

윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 01. 자료구조와 알고리즘의 이해

Introduction To Data Structures Using C

Chapter 01. 자료구조와 알고리즘의 이해



Chapter 01-1:

자료구조에 대한 기본적인 이해



C언어와 관련하여 알고 있다고 가정하는 부분

- ▶ 구조체를 정의할 줄 안다.
- ▶ 메모리의 동적 할당과 관련하여 이해한다.
- ▶ 포인터와 관련하여 이해와 활용에 부담이 없다.
- ▶ 헤더파일의 정의하고 활용할 줄 안다.
- ▶ 각종 매크로를 이해하고 #ifndef ~ #endif의 의미를 안다.
- ▶ 다수의 소스파일, 헤더파일로 구성된 프로그램 작성 가능!
- ▶ 재귀함수에 어느 정도 익숙하다.



자료구조란 무엇인가?

자료구조

“프로그램이란 데이터를 표현하고,

표현에는 저장의 의미가 포함된다!

알고리즘

그렇게 표현된 데이터를 처리하는 것이다.”



자료구조의 분류



자료구조와 알고리즘

```
int main(void)
{
    // 배열의 선언
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    . . .

    // 배열에 저장된 값의 합
    for(idx=0; idx<10; idx++)
        sum += arr[idx];

    . . .
}
```

자료구조

알고리즘

알고리즘은 자료구조에 의존적이다!



본서에서 자료구조를 설명하는 방향

- ▶ 자료구조 학습 방법의 구분
 - 자료구조의 모델 자체에 대한 이해 중심 학습
 - 코드 레벨에서의 자료구조 구현 중심 학습
- ▶ 자료구조 학습에 있어서 기억할 것 두 가지
 - 자료구조의 모델을 그림으로 우선 이해해야 한다.
 - 구현이 가능해야만 의미가 있는 것은 아니다.



Chapter 01. 자료구조와 알고리즘의 이해

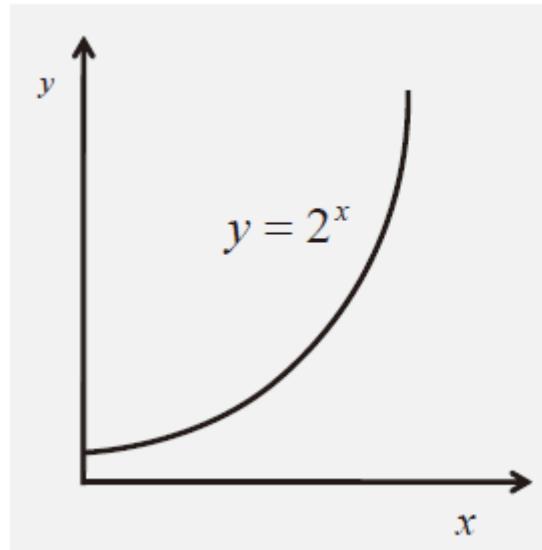


Chapter 01-2:

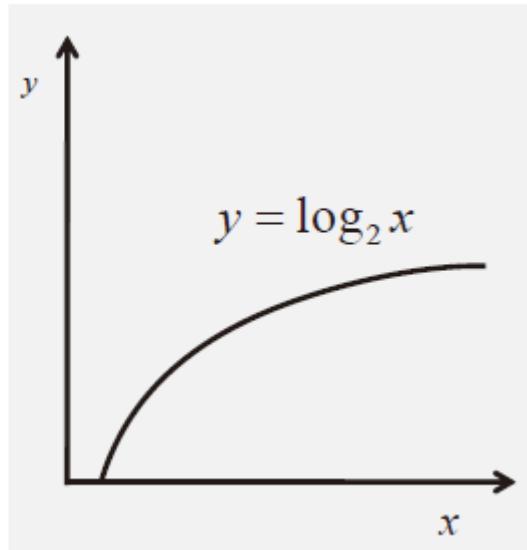
알고리즘의 성능분석 방법



수학과 관련해서 알고 있다고 가정하는 것



지수식



로그식

X 축이 데이터의 수! Y축이 연산의 횟수를 의미한다면?



시간 복잡도 & 공간 복잡도 1

- ▶ 알고리즘을 평가하는 두 가지 요소
 - 시간 복잡도(time complexity) → 얼마나 빠른가?
 - 공간 복잡도(space complexity) → 얼마나 메모리를 적게 쓰는가?
 - 시간 복잡도를 더 중요시 한다.

