

# 자료구조

2주차(16. 09. 08)

## 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 n에 대한 함수

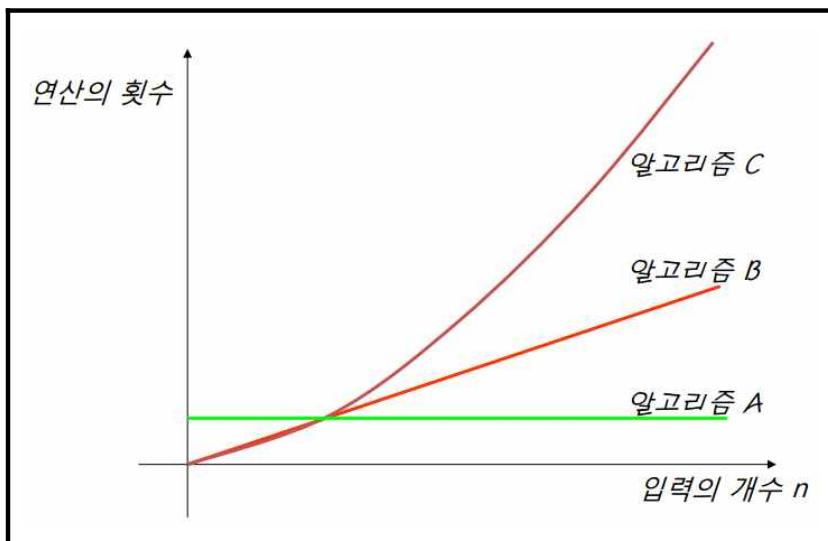
// ex) 연산의 수 = 8  $\rightarrow 3n+2$

### \* 복잡도 분석의 예 (n을 n번 더하는 문제)

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

\* 알고리즘 A가 제일 좋음

- \* 상수시간대 : 입력 개수와 연산의 횟수가 고정되어 있는 것

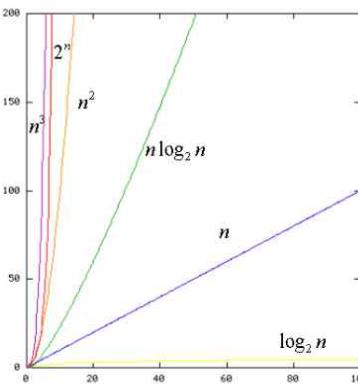


- \* 빅오 표기법 : 연산의 횟수를 대략적으로 표기한 것 / 데이터의 값에 따라 수행시간이 달라짐

- 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

- $O(1)$  : 상수형
- $O(\log n)$  : 로그형
- $O(n)$  : 선형
- $O(n\log n)$  : 로그선형
- $O(n^2)$  : 2차형
- $O(n^3)$  : 3차형
- $O(n^k)$  : k차형
- $O(2^n)$  : 지수형
- $O(n!)$  : 팩토리얼형



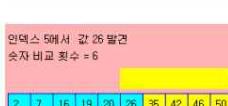
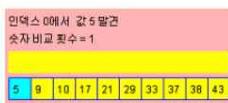
시간복잡도	$n$						
	1	2	4	8	16	32	
1	1	1	1	1	1	1	
$\log n$	0	1	2	3	4	5	
$n$	1	2	4	8	16	32	
$n \log n$	0	2	8	24	64	160	
$n^2$	1	4	16	64	256	1024	
$n^3$	1	8	64	512	4096	32768	
$2^n$	2	4	16	256	65536	4294967296	
$n!$	1	2	24	40326	20922789888000	$26313 \times 10^{33}$	

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



### \* 연습문제 2 : 코드의 각 명령문의 수행 횟수를 계산하여 시간 복잡도 함수를 계산하시오

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&lt;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&lt;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 &lt;= n)         i = i*b; // log n 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 나누어진다.

#### 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 :  $fib(n) = fib(n-2) + fib(n-1)$

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

#### 하노이 탑 코드

```
#include <stdio.h>
void hanoi(int n, char from, char tmp, char to)
{
    if (n == 1)
        printf("원판 1을 %c에서 %c으로 옮긴다.\n", from, to);
    else {
        hanoi(n - 1, from, to, tmp);
        printf("원판 %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);
}
```

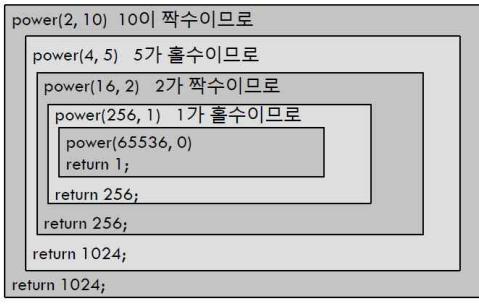
\* **거듭제곱 값**

- ex1)  $3^9 = 3*3^8 = 3*(3^2)^4 = 3*((3^2)^2)^2 = 3*((9^2)^2)^2 = 3*(81)^2 = 3*6561*6561^0 = 3*6561*1 = 19683$
- 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(x2, n/2);
else if n이 홀수 then return x*power(x2, (n-1)/2);
```

##### • $2^{10}$ 을 계산하는 과정

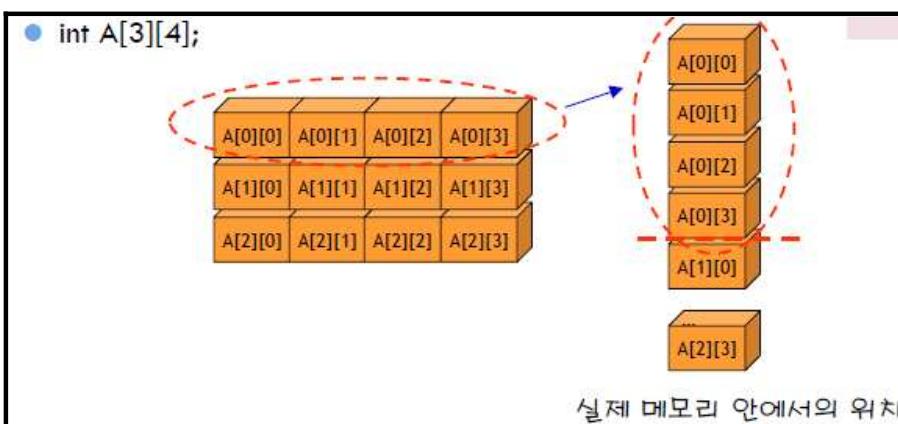


\* **이진탐색** : 가운데 수를 기준으로 반씩 쪼개서 탐색함

## 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)별(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

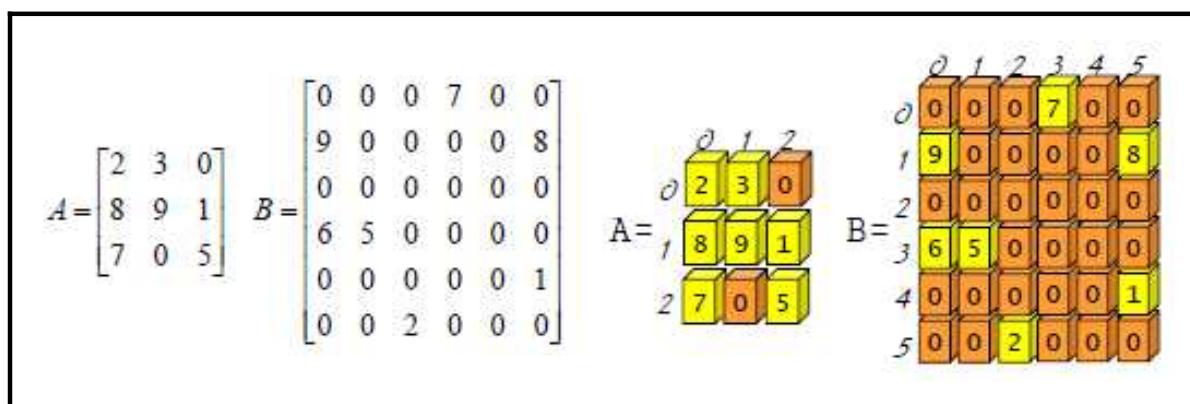
- 두 번째 방법

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

\* 희소행렬 : 대부분의 항들이 0인 배열

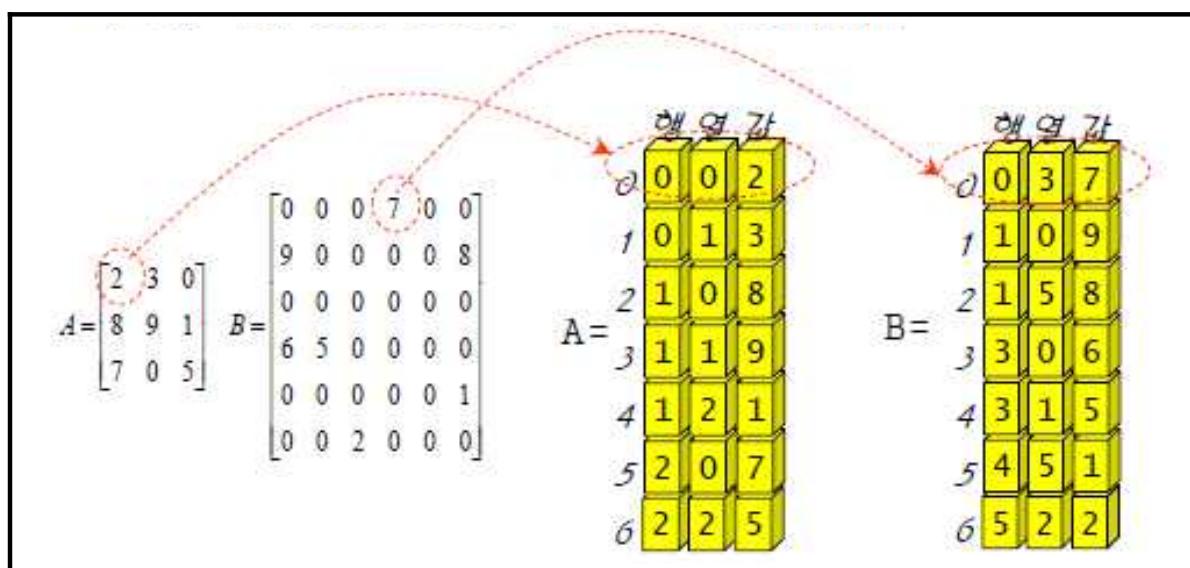
\* 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법

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



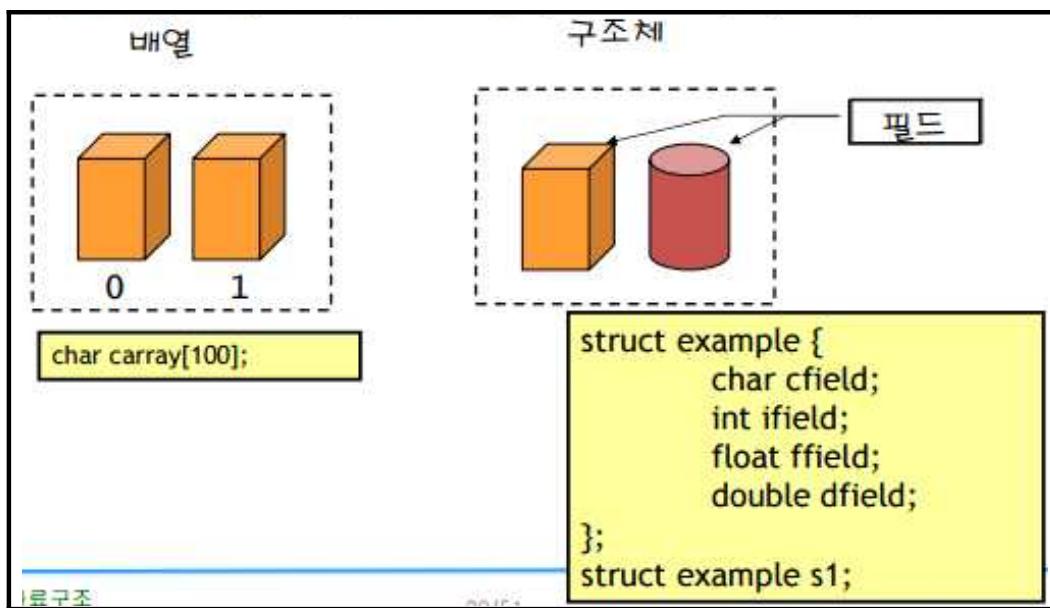
\* 0이 아닌 요소들만 저장하는 방법

- 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
```

