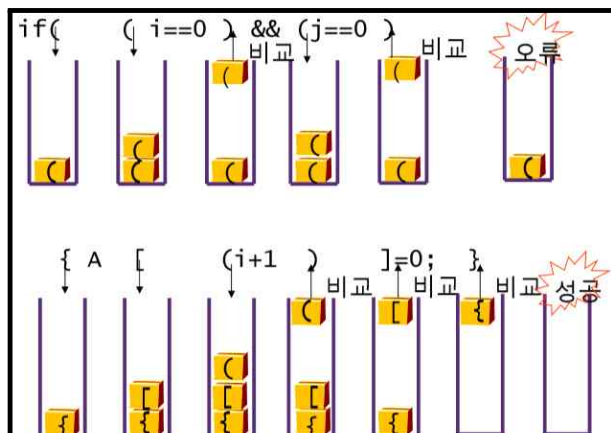
연결된 스택의 관련 데이터

※ 연결된 스택 push(), pop()

```
void push(LinkedStackType *s, element item) // 삽입 함수
{
    StackNode *temp=(StackNode *)malloc(sizeof(StackNode));
    if( temp == NULL ){
        fprintf(stderr, "메모리 할당에러\n");
        return;
    }
    else{
        temp->item = item;
        temp->link = s->top;
        s->top = temp;
    }
}
```

```
element pop(LinkedStackType *s) // 삭제 함수
{
    if( is_empty(s) ) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else{
        StackNode *temp=s->top;
        int item = temp->item;
        s->top = s->top->link;
        free(temp);
        return item;
    }
}
```

※ 스택 응용 : 괄호 검사



※ 괄호 검사 알고리즘

스택 예제 - 괄호 검사

```
#include "stack.h"
#include <string.h>

// 스택 초기화 함수
void init(StackType *s) {
    s->top = -1;
}

// 공백 상태 검출 함수
int is_empty(StackType *s){
    return (s->top == -1);
}

// 포화 상태 검출 함수
int is_full(StackType *s) {
    return (s->top == (MAX_STACK_SIZE - 1));
}

void push(StackType *s, element item) {
    // 삽입함수
    if (is_full(s))
    {
        printf("스택 포화 상태 \n");
        exit(0);
    }
    else
    {
        s->top++;
        s->stack[s->top] = item;
    }
}

element pop(StackType *s) { // 삭제함수
    if (is_empty(s))
    {
        printf("스택 공백 상태\n");
        exit(0);
    }
    else
    {
        element tmp = s->stack[s->top];
        s->stack[s->top] = 0;
        s->top--;
        return tmp;
    }
}

element peek(StackType *s) { // 피크함수
    if (is_empty(s)) {
        printf("스택 공백 에러\n");
        exit(0);
    }
    else return s->stack[s->top];
}

int check_matching(char in[])
{
    StackType s;
    char ch, open_ch;
    int i, n = strlen(in);

    init(&s); // 스택 초기화

    for (i = 0; i < n; i++) { // i=0: while( in[i]
        ch = in[i];
        switch (ch){
            case '(':
            case '[':
            case '{':
                push(&s, ch);
                break;
            case ')':
            case ']':
            case '}':
                if (is_empty(&s)) return
                    false;
                open_ch = pop(&s);
                if ((open_ch == '(' && ch
                    != ')') ||
                    (open_ch == '['
                    && ch != ']') ||
                    (open_ch == '{'
                    && ch != '}')) return false;
                break;
            default:
                if (!is_empty(&s)) return false;
                return true;
        }
    }
}
```

○ 수식의 계산

※ 수식의 표기 방법

- * 전위(prefix), 중위(infix), 후위(postfix)