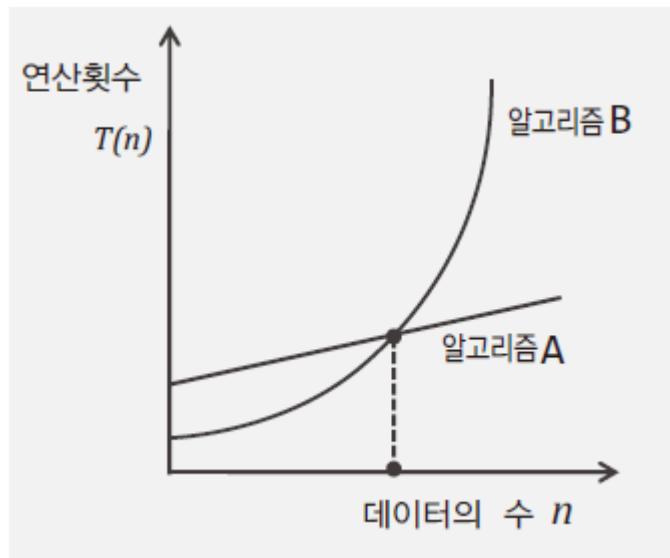
- ▶ 시간 복잡도의 평가 방법
 - 중심이 되는 특정 연산의 횟수를 세어서 평가를 한다.
 - 데이터의 수에 대한 연산횟수의 함수 $T(n)$ 을 구한다.



시간 복잡도 & 공간 복잡도 2

▶ 알고리즘의 수행 속도 비교 기준

- 데이터의 수가 적은 경우의 수행 속도는 큰 의미가 없다.
- 데이터의 수에 따른 수행 속도의 변화 정도를 기준으로 한다.



두 알고리즘을 비교 평가해보면?



순차 탐색 알고리즘과 시간 복잡도

```
// 순차 탐색 알고리즘 적용된 함수  
int LSearch(int ar[], int len, int target)  
{  
    int i;  
    for(i=0; i<len; i++)  
    {  
        if(ar[i] == target)  
            return i;      // 찾은 대상의 인덱스 값 반환  
    }  
    return -1;      // 찾지 못했음을 의미하는 값 반환  
}
```

어떠한 연산자의 연산 횟수를
세어야 하겠는가?



최악의 경우와 최상의 경우

▶ 순차 탐색 상황 하나: 운이 좋은 경우

- 배열의 맨 앞에서 대상을 찾는 경우
- 만족스러운 상황이므로 성능평가의 주 관심이 아니다!
- ‘최상의 경우(best case)’라 한다.

시간 복잡도와 공간 복잡도 각각에 대해서 최악의 경우와 최상의 경우를 구할 수 있다.

▶ 순차 탐색 상황 둘: 운이 좋지 않은 경우

- 배열의 끝에서 찾거나 대상이 저장되지 않은 경우
- 만족스럽지 못한 상황이므로 성능평가의 주 관심이다!
- ‘최악의 경우(worst case)’라 한다.



평균적인 경우(Average Case)

- ▶ 가장 현실적인 경우에 해당한다.
 - 일반적으로 등장하는 상황에 대한 경우의 수이다.
 - 최상의 경우와 달리 알고리즘 평가에 도움이 된다.
 - 하지만 계산하기가 어렵다. 객관적 평가가 쉽지 않다.

- ▶ 평균적인 경우의 복잡도 계산이 어려운 이유
 - ‘평균적인 경우’의 연출이 어렵다.
 - ‘평균적인 경우’임을 증명하기 어렵다.
 - ‘평균적인 경우’는 상황에 따라 달라진다. 반면 최악의 경우는 늘 동일하다.



순차 탐색 최악의 경우 시간 복잡도

“데이터의 수가 n 개일 때,

최악의 경우에 해당하는 연산횟수는(비교연산의 횟수는) n 이다.”