#### ● typedef를 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

<ul style="list-style-type: none"> <li>● 포인터: 다른 변수의 주소를 가지고 있는 변수</li> </ul> <pre>char a='A'; char *p; p = &amp;a;</pre>	<ul style="list-style-type: none"> <li>● 포인터가 가리키는 내용의 변경: * 연산자 사용</li> </ul> <pre>*p= 'B';</pre>	<table border="1"> <thead> <tr> <th>주소</th> <th>값</th> <th>변수</th> <th>기능</th> </tr> </thead> <tbody> <tr> <td>26</td> <td>'B'</td> <td>a</td> <td>일반 변수</td> </tr> <tr> <td>↑(30)</td> <td>26</td> <td>p</td> <td>포인터</td> </tr> <tr> <td>↑</td> <td>30</td> <td>pp</td> <td>이중 포인터</td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th>형태</th> <th>값</th> <th>설명</th> </tr> </thead> <tbody> <tr> <td>p</td> <td>26</td> <td>a의 주소</td> </tr> <tr> <td>*p</td> <td>'B'</td> <td>a의 값</td> </tr> <tr> <td>pp</td> <td>30</td> <td>p의 주소</td> </tr> <tr> <td>*pp</td> <td>26</td> <td>p의 값</td> </tr> </tbody> </table>	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
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

<ul style="list-style-type: none"> <li>● <code>&amp;</code> 연산자: 변수의 주소를 추출</li> <li>● <code>*</code> 연산자: 포인터가 가리키는 곳의 내용을 추출</li> </ul> <pre>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.</pre>
---

<pre>int a; // 정수 변수 선언 int *p; // 정수 포인터 선언 int **pp; // 정수 포인터의 포인터 선언: 이중포인터 p = &amp;a; // 변수 a와 포인터 p를 연결 pp = &amp;p; // 포인터 p와 포인터의 포인터 pp를 연결</pre>
---

<pre>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</pre>
--

### \* 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]++;

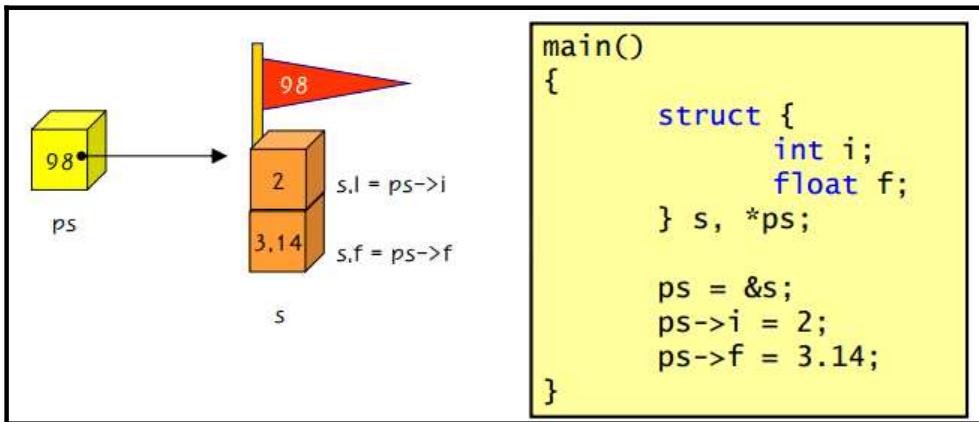
계산	p=5	p++	++p=5, (*p)++							
값	5	0	6	0	0	0	0	0	0	0

### \* 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)

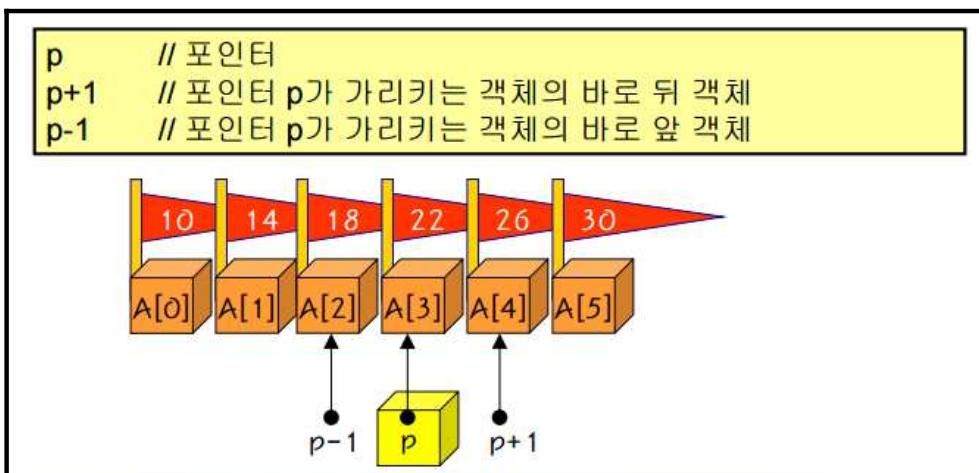
### \* 구조체의 포인터

- -> : 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 수는 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 새 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)

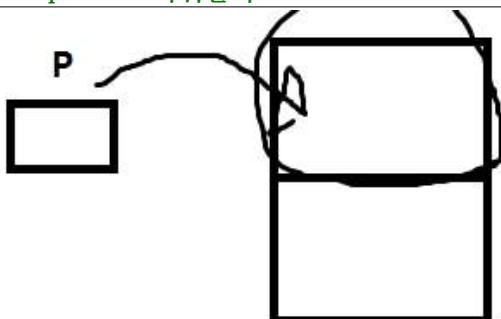
```
(char *)malloc(100); /* 100 바이트로 100개의 문자를 저장 */
(int *)malloc(sizeof(int));/* 정수 1개를 저장할 메모리 확보 */
(struct Book *)malloc(sizeof(struct Book));/* 하나의 구조체 생성 */
```

### 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'tallocatemory\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 쪍으면 크기가 다르기 때문에(더 크게 할  
당해줌) (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)

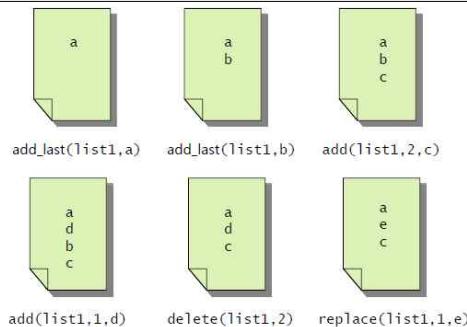
**2\*sizeof(struct Example)**  
(2개 선언시)

# 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

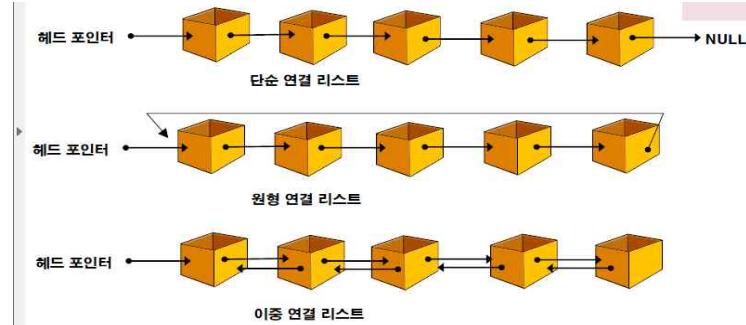
#### \* 배열을 이용하는 방법

- 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)
- typeef 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;
}

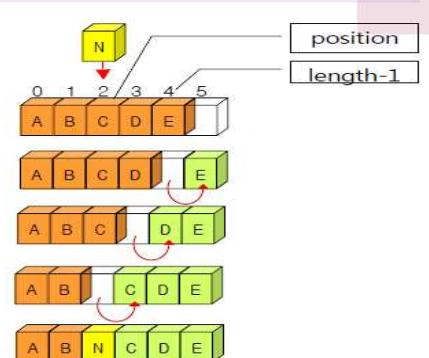
// 리스트가 가득 차 있으면 1을 반환
// 그렇지 않으면 1을 반환
int is_full(ArrayListType *L) {
    return L->length == MAX_LIST_SIZE;
}
```

### ArrayListType - 리스트 초기화

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

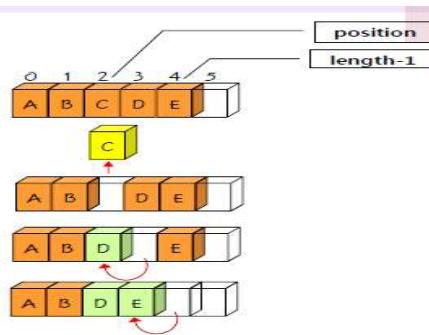
### ArrayListType - 삽입 연산

```
// 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;
}
```

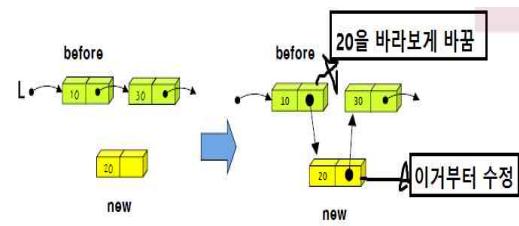


## \* 단순 연결 리스트

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

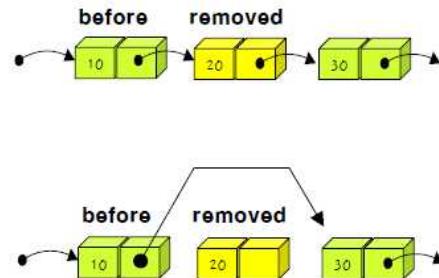
### 단순 연결 리스트 - 삽입

```
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);  
    }  
}
```



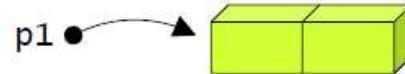
- \* 데이터 필드 : 구조체로 정의

- \* 링크 필드 : 포인터 사용

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

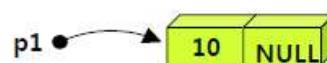
- \* 노드의 생성 : 동적 메모리 생성 라이브러리 malloc 함수 이용

```
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.h>

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;

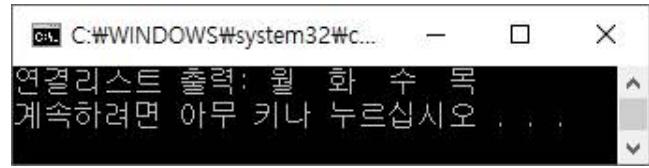
    // 메모리 할당
    tmp1 = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(tmp1->day, "월"); // C에서는 스트링이 없으므로 strcpy(스트링카피)를 사용
    pDays = tmp1;

    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.h>

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 : 삽입 노드
// new_node : 삽입될 노드
void insert_node(ListNode **phead, ListNode *before, ListNode *new_node)
{
    if (*phead == NULL){ // 빈리스트인 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;

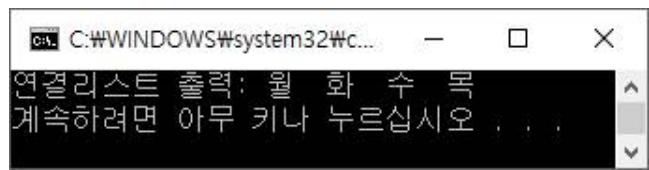
    // 메모리 할당
    tmp1 = (ListNode*)malloc(sizeof(ListNode)); // 첫 번째 방법
    strcpy_s(tmp1->day, "월"); // C에서의 스트링이 있으므로 strcpy(스트링카피)를 사용
    insert_node(&pDays, NULL, tmp1); // 두 번째 방법 (insert 함수 사용)

    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, tmp3, 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.h>

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 필요없어짐
    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);

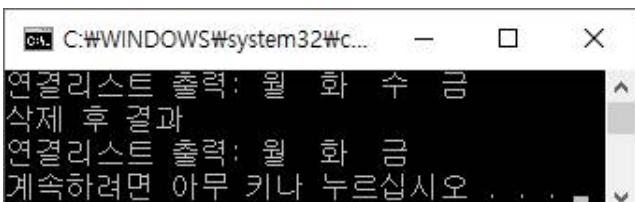
    /*화*/
    new_node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(new_node->day, "화");
    new_node->link = NULL;
    before = search(pDays, "월");
    insert_node(before, new_node);