중위 표기법	전위 표기법	후위 표기법
2+3*4	+2*34	234*+
a*b+5	+5*ab	ab*5+
(1+2)+7	+7+12	12+7+

※ 컴퓨터에서의 수식 계산 순서

- * 중위표기식 -> 후위표기식 -> 계산
- * ex) 2+3*4 -> 234*+ -> 14
- * 모두 스택을 사용

※ 후위 표기식 계산

- 1) 수식을 왼쪽에서 오른쪽으로 스캔하여,
- 2) 피연산자이면 스택에 저장하고,
- 3) 연산자이면 필요한 수만큼의 피연산자를 스택에서 꺼내 연산을 실행하고,
- 4) 연산의 결과를 다시 스택에 저장

● (예) 82/3-32*+

스택에서 변화되는 값

토 컨	스 택						
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	8						
2	8	2					
/	4						
3	4	3					
-	1						
3	1	3					
2	1	3	2				
*	1	6					
+	7						

※ 중위 표기식 -> 후위 표기식

* 중위 표기와 후위 표기

- * 중위 표기법과 후위 표기법의 공통점은 피연산자의 순서는 동일
- * 연산자들의 순서만 다름 (우선순위순서) -> 연산자만 스택에 저장했다가 출력하면 됨

* 알고리즘

- * 피연산자를 만나면 그대로 출력
- * 연산자 op를 만나면 '스택'의 top에 있는 연산자와 비교
 - '스택'의 top에 있는 연산자 우선순위가 높으면 top에 있는 연산자를 출력, 연산자 op는 스택에 삽입
- * 왼쪽 괄호는 우선순위가 가장 낮은 연산자로 취급
- * 오른쪽 괄호가 나오면 스택에서 왼쪽 괄호 위에 쌓여있는 모든 연산자를 출력
- * 우선순위 : (< +, - < *, /
- * 우선순위가 작은 것이 스택에 들어갈 때 자기보다 작은 것이 나올 때까지 pop해줌
- * ex) 2+3*4 -> 234*+
- * ex) 4/(3-1)+5 -> 431-/5+
- * ex) 8/(2-3)+3*2 -> 8(23-)/32*+
- * ex) 2*5-6/2+5-4 -> 25*62/-5+4-

스택 예제 - 괄호 검사 - 수식 계산	
후위 표기식 계산	중위표기식 -> 후위 표기식
<pre> int eval(char exp[]) { int op1, op2, value; int len = strlen(exp); char ch; StackType s; init(&s); for (int i = 0; i < len; i++){ ch = exp[i]; if (ch != '+' && ch != '-' && ch != '*' && ch != '/') // 입력이 피연산자이면 { value = ch - '0'; push(&s, value); } else{ //연산자이면 피연산자를 op2 = pop(&s); op1 = pop(&s); switch (ch){ //연산을 case '+': value = op2 + op1; push(&s, value); break; case '-': value = op2 - op1; push(&s, value); break; case '*': value = op2 * op1; push(&s, value); break; case '/': value = op2 / op1; push(&s, value); break; case '%': value = op2 % op1; push(&s, value); break; } value = pop(&s); return value; } } // 우선순위 int prec(int op) { switch (op) { case '(': case ')': return 0; case '+': case '-': return 1; case '*': case '/': return 2; } return -1; } </pre>	<pre> void infix_to_postfix(char exp[]) { int i = 0; char ch, top_op; int len = strlen(exp); StackType s; init(&s); // 스택 초기화 for (i = 0; i < len; i++){ ch = exp[i]; // 연산자이면 switch (ch){ case '+': case '-': case '*': case '/': // 연산자 while (!is_empty(&s) && (prec(ch) <= prec(peek(&s)))) // 스택에 있 는 연산자의 우선순위가 더 크거나 같으면 출력 printf("%c", pop(&s)); push(&s, ch); break; case '(': // 왼쪽 괄호 push(&s, ch); break; case ')': // 오른쪽 괄호 top_op = pop(&s); // 왼쪽 괄호를 만날때 while (top_op != '('){ printf("%c", top_op); top_op = pop(&s); } break; default: // 피연산자 printf("%c", ch); break; } } while (!is_empty(&s)) // 스택에 저 장된 연산자들 출력 printf("%c", pop(&s)); } </pre>