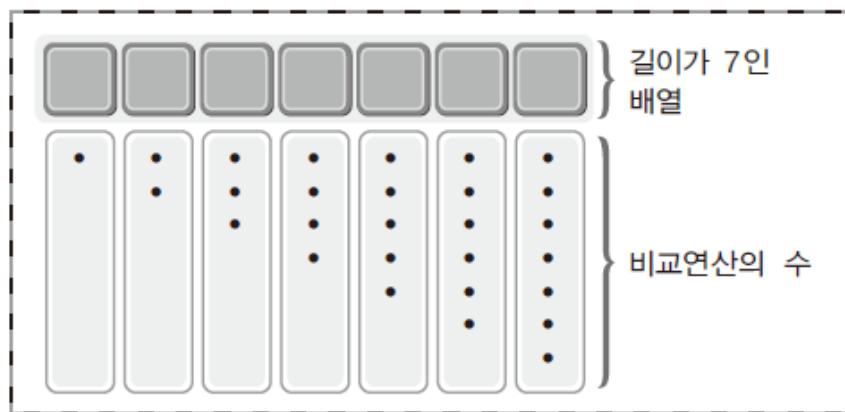
$$T(n) = n$$

최악의 경우를 대상으로 정의한 함수 $T(n)$



순차 탐색 평균적 경우 시간 복잡도

- 가정 1 탐색 대상이 배열에 존재하지 않을 확률 **50%**
- 가정 2 배열 첫 요소부터 마지막 요소까지 탐색 대상 존재 확률 **동일!**
- 탐색 대상이 **존재하지 않는 경우**의 연산횟수는 n
- 가정 2에 의해서 탐색 대상이 **존재하는 경우**의 연산횟수는 $n/2$



$$T(n) = n \times \frac{1}{2} + \frac{n}{2} \times \frac{1}{2} = \frac{3}{4}n$$

결론!

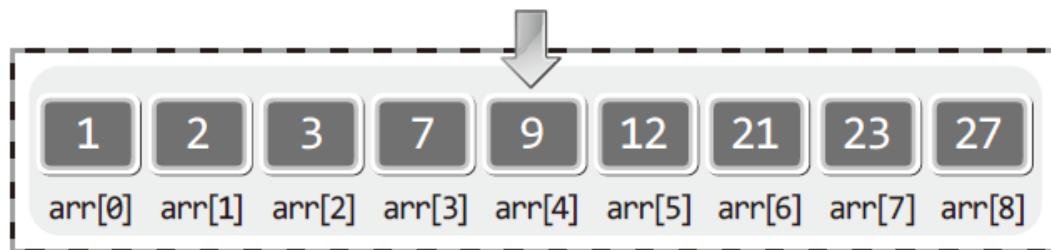
가정 1과 2는 평균적인 경우라 할 수 있겠는가? 그렇다면 무엇이 평균적인 경우이겠는가?



이진 탐색 알고리즘의 소개 1

☞ 이진 탐색 알고리즘의 첫 번째 시도:

1. 배열 인덱스의 시작과 끝은 각각 0과 8이다.
2. 0과 8을 합하여 그 결과를 2로 나눈다.
3. 2로 나눠서 얻은 결과 4를 인덱스 값으로 하여 arr[4]에 저장된 값이 3인지 확인!



3이 저장되어 있는지를
탐색한다!

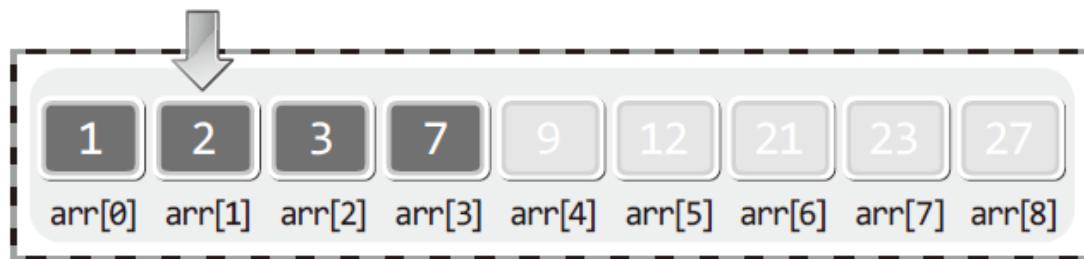
순차 탐색보다 훨씬 좋은 성능을 보이지만, 배열이 정렬되어 있어야 한다는 제약이 따른다.



이진 탐색 알고리즘의 소개 2

☞ 이진 탐색 알고리즘의 두 번째 시도:

1. arr[4]에 저장된 값 9와 탐색 대상인 3의 대소를 비교한다.
2. 대소 비교결과는 $\text{arr}[4] > 3$ 이므로 탐색 범위를 인덱스 기준 0~3으로 제한!
3. 0과 3을 더하여 그 결과를 2로 나눈다. 이때 나머지는 버린다.
4. 2로 나눠서 얻은 결과가 1이니 arr[1]에 저장된 값이 3인지 확인한다.