    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

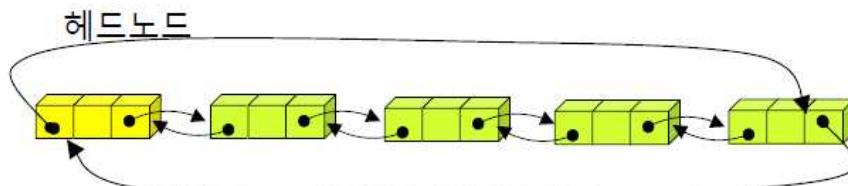
\* 이중연결 리스트 : 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트

- 링크가 양방향이므로 양방향으로 검색이 가능

- 단점은 공간을 많이 차지하고 코드가 복잡

\* 실제 사용되는 이중 연결 리스트의 형태

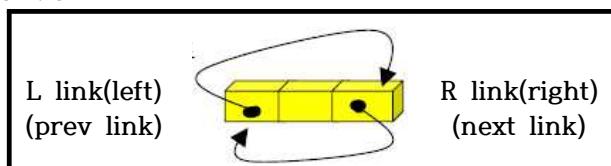
- 헤드노드 + 이중연결 리스트 + 원형연결 리스트



\* 헤드노드 : 데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드

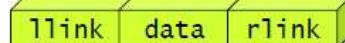
- 헤드 포인터와의 구별이 필요

- 공백상태에서는 헤드 노드만 존재



\* 이중연결리스트에서의 노드 구조

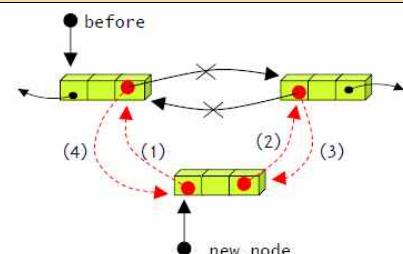
```
typedef int element;
typedef struct DlistNode {
    element data;
    struct DlistNode *llink;
    struct DlistNode *rlink;
} DlistNode;
```



\* 삽입연산

#### 이중 연결 리스트 - 삽입

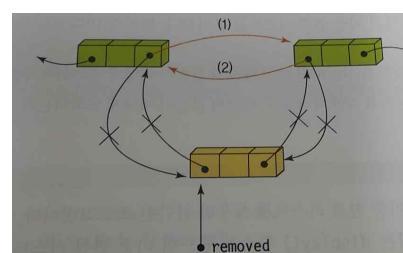
```
// 노드 new_node를 노드 before의 오른쪽에 삽입한다.
void dinsert_node(DListNode *before, DListNode *new_node){
    new_node->llink = before;
    new_node->rlink = before->rlink; // (1)
    before->rlink->llink = new_node; // (2)
    before->rlink = new->link; // (3)
}
```



#### 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;

// p는 headnode (pDays)
void printList( 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;

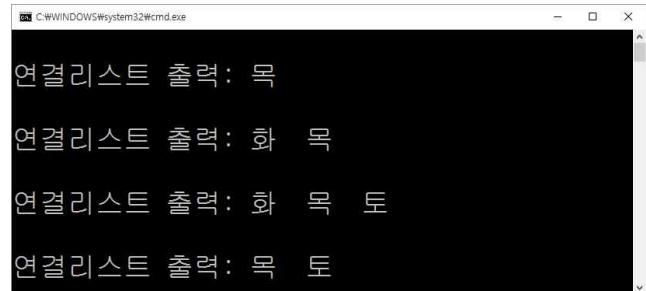
    pDays = (DlistNode*)malloc(sizeof(DlistNode));
    pDays->next = pDays;
    pDays->prev = pDays;

    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 );
}
```



```
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.h>

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
}
```

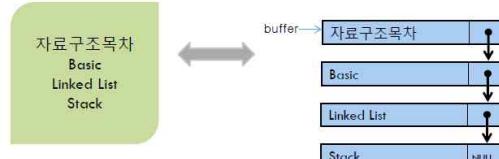
```
void main()
{
    DlistNode *pDays, *new_node, *before, *remove;
    char select;
    char input[200];
    char where[200];

    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;
            new_node->prev = pDays;
            dinsert_node(pDays, new_node);
        }
    }
}
```

### 도전 문제-라인에디터

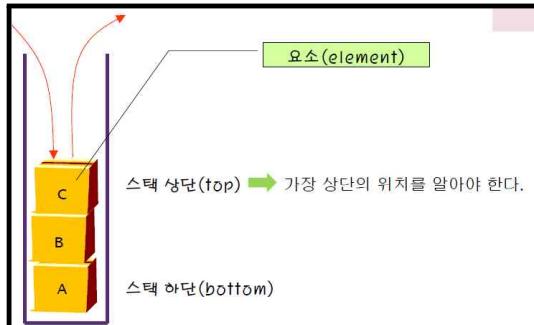
- 라인단위로 텍스트를 입력하거나 삭제할 수 있는 단순한 에디터
- 라인 단위로 이루어지므로 커서를 사용하지 않는다.
- 입력되는 라인의 개수가 정해지지 않았으므로 연결리스트 사용
- 삽입
  - 라인의 번호와 라인 내용을 입력 받아 삽입
- 삭제
  - 삭제하고자 하는 라인 번호를 입력 받아 삭제



# 스택

## ○ 스택 정의

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

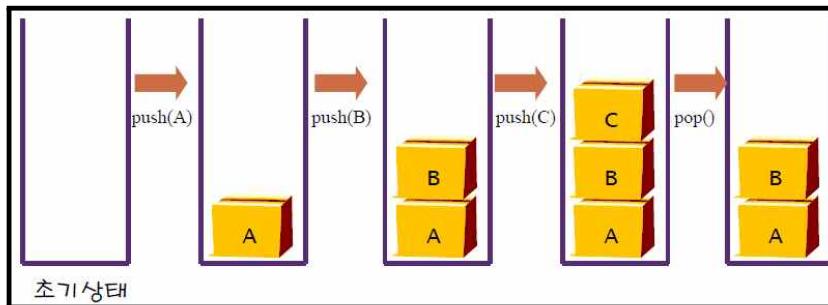


## \* 스택 추상데이터타입 (ADT)

- 객체: n개의 element형의 요소들의 선형 리스트
- 연산:
  - create() ::= 스택을 생성한다.
  - is\_empty(s) ::= 스택이 비어있는지를 검사한다.
  - is\_full(s) ::= 스택이 가득 찬는가를 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



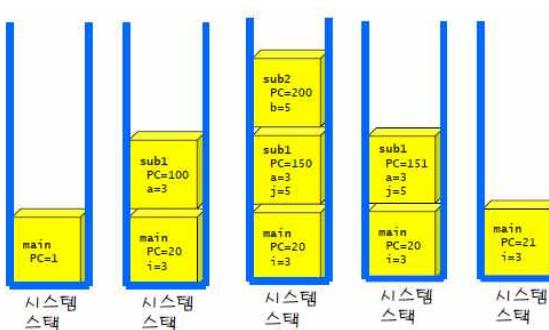
- \* is\_empty(s) : 스택이 공백상태인지 검사
- \* is\_full(s) : 스택이 포화상태인지 검사
- \* create() : 스택을 생성
- \* peek(s) : 요소를 스택에서 삭제하지 않고 보기만 하는 연산  
// (참고) pop 연산은 요소를 스택에서 완전히 삭제하면서 가져온다.

## ※ 스택의 용도

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

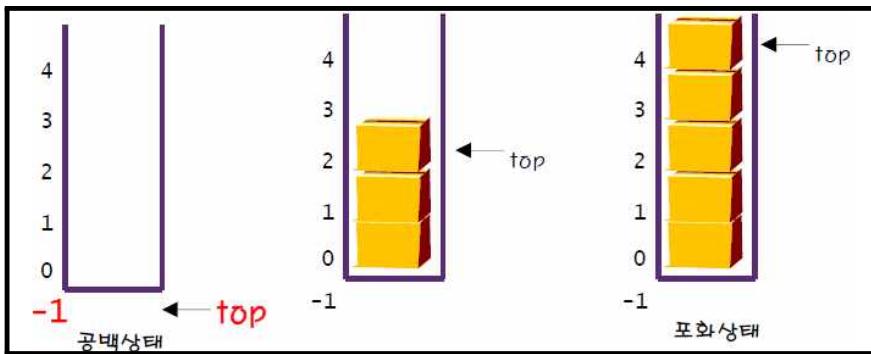
```

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) {
    ...
}
  
```

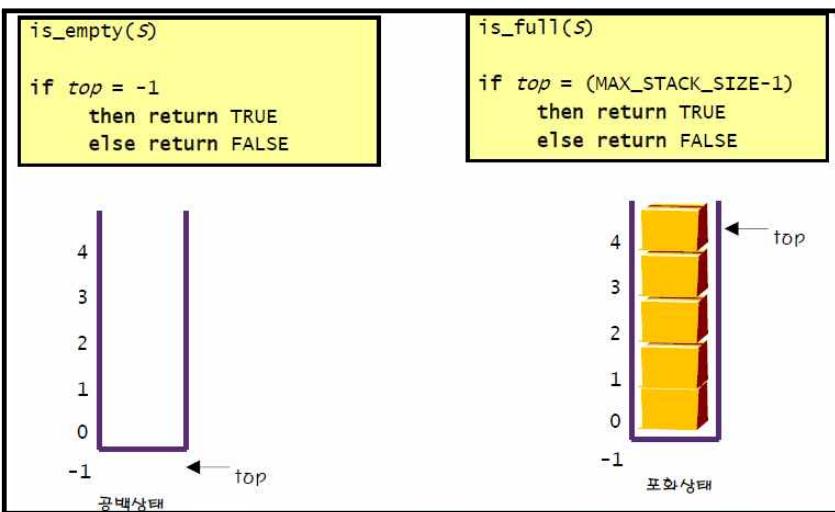


## ○ 배열을 이용한 스택의 구현

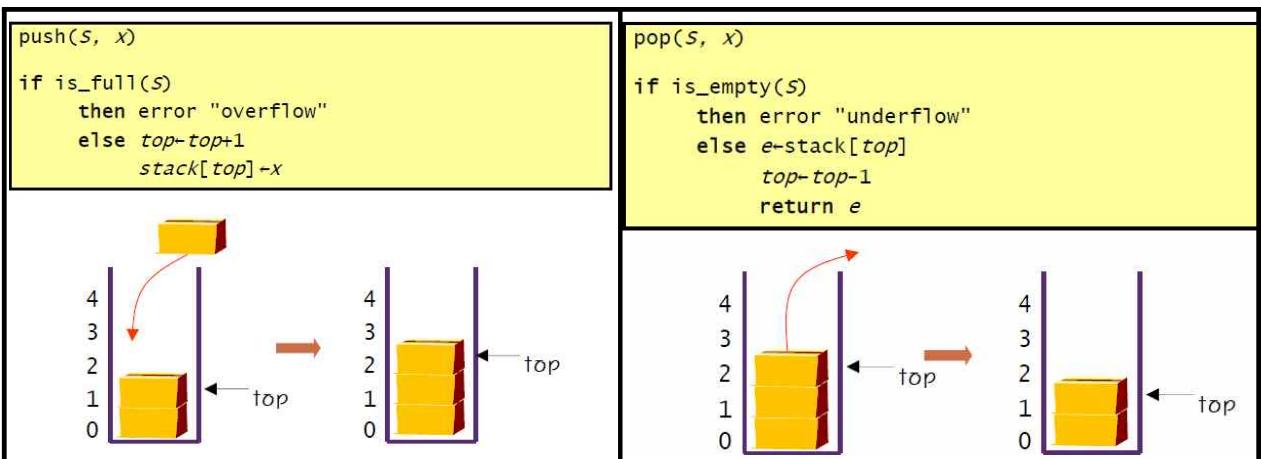
- \* 1차원 배열 stack[]
- \* 스택에서 가장 최근에 입력되었던 자료를 가리키는 top변수
- \* 가장 먼저 들어온 요소는 stack[0]에, 가장 최근에 들어온 요소는 stack[top]에 저장
- \* 스택이 공백상태이면 top은 -1



### ※ is\_empty(), is\_full()

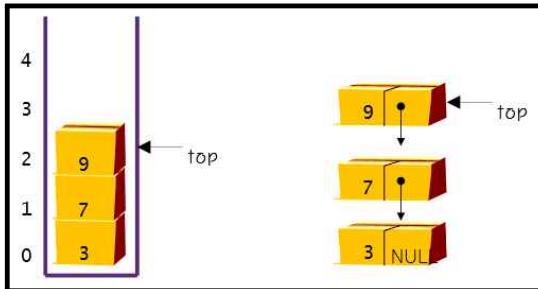


### ※ push(), pop()

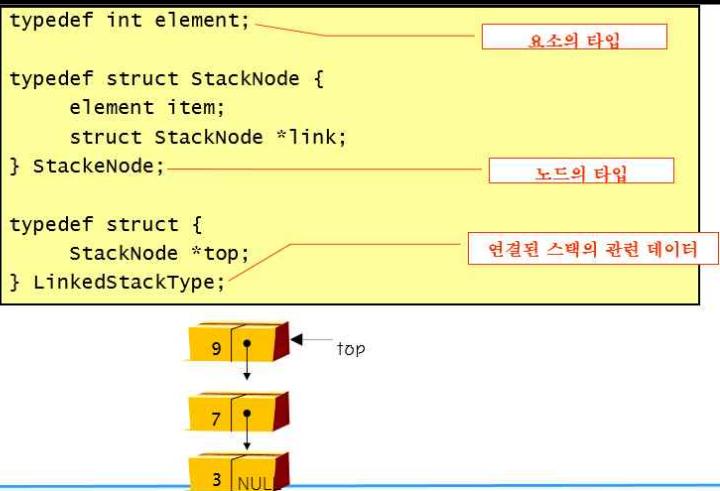


## ○ 연결된 스택 : 연결리스트를 이용하여 구현한 스택

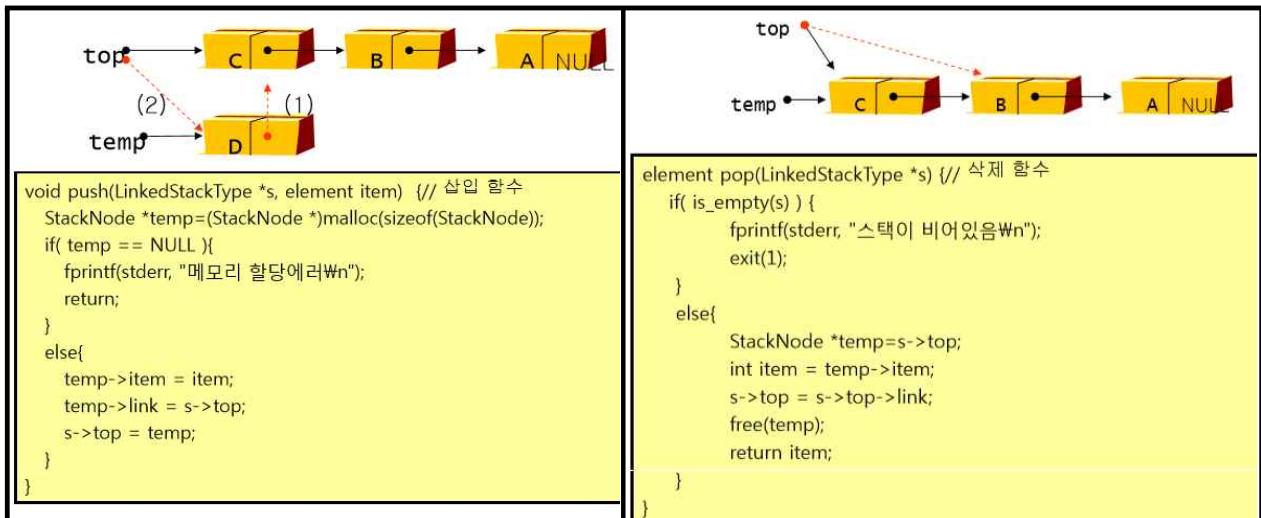
- \* 장점 : 크기가 제한되지 않음
- \* 단점 : 구현이 복잡하고 삽입, 삭제 시간이 오래 걸림



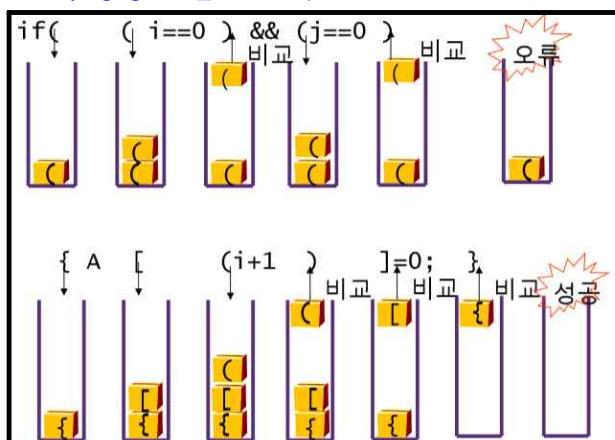
### \* 연결된 스택 정의



### \* 연결된 스택 push(), pop()



### \* 스택 응용 : 괄호 검사



## \* 팔호 검사 알고리즘

### 스택 예제 - 팔호 검사

```

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

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

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

```

```

// 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));
}

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;
        }
    }
    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)  $2+3*4 \rightarrow 234*+ \rightarrow 14$
- \* 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 큰	스택						
	[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

```
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
스택에서 제거
            op2 = pop(&s);
            op1 = pop(&s);
            switch (ch){ //연산을
수행하고 스택에 저장
                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;
}
```

#### 중위표기식 -> 후위 표기식

```
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 있는 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 까지 출력
                    while (top_op != '('{
                        printf("%c",
top_op);
                        top_op =
pop(&s);
                    }
                    break;
        default: // 피연산자
                    printf("%c", ch);
                    break;
        }
    }
    // 스택에 저장된 연산자들 출력
    while (!is_empty(&s))
        printf("%c", pop(&s));
}
```

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

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

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>\n", tmp->day);
        tmp = tmp->next;
    }
    printf("\n-----\n\n");
}

// before : 선택 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: 삭제된 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;

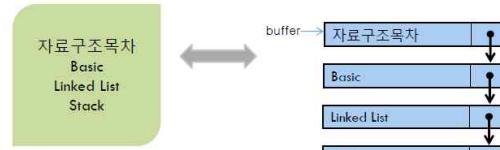
    while (true){
        printf("a : Add \n");
        printf("r : Remove \n");
        printf("e : Exit \n");
        printf("Input Select >> ");
        scanf("%c", &select);
        fflush(stdin);

        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;
            new_node->prev = NULL;

            if (pDays->next == pDays){
                dinsert_node(pDays, new_node);
            } else {
                printf("Input where? (Input String) >> ");
                fgets(where, sizeof(where), stdin);
                where[strlen(where) - 1] = '\0';
                //scanf("%s", &where);
                before = search(pDays, where);
                dinsert_node(before, new_node);
            }
            printList(pDays);
            break;
        case 'r':
            printf("Remove what? (Input String) >> ");
            fgets(input, sizeof(input), stdin);
            input[strlen(input) - 1] = '\0';
            //scanf("%s", &input);
            break;
        }
    }
}
```

## 도전 문제-라인에디터

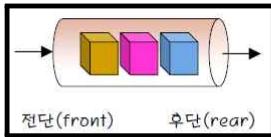
- 라인단위로 텍스트를 입력하거나 삭제할 수 있는 단순한 에디터
- 라인 단위로 이루어지므로 커서를 사용하지 않는다.
- 입력되는 라인의 개수가 정해지지 않았으므로 연결리스트 사용
- 삽입
  - 라인의 번호와 라인 내용을 입력 받아 삽입
- 삭제
  - 삭제하고자 하는 라인 번호를 입력 받아 삭제



# 큐

## ○ 큐 정의

- \* 큐 : 먼저 들어온 데이터가 먼저 나가는 자료구조
- \* 선입선출 : 가장 최근에 들어온 데이터가 가장 먼저 나감
- \* 큐의 구조 : 삽입은 큐의 후단(rear), 삭제는 전단(front)에서 이루어짐



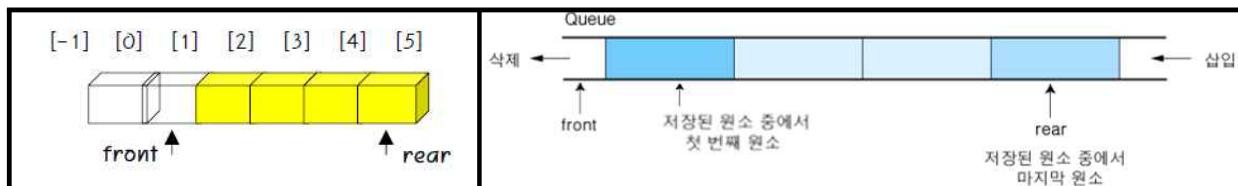
## \* 큐 추상데이터타입 (ADT)

· 객체: n개의 element형으로 구성된 요소들의 순서 있는 모임  
 · 연산:

- **create()** := 큐를 생성한다.
- **init(q)** := 큐를 초기화한다.
- **is\_empty(q)** := 큐가 비어있는지를 검사한다.
- **is\_full(q)** := 큐가 가득 찬는가를 검사한다.
- **enqueue(q, e)** := 큐의 뒤에 요소를 추가한다.
- **dequeue(q)** := 큐의 앞에 있는 요소를 반환한 다음 삭제한다.
- **peek(q)** := 큐에서 삭제하지 않고 앞에 있는 요소를 반환한다.

## ○ 선형 큐 : 배열을 선형으로 사용하여 큐를 구현

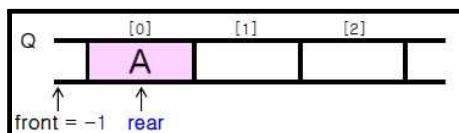
- \* 큐의 전단과 후단을 관리하기 위한 2개의 변수 필요
- \* **front** : 첫 번째 요소 하나 앞의 인덱스
- \* **rear** : 마지막 요소의 인덱스



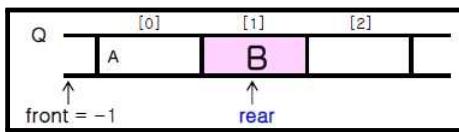
### \* 공백 큐 생성 : createQueue();



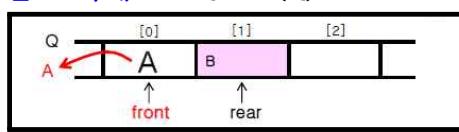
### \* 원소 A 삽입 : enqueue(Q, A);



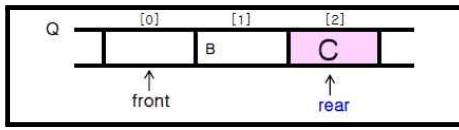
### \* 원소 B 삽입 : enqueue(Q, B);



### \* 원소 삭제 : dequeue(Q);

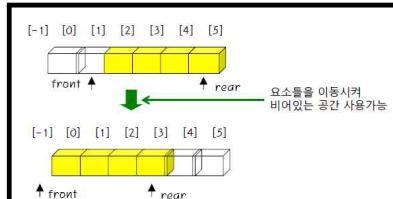


### \* 원소 C 삽입 : enqueue(Q, C);



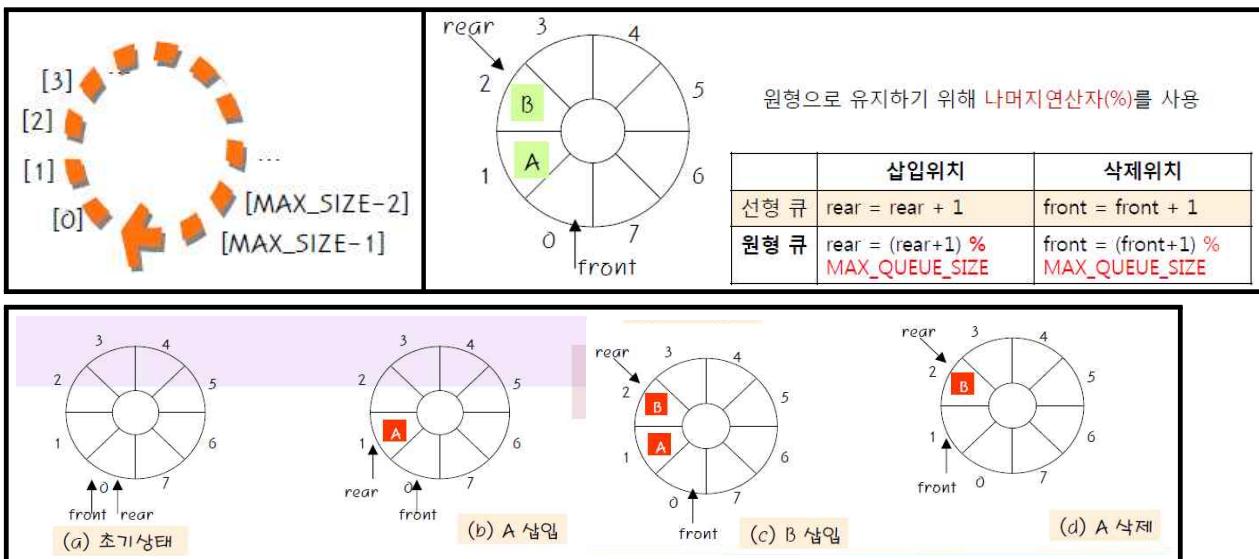
### \* 선형 큐의 문제점

- 삽입을 계속하기 위해서는 요소들을 이동시켜야 함
- 문제점이 많아 사용되지 않음



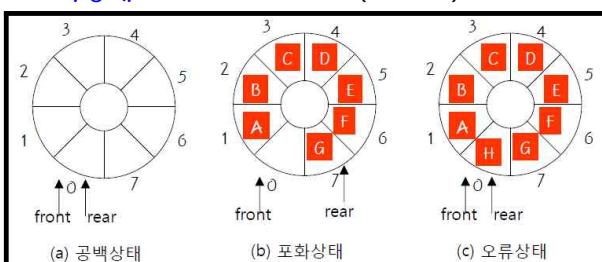
## ○ 원형 큐 : 배열을 원형으로 사용하여 큐를 구현

- \* 큐의 전단과 후단을 관리하기 위한 2개의 변수 필요
- \* **front** : 첫 번째 요소 하나 앞의 인덱스
- \* **rear** : 마지막 요소의 인덱스



## ※ 공백상태, 포화상태

- \* **공백상태** :  $\text{front} == \text{rear}$
- \* **포화상태** :  $\text{front} \& \text{M} == (\text{rear}+1) \% \text{M}$



## ※ 큐를 위한 구조체 선언