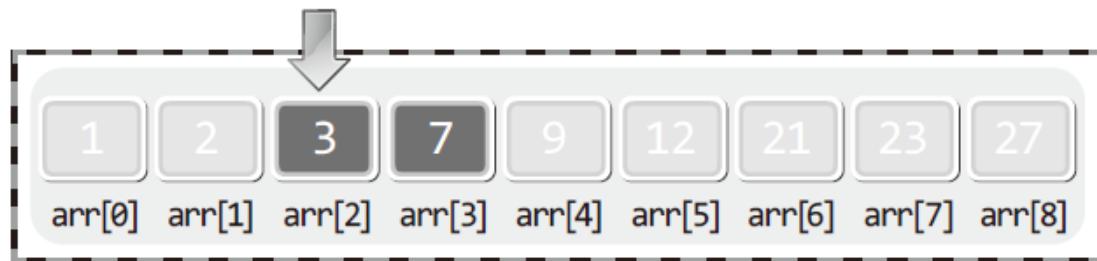
탐색의 대상이 첫 번째 시도 이후 반으로 줄었다!



이진 탐색 알고리즘의 소개 3

☞ 이진 탐색 알고리즘의 세 번째 시도:

1. arr[1]에 저장된 값 2와 탐색 대상인 3의 대소를 비교한다.
2. 대소 비교결과 $\text{arr}[1] < 3$ 이므로 탐색의 범위를 인덱스 기준 2~3으로 제한!
3. 2와 3을 더하여 그 결과를 2로 나눈다. 이때 나머지는 버린다.
4. 2로 나눠서 얻은 결과가 2이니 arr[2]에 저장된 값이 3인지 확인한다.



탐색의 대상이 다시 반으로 줄었다!



이진 탐색 알고리즘의 소개 4

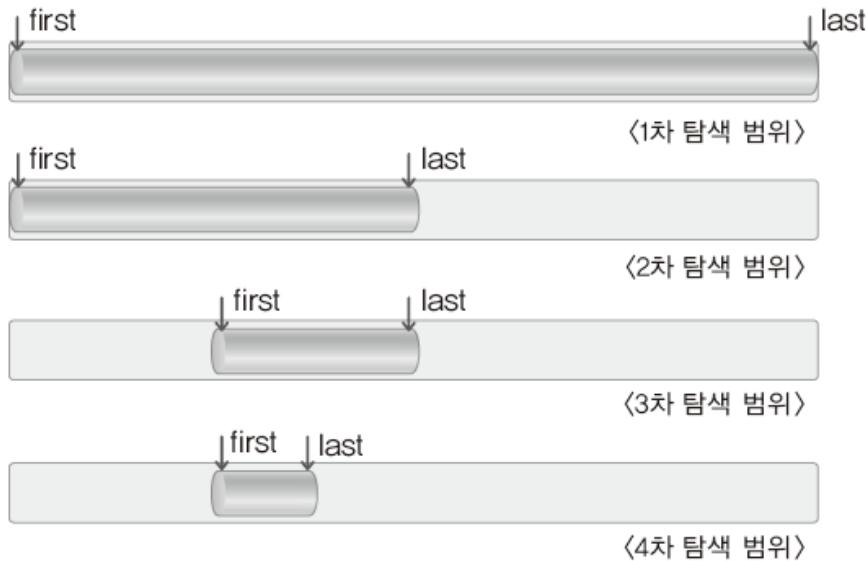


이진 탐색 알고리즘의 핵심!

이진 탐색의 매 과정마다 탐색의 대상을 반씩 줄여나가기 때문에 순차 탐색보다 비교적 좋은 성능을 보인다.



이진 탐색 알고리즘의 구현 1



- first와 last가 만났다는 것은 탐색 대상이 하나 남았다는 것을 뜻함!
- 따라서 first와 last가 역전될때까지 탐색의 과정을 계속!

이진 탐색의 기본 골격

```
while(first <= last)      // first <= last 가 반복의 조건임에 주의!
{
    // 이진 탐색 알고리즘의 진행
}
```



이진 탐색 알고리즘의 구현 2

```
int BSearch(int ar[], int len, int target)
{
    int first = 0;      // 탐색 대상의 시작 인덱스 값
    int last = len-1;   // 탐색 대상의 마지막 인덱스 값
    int mid;

    while(first <= last)
    {
        mid = (first+last) / 2;      // 탐색 대상의 중앙을 찾는다.
        if(target == ar[mid])       // 중앙에 저장된 것이 타겟이라면
        {
            return mid; // 탐색 완료!
        }
        else // 타겟이 아니라면 탐색 대상을 반으로 줄인다.
        {
            if(target < ar[mid])
                last = mid-1;      // 왜 -1을 하였을까?
            else
                first = mid+1;    // 왜 +1을 하였을까?
        }
    }
    return -1; // 찾지 못했을 때 반환되는 값 -1
}
```

-1 혹은 +1을 추가하지 않으면
 $\text{first} \leq \text{mid} \leq \text{last}$ 가 항상 성립이 되어,
탐색 대상이 존재하지 않는 경우 first 와 last 의
역전 현상이 발생하지 않는다!