```
typedef int element;

typedef struct {
    element queue[MAX_QUEUE_SIZE];
    int front, rear;
} QueueType;

QueueType queue;
```

※ 공백 상태 검사 : `is_empty(QueueType *q)`

※ 포화 상태 검사 : `is_full(QueueType *q)`

```
// 공백 상태 검출 함수
int is_empty(QueueType *q)
{
    return (q->front == q->rear);
}

// 포화 상태 검출 함수
int is_full(QueueType *q)
{
    return ((q->rear+1)%MAX_QUEUE_SIZE == q->front);
}
```

- \* 삽입 함수 : enqueue(QueueType \*q, element item)
- \* 삭제 함수 : element dequeue(QueueType \*q)

```

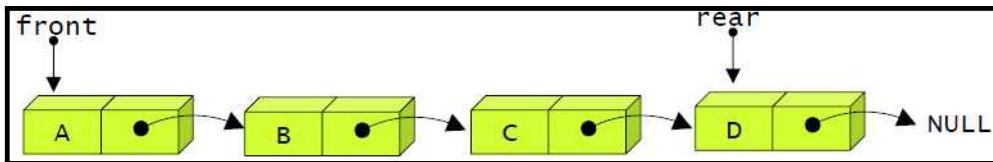
void enqueue(QueueType *q, element item) // 삽입 함수
{
    if( is_full(q) )
        error("큐가 포화상태입니다");
    q->rear = (q->rear+1) % MAX_QUEUE_SIZE;
    q->queue[q->rear] = item;
}

element dequeue(QueueType *q) // 삭제 함수
{
    if( is_empty(q) )
        error("큐가 공백상태입니다");
    q->front = (q->front+1) % MAX_QUEUE_SIZE;
    return q->queue[q->front];
}

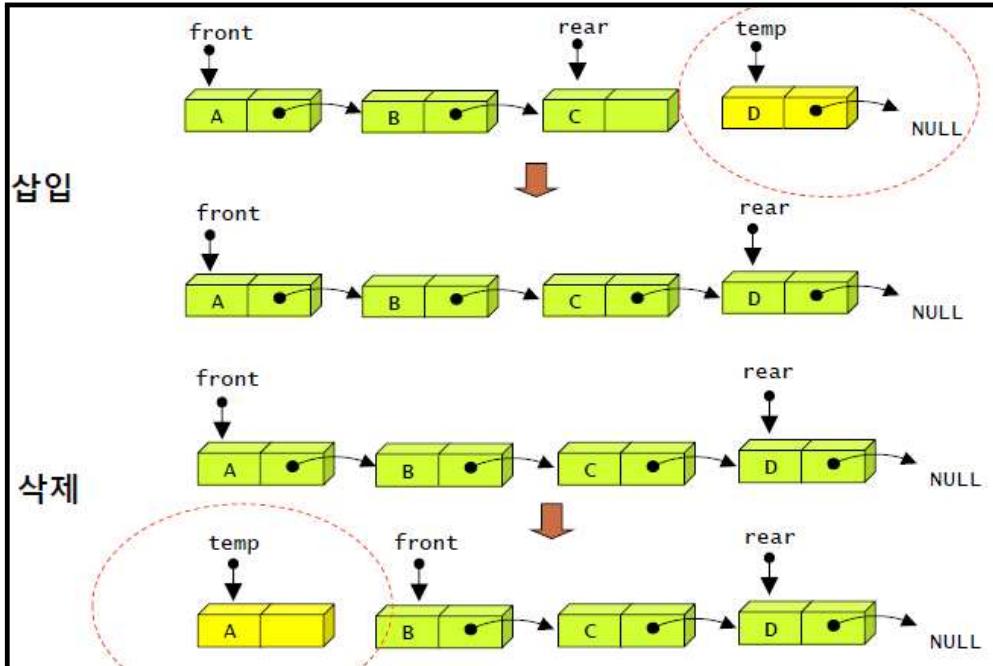
```

## ○ 연결리스트 큐

- \* front 포인터는 삭제와 관련되며 rear 포인터는 삽입
- \* front는 연결 리스트의 맨 앞에 있는 요소를 가리키며, rear 포인터는 맨 뒤에 있는 요소를 가리킴
- \* 큐에 요소가 없는 경우에는 front와 rear는 NULL

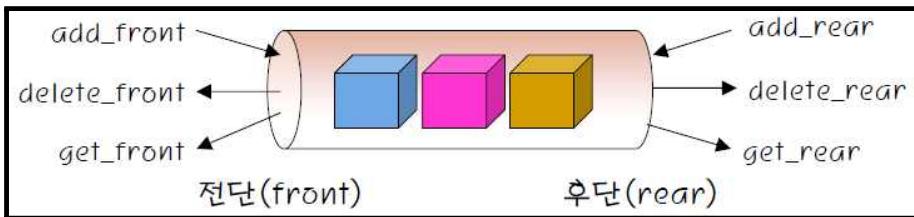


## ※ 연결된 큐에서의 삽입과 삭제



## 덱 (deque)

○ 덱(double-ended queue) : 큐의 전단(front)과 후단(rear)에서 모두 삽입과 삭제가 가능한 큐



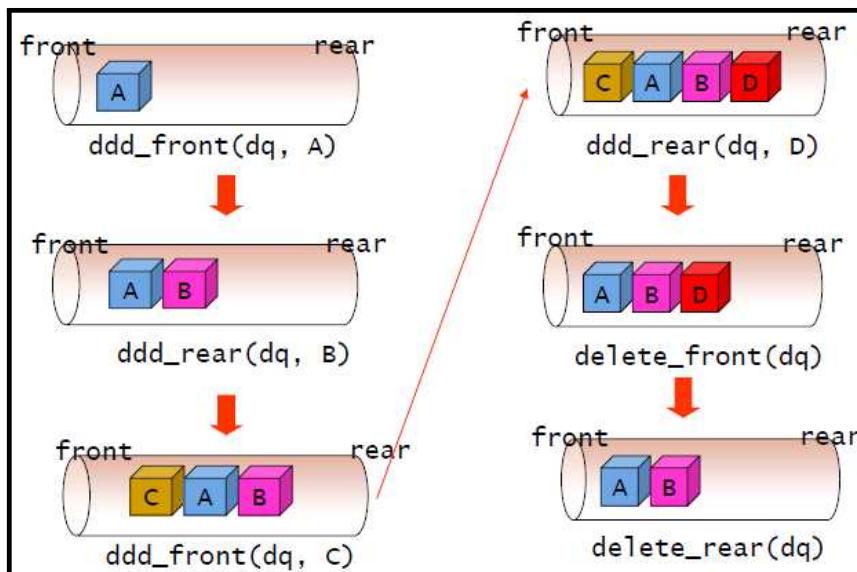
### \* 덱 추상데이터타입 (ADT)

• 객체: n개의 element형으로 구성된 요소들의 순서있는 모임

• 연산:

- `create()` ::= 덱을 생성한다.
- `init(dq)` ::= 덱을 초기화한다.
- `is_empty(dq)` ::= 덱이 공백상태인지를 검사한다.
- `is_full(dq)` ::= 덱이 포화상태인지를 검사한다.
- `add_front(dq, e)` ::= 덱의 앞에 요소를 추가한다.
- `add_rear(dq, e)` ::= 덱의 뒤에 요소를 추가한다.
- `delete_front(dq)` ::= 덱의 앞에 있는 요소를 반환한 다음 삭제한다.
- `delete_rear(dq)` ::= 덱의 뒤에 있는 요소를 반환한 다음 삭제한다.
- `get_front(q)` ::= 덱의 앞에서 삭제하지 않고 앞에 있는 요소를 반환한다.
- `get_rear(q)` ::= 덱의 뒤에서 삭제하지 않고 뒤에 있는 요소를 반환한다.

### \* 덱의 연산



### \* 덱의 구현

\* 양쪽에서 삽입, 삭제가 가능하여 하므로 일반적으로 이중 연결 리스트 사용

```

typedef int element;           // 요소의 타입

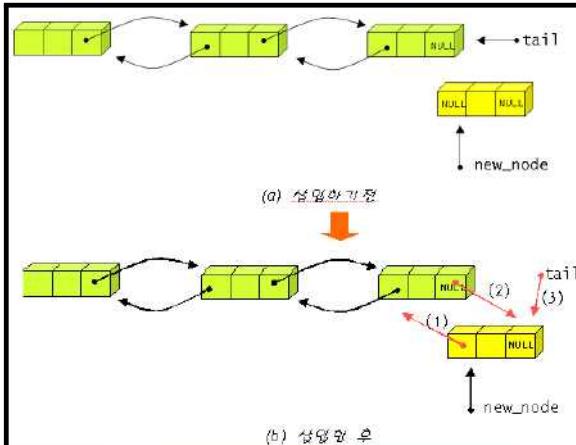
typedef struct DlistNode {      // 노드의 타입
    element data;
    struct DlistNode *llink;
    struct DlistNode *rlink;
} DlistNode;

typedef struct DequeType {      // 덱의 타입
    DlistNode *head;
    DlistNode *tail;
} DequeType;

```

## \* 맵의 삽입 연산

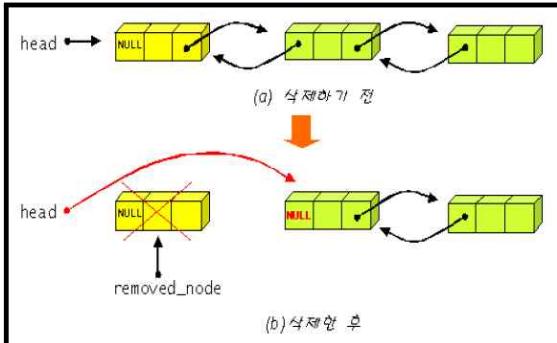
\* 헤더 포인터 대신 head와 tail 포인터 사용



```
void add_rear(DequeType *dq, element item)
{
    DlistNode *new_node = create_node(dq->tail, item, NULL);
    if( is_empty(dq))      dq->head = new_node;
    else      dq->tail->rlink = new_node;
    dq->tail = new_node;
}

void add_front(DequeType *dq, element item)
{
    DlistNode *new_node = create_node(NULL, item, dq->head);
    if( is_empty(dq))      dq->tail = new_node;
    else      dq->head->llink = new_node;
    dq->head = new_node;
}
```

## \* 맵의 삭제 연산



```
// 전단에서의 삭제
element delete_front(DequeType *dq)
{
    element item;
    DlistNode *removed_node;

    if( is_empty(dq))  error("공백 큐에서 삭제");
    else {
        removed_node = dq->head; // 삭제할 노드
        item = removed_node->data; // 데이터 추출
        dq->head = dq->head->rlink; // 헤드 포인터 변경
        free(removed_node); // 메모리 공간 반납
        if (dq->head == NULL) // 공백상태이면
            dq->tail = NULL;
        else // 공백상태가 아니면
            dq->head->llink=NULL;
    }
    return item;
}
```

## \* 큐 표 시험 예상 문제

번호	연산	0	1	2	3	4	5	front	rear
1	A삽입		A					0	1
2	B삽입		A	B				0	2
3	C삽입		A	B	C			0	3
4	A삭제			B	C			1	3
5	B삭제				C			2	3
6	D삽입				C	D		2	4

\* buster부스터 헤더파일 / 메르스 난수를 사용 // 랜덤보다 나음

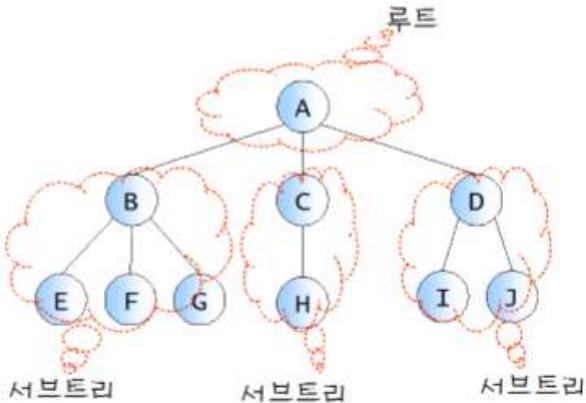
# 트리 (Tree)

## ○ 트리 : 계층적인 구조를 나타내는 자료구조

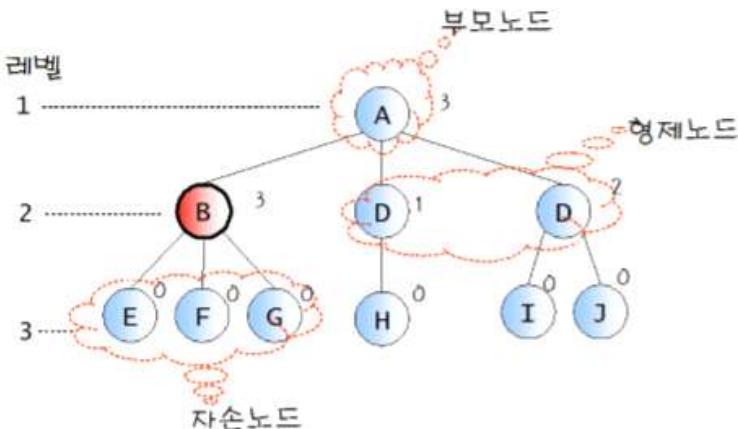
- \* 리스트, 스택, 큐 등은 선형 구조
- \* 트리는 부모 - 자식 관계의 노드들로 이루어진다.
- \* 응용 분야
  - 계층적인 조직 표현
  - 컴퓨터 디스크의 디렉터리 구조
  - 인공지능에서의 결정트리 (decision tree)

### \* 트리의 용어

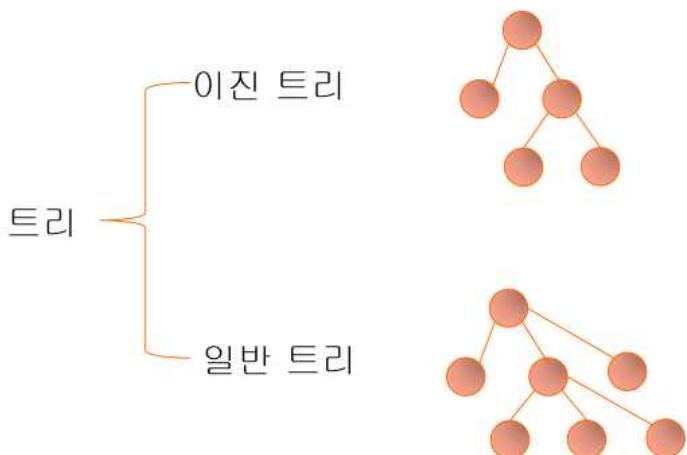
- \* 노드 (node) : 트리의 구성요소
- \* 루트 (root) : 부모가 없는 노드(A)
- \* 서브트리 (subtree) : 하나의 노드와 그 노드들의 자손들로 이루어진 트리
- \* 단말노드 (terminal node) : 자식이 없는 노드 (E, F, G, H, I, J)
- \* 비단말노드 : 적어도 하나의 자식을 가지는 노드 (A, B, C, D)



- \* 자식, 부모, 형제, 조상, 자손 노드 : 인간과 동일
- \* 레벨(level) : 트리의 각층의 번호
- \* 높이(height) : 트리의 최대 레벨(3)
- \* 차수(degree) : 노드가 가지고 있는 자식 노드의 개수



### \* 트리의 종류

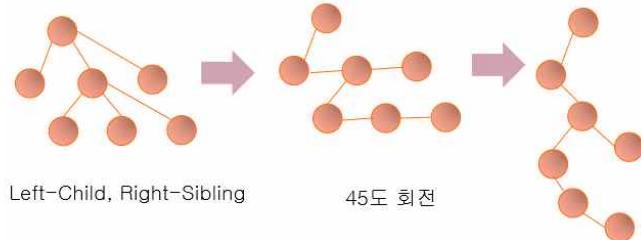


## \* 일단 트리를 이진 트리로 변환

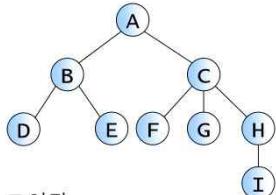
\* 일반 트리를 이진 트리로 만드는 방법

- 왼쪽 노드는 자식 노드로, 오른쪽 노드들은 형제 노드로 배치(Left-Child, Right-Sibling)

\* 모든 트리를 이진 트리 형태로 재구성 가능



시험 : 예제



- A는 루트 노드이다.
- B는 D와 E의 부모노드이다.
- C는 B의 형제 노드이다.
- D와 E는 B의 자식노드이다.
- B의 차수는 2이다.
- 위의 트리의 높이는 4이다.

## ○ 이진 트리 (binary tree)

### \* 이진 트리 : 모든 노드가 2개의 서브 트리를 가지고 있는 트리

- 서브 트리는 공집합일 수 있다.

- \* 이진 트리의 노드에는 최대 2개까지의 자식 노드가 존재
- \* 모든 노드의 차수가 2 이하가 된다  $\rightarrow$  구현하기가 편리
- \* 이진 트리에는 서브 트리간의 순서가 존재

### \* 이진 트리 검증

\* 이진 트리는 공집합

\* 루트와 왼쪽 서브 트리, 오른쪽 서브 트리로 구성된 노드들의 유한 집합으로 정의된다.

\* 이진 트리의 서브 트리들은 모두 이진 트리여야 한다.

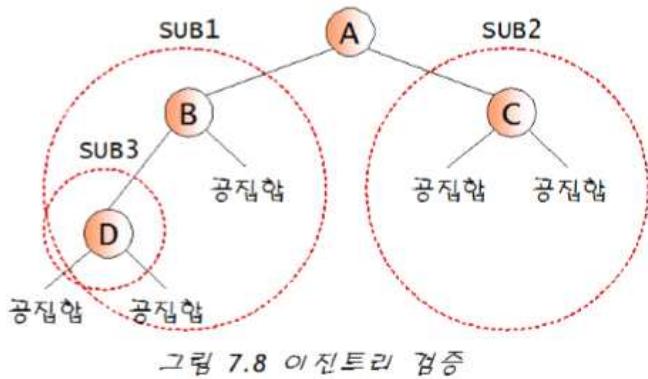


그림 7.8 이진트리 검증

### \* 이진트리의 성질

\* 노드의 개수가 n개이면 간선의 개수는 n-1

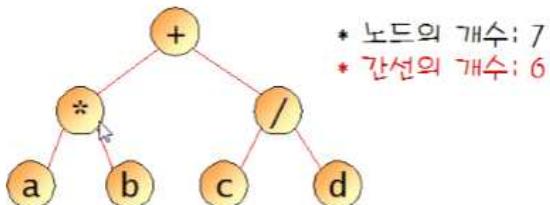
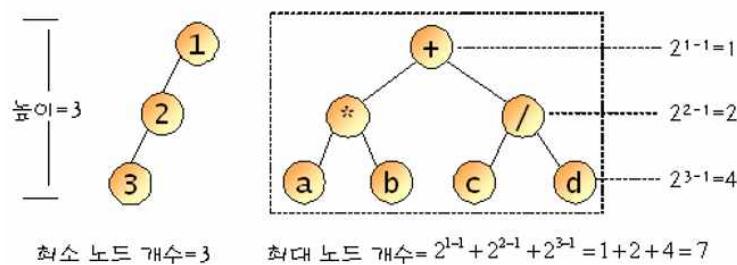


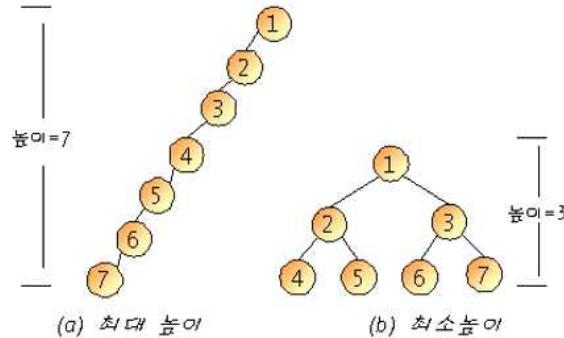
그림 7.10 노드의 개수와 간선의 개수와의 관계

- \* 높이가  $h$ 인 이진트리의 경우, 최소  $h$ 개의 노드를 가지며, 최대  $2^h - 1$ 개의 노드를 가진다.



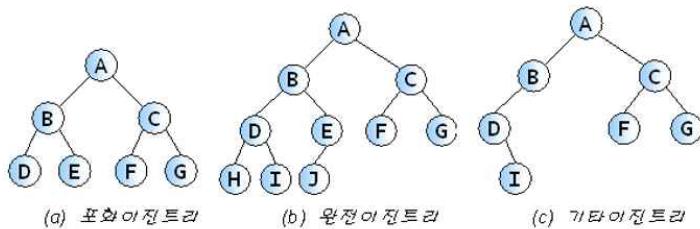
- \*  $n$ 개의 노드를 가지는 이진트리의 높이

- 최대 :  $n$
- 최소 :  $\lceil \log_2(n+1) \rceil \Rightarrow (n+1)$ 을 2의 거듭제곱으로 만들  $\Rightarrow$  나온 값을 올림



### \* 이진트리의 분류

- \* 포화 이진 트리 (full binary tree)
- \* 완전 이진 트리 (complete binary tree) : 자식 노드가 왼쪽부터 붙음
- \* 기타 이진 트리



### \* 완전 이진 트리

- \* 완전 이진 트리 : 레벨 1부터  $k-1$ 까지는 노드가 모두 채워져 있고, 마지막 레벨  $k$ 에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진트리

- \* 포화 이진 트리와 노드 번호가 일치

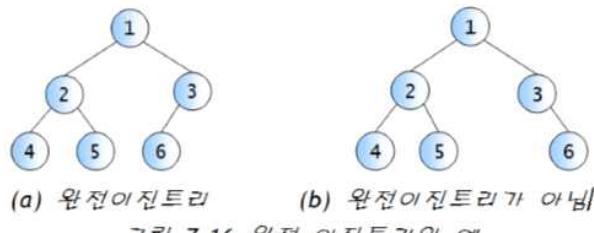
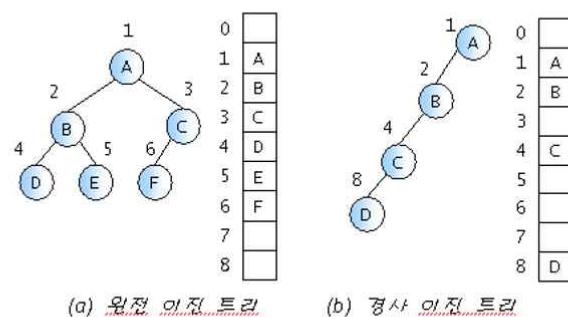


그림 7.16 완전 이진트리의 예

### \* 배열 표현법

- \* 모든 이진 트리를 포화 이진 트리라 가정하고, 각 노드에 번호를 붙여서 그 번호를 배열의 인덱스로 삼아 노드의 데이터를 배열에 저장하는 방법

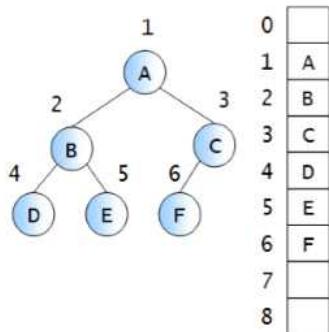


## 예제

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	A		B			C	D					E	F														G

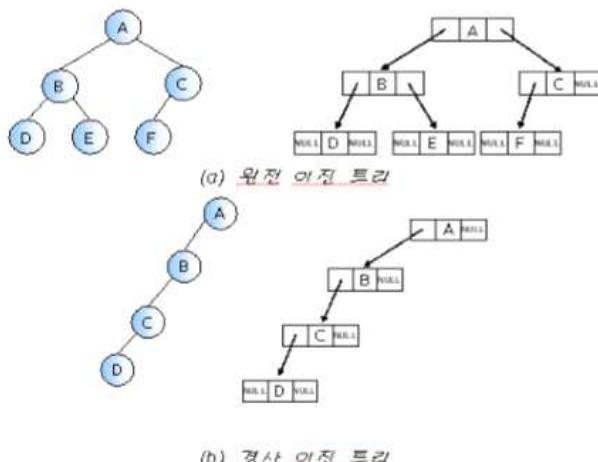
## \* 부모와 자식 인덱스 관계

- \* 노드 I의 부모 노드 인덱스 :  $i / 2$
- \* 노드 I의 왼쪽 자식 노드 인덱스 :  $2 * i$
- \* 노드 I의 오른쪽 자식 노드 인덱스 :  $2 * i + 1$



## \* 링크 표현법

- \* 링크 표현법 : 포인터를 이용하여 부모 노드가 자식 노드를 가리키게 하는 방법



## \* 링크의 구현 :: 노드

```
typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right;
} TreeNode;
```

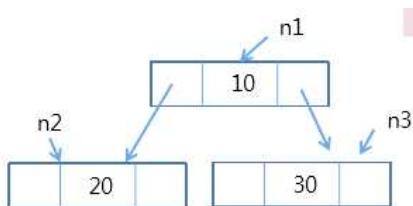
## \* 링크 표현법 프로그램

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

typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right;
} TreeNode;

void main()
{
    TreeNode *n1, *n2, *n3;

    n1= (TreeNode *)malloc(sizeof(TreeNode));
    n2= (TreeNode *)malloc(sizeof(TreeNode));
    n3= (TreeNode *)malloc(sizeof(TreeNode));
```



```
n1->data = 10;
n1->left = n2;
n1->right = n3;

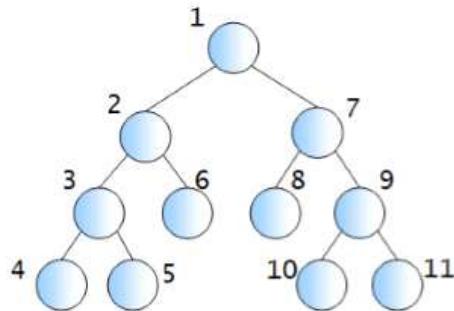
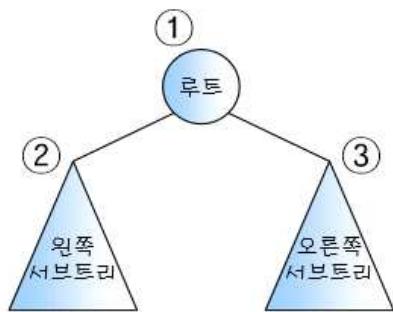
n2->data = 20;
n2->left= n2->right = NULL;

n3->data = 30;
n3->left = n3->right = NULL;
```

## ○ 이진 트리의 순회

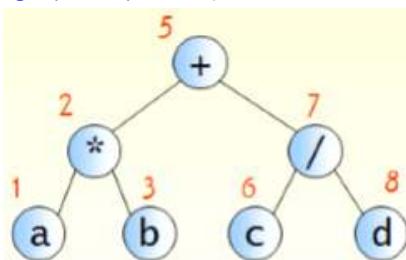
- \* 순회 (traversal) : 트리의 노드들을 체계적으로 방문하는 것
- \* 3가지의 기본적인 순회방법
  - 전위순회 : 자손 노드보다 루트 노드를 먼저 방문
  - 중위순회 : 왼쪽 자손, 루트, 오른쪽 자손 순으로 방문
  - 후위순회 : 루트 노드보다 자손을 먼저 방문

\* 전위 순회 : 루트 노드  $\rightarrow$  왼쪽 서브트리  $\rightarrow$  오른쪽 서브트리 순으로 방문



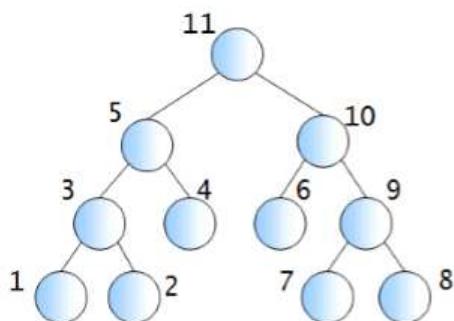
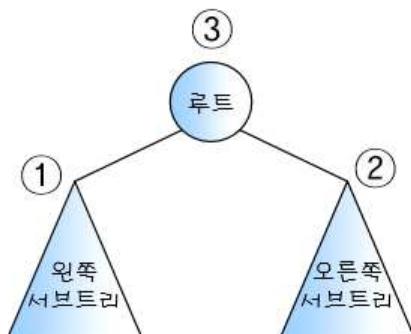
```
preorder(x)
if x≠NULL
    then      print DATA(x);
              preorder(LEFT(x));
              preorder(RIGHT(x));
```

\* 중위 순회 : 왼쪽 서브트리  $\rightarrow$  루트 노드  $\rightarrow$  오른쪽 서브트리 순으로 방문



```
inorder(x)
if x≠NULL
    then      inorder(LEFT(x));
              print DATA(x);
              inorder(RIGHT(x));
```

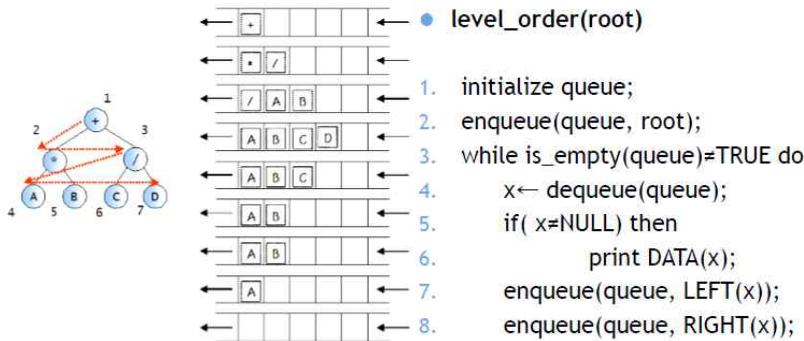
\* 후위 순회 : 왼쪽 서브트리  $\rightarrow$  오른쪽 서브트리  $\rightarrow$  루트 노드 순으로 방문



```
postorder(x)
if x≠NULL
    then      postorder(LEFT(x));
              postorder(RIGHT(x));
              print DATA(x);
```

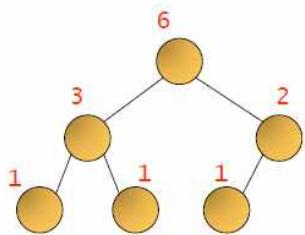
## \* 레벨 순회 : 각 노드를 레벨 순으로 검사

- \* 다른 순회들과 달리 큐를 사용하여 순회



## \* 이진 트리 연산 : 노드 개수

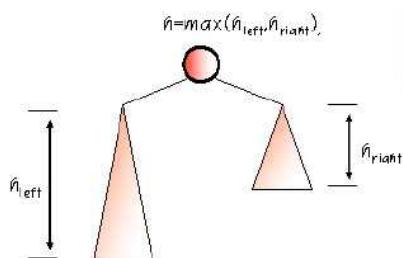
- \* 탐색 트리 안의 노드의 개수를 계산
- \* 각각의 서브트리에 대한 순환 호출한 다음, 반환되는 값에 1을 더하여 반환



```
int get_node_count(TreeNode *node)
{
    int count=0;
    if( node != NULL )
        count = 1 + get_node_count(node->left)+get_node_count(node->right);
    return count;
}
```

## \* 이진 트리 연산 : 높이

- \* 서브트리에 대하여 순환 호출하고 서브 트리들의 반환 값 중에서 최대 값을 구하여 반환



```
int get_height(TreeNode *node)
{
    int height=0;
    if( node != NULL )
        height = 1 + max(get_height(node->left),
                          get_height(node->right));
    return height;
}
```

## \* 스레드 이진 트리 구현

- \* 단말 노드와 비단말 노드의 구별을 위해 is\_thread 필드 필요

```
typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right;
    int is_thread; //만약 오른쪽 링크가 스레드이면 TRUE
} TreeNode;
```

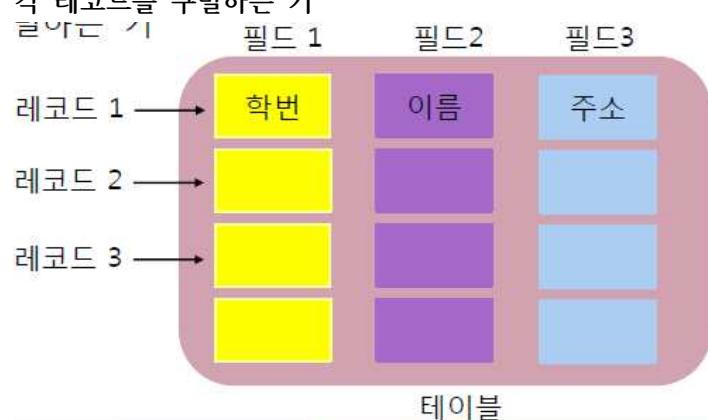
- \* 스레드 버전 중위 함수 작성

```
void thread_inorder(TreeNode *t){
    TreeNode *q;
    q=t;
    while (q->left) q = q->left; //가장 왼쪽 노드로 간다.
    do{
        printf("%c ", q->data); //데이터 출력
        q = find_successor(q); //후속자 함수 호출
    } while(q); // NULL이 아니면
}
```

```
TreeNode *find_successor(TreeNode *p) {
    TreeNode *q = p->right; // q는 p의 오른쪽 포인터
    // 만약 오른쪽 포인터가 NULL이거나 스레드이면 오른쪽 포인터를 반환
    if( q==NULL || p->is_thread == TRUE)
        return q;
    // 만약 오른쪽 자식이면 다시 가장 왼쪽 노드로 이동
    while( q->left != NULL ) q = q->left;
    return q;
}
```

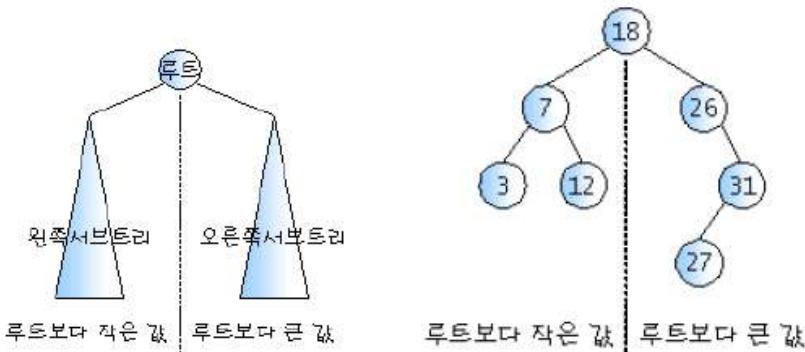
## \* 이진탐색트리

- \* 탐색작업을 효율적으로 하기 위한 자료구조
- \* 탐색은 레코드 집합에서 특정한 레코드를 찾아내는 작업
- \* 주요키 (primary key)
  - 각 레코드를 구별하는 키



## \* 이진탐색트리의 정의

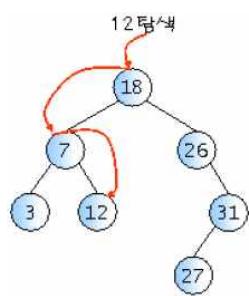
- 모든 노드의 키는 유일하다
- 왼쪽 서브 트리의 키들은 루트의 키보다 작다
- 오른쪽 서브 트리의 키들은 루트의 키보다 크다
- 왼쪽과 오른쪽 서브트리도 이진 탐색 트리이다.



## \* 이진탐색트리에서의 탐색 연산

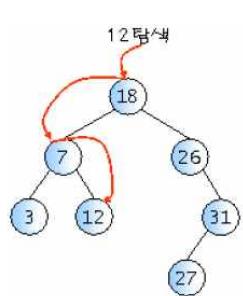
- 비교한 결과가 같으면 탐색이 성공적으로 끝난다.
- 비교한 결과가, 주어진 키 값이 루트 노드의 키 값보다 작으면 탐색은 이 루트 노드의 왼쪽 자식을 기준으로 다시 시작한다.
- 비교한 결과가, 주어진 키 값이 루트 노드의 키 값보다 크면 탐색이 루트 노드의 오른쪽 자식을 기준으로 다시 시작한다.

## \* 순환적인 방법



```
//순환적인 탐색 함수
TreeNode *search(TreeNode *node, int key)
{
    if ( node == NULL ) return NULL;
    if ( key == node->key ) return node;    (1)
    else if ( key < node->key )
        return search(node->left, key);     (2)
    else
        return search(node->right, key);    (3)
}
```

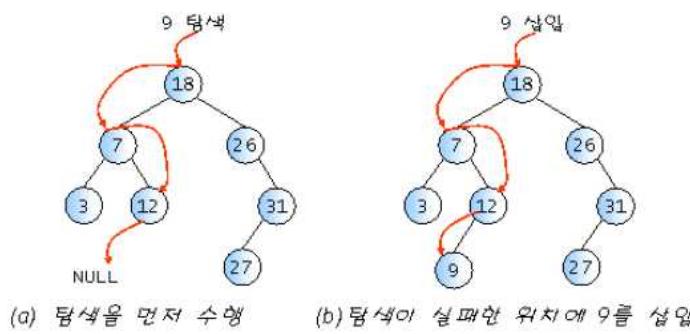
## \* 반복적인 방법



```
// 반복적인 탐색 함수
TreeNode *search(TreeNode *node, int key)
{
    while(node != NULL){
        if( key == node->key ) return node;
        else if( key < node->key )
            node = node->left;
        else
            node = node->right;
    }
    return NULL; // 탐색에 실패했을 경우 NULL 반환
}
```

### \* 이진탐색트리 삽입 연산

- 삽입 전 먼저 탐색을 수행
- 탐색에 실패한 위치가 바로 새로운 노드를 삽입하기 위한 위치



*insert\_node(T,z)*

```

p←NULL;
t←root;
while t≠NULL do
  p←t;
  if z->key < p->key
    then t←p->left;
    else t←p->right;
  if p=NULL
    then root←z;           // 트리가 비어있음
  else if z->key < p->key
    then p->left←z
    else p->right←z
  end if;
end while;
  
```

// key를 이진 탐색 트리 root에 삽입한다.  
// key가 이미 root안에 있으면 삽입되지 않는다.

```

void insert_node(TreeNode **root, int key)
{
  TreeNode *p, *t; // p는 부모노드, t는 현재노드
  TreeNode *n;     // n은 새로운 노드
  t = *root;
  p = NULL;
  // 탐색을 먼저 수행
  while (t != NULL){
    if( key == t->key ) return;
    p = t;
    if( key < t->key ) t = t->left;
    else t = t->right;
  }
}
  
```

// key가 트리 안에 없으므로 삽입 가능  
n = (TreeNode \*) malloc(sizeof(TreeNode));
if( n == NULL ) return;
n->key = key; // 데이터 복사
n->left = n->right = NULL;
// 부모 노드와 링크 연결
if( p != NULL )
 if( key < p->key )
 p->left = n;
 else p->right = n;
else \*root = n;
}

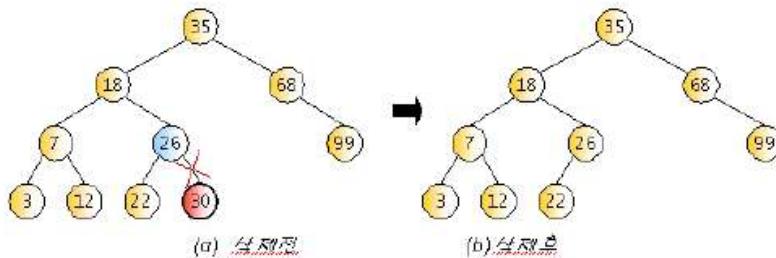
- 함수 안에서 포인터의 값이 바뀌기 때문에 이중 포인터를 쓴다.

### \* 이진탐색트리 삭제 연산

1. 삭제하려는 노드가 단말 노드일 경우
2. 삭제하려는 노드가 하나의 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우
3. 삭제하려는 노드가 두 개의 서브트리 모두 가지고 있는 경우

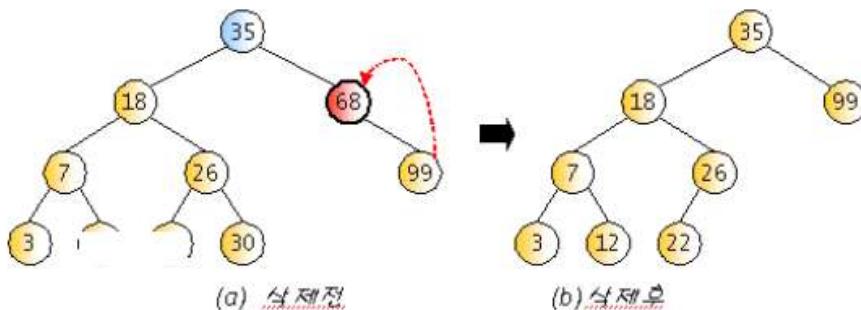
#### \* CASE 1: 삭제하려는 노드가 단말 노드일 경우

- 단말 노드의 부모 노드를 찾아서 연결을 끊으면 된다.

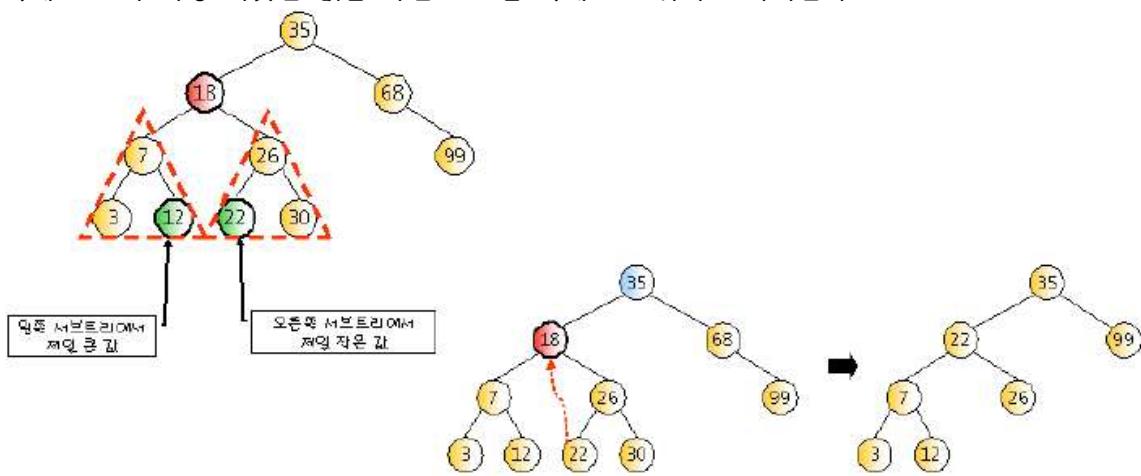


#### \* CASE 2: 삭제하려는 노드가 하나의 서브트리만 갖고 있는 경우

- 삭제되는 노드가 왼쪽이나 오른쪽 서브 트리 중 하나만 갖고 있을 때, 그 노드는 삭제하고 서브 트리는 부모 노드에 붙여준다.



- \* CASE 3: 삭제하려는 노드가 두 개의 서브트리를 갖고 있는 경우
  - 삭제 노드와 가장 비슷한 값을 가진 노드를 삭제노드 위치로 가져온다.



// 삭제 함수

```
void delete_node(TreeNode **root, int key)
{
    TreeNode *p, *t, *child, *succ, *succ_p;
    // key를 갖는 노드 t를 탐색, p는 t의 부모노드
    p = NULL;
    t = *root;
    // key를 갖는 노드 t를 탐색한다.
    while( t != NULL && t->key != key ){
        p = t;
        t = ( key < t->key ) ? t->left : t->right;
    }
    // 탐색이 종료된 시점에 t가 NULL이면 트리안에 key가 없음
    if( t == NULL ) { // 탐색트리에 없는 키
        printf("key is not in the tree");
        return;
    }
}
```

// 첫번째 경우: 단말노드인 경우

```
if( (t->left==NULL) && (t->right==NULL) ){
    if( p != NULL ){
        // 부모노드의 자식필드를 NULL로 만든다.
        if( p->left == t )
            p->left = NULL;
        else p->right = NULL;
    }
    else // 만약 부모노드가 NULL이면 삭제되는 노드가 루트
        *root = NULL;
}
```

// 세번째 경우: 두개의 자식을 가지는 경우

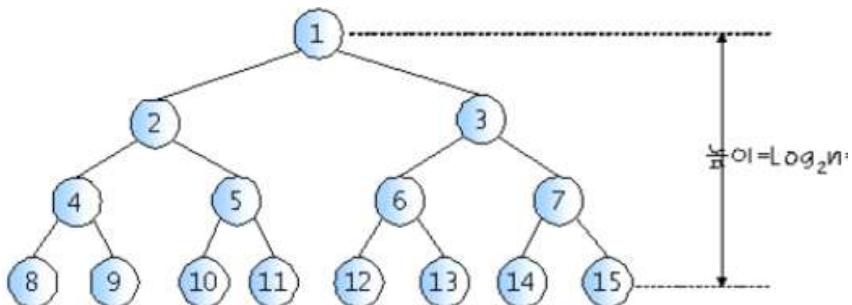
```
else{
    // 오른쪽 서브트리에서 후계자를 찾는다.
    succ_p = t;
    succ = t->right;
    // 후계자를 찾아서 계속 원쪽으로 이동한다.
    while(succ->left != NULL){
        succ_p = succ;
        succ = succ->left;
    }
    // 후속자의 부모와 자식을 연결
    if( succ_p->left == succ )
        succ_p->left = succ->right;
    else
        succ_p->right = succ->right;
    t->key = succ->key; // 후속자가 가진 키값을 현재 노드에 복사
    t = succ; // 원래의 후속자 삭제
}
free(t);
}
```

// 두번째 경우: 하나의 자식만 가지는 경우

```
else if((t->left==NULL)|(t->right==NULL)){
    child = (t->left != NULL) ? t->left : t->right;
    if( p != NULL ){
        if( p->left == t ) // 부모를 자식과 연결
            p->left = child;
        else p->right = child;
    }
    else //만약 부모노드가 NULL이면 삭제되는 노드가 루트
        *root = child;
}
```

#### \* 이진탐색트리의 성능 분석

- 탐색, 삽입, 삭제 연산의 시간 복잡도는  $h(\text{높이})$ 에 비례 한다.



#### \* 최선의 경우

- 이진 트리가 균형적으로 생성되어 있는 경우
- $h = \log_2 n$

#### \* 최악의 경우

- 한쪽으로 치우친 경사 이진트리의 경우
- $h = n$
- 순차탐색과 시간복잡도가 같다.

필드 : 엑셀의 열이라 보면 됨.

시험 : 그림 위주로 공부하기

- \* 트리주고, 번호로 표기하라 (번호 : 배열의 인덱스가 됨)