이진 탐색 알고리즘 최악의 경우 시간 복잡도1

```
while(first <= last)
{
    mid = (first+last) / 2;
    if(target == ar[mid])
    {
        return mid;
    }
    else // 타겟이 아니라면
    {
        ....
    }
}
```

시간 복잡도 계산을 위한 핵심 연산은?
== 연산자!

따라서 == 연산자의 연산 횟수를 기준으로
시간 복잡도를 결정할 수 있다.

데이터의 수가 n 개일 때, 최악의 경우에 발생하는 비교연산의 횟수는?



이진 탐색 알고리즘 최악의 경우 시간 복잡도2

☞ 데이터의 수가 n개일 때 비교 연산의 횟수

- 처음에 데이터 수가 n개일 때의 탐색과정에서 1회의 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 $n/2$ 개일 때의 탐색과정에서 1회 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 $n/4$ 개일 때의 탐색과정에서 1회 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 $n/8$ 개일 때의 탐색과정에서 1회 비교연산 진행
- $n/1$ 얼마인지 결정되지 않았으니
이 사이에 도대체 몇 번의 비교연산이 진행되는지 알 수가 없다!
- 데이터 수를 반으로 줄여서 그 수가 1개일 때의 탐색과정에서 1회의 비교연산 진행

이러한 표현 방법으로는 객관적 성능 비교 불가!



이진 탐색 알고리즘 최악의 경우 시간 복잡도3

☞ 비교 연산의 횟수의 예

- 8이 10이 되기까지 2로 나눈 횟수 3회, 따라서 **비교연산 3회** 진행
- 데이터가 1개 남았을 때, 이때 마지막으로 **비교연산 1회** 진행

☞ 비교 연산의 횟수의 일반화

- n 이 10이 되기까지 2로 나눈 횟수 k 회, 따라서 **비교연산 k 회** 진행
- 데이터가 1개 남았을 때, 이때 마지막으로 **비교연산 1회** 진행

☞ 비교 연산의 횟수의 1차 결론!

- 최악의 경우에 대한 시간 복잡도 함수 $T(n)=k+1$ 이제 k 만 구하면 된다!



이진 탐색 알고리즘 최악의 경우 시간 복잡도4

$$n \times \left(\frac{1}{2}\right)^k = 1$$

n을 몇번 반으로 나눠야 1이 되는가?
k가 나눠야 하는 횟수를 말한다.

☞ k에 관한 식으로…

$$n \times \left(\frac{1}{2}\right)^k = 1 \quad \blacktriangleright \quad n \times 2^{-k} = 1 \quad \blacktriangleright \quad n = 2^k$$

$$n = 2^k \quad \blacktriangleright \quad \log_2 n = \log_2 2^k \quad \blacktriangleright \quad \log_2 n = k \log_2 2 \quad \blacktriangleright \quad \log_2 n = k$$



이진 탐색 알고리즘 최악의 경우 시간 복잡도5

☞ 이제 결론

- 함수 $T(n) = k+1$ 이므로...
- $T(n) = \log_2 n + 1$

☞ 그러나 +1은 생략하여

- $T(n) = \log_2 n$

시간 복잡도의 목적은 n 의 값에 따른 $T(n)$ 의 증가 및 감소의 정도를 판단하는 것이므로 +1은 생략 가능!

그렇다면 +/이 아닌 +200 인 경우도 생략이 가능한가?

빅-오에 대한 이해를 통해서 이에 대한 답을 얻자!



빅-오 표기법(Big-Oh Notation)

$$T(n) = n^2 + 2n + 1$$

↓ 1차 근사치 식

$$T(n) = n^2 + 2n$$

↓ 2차 근사치 식

$$T(n) = \textcircled{n}^2$$

↓ $T(n)$ 의 빅-오

$$O(n^2)$$

빅-오의 표기 방법

$T(n)$ 에서 실제로 영향력을 끼치는 부분을
가리켜 빅-오(Big-Oh)라 한다.

- ☞ n 의 변화에 따른 $T(n)$ 의 변화 정도를 판단하는 것이
목적이니 $+1$ 은 무시할 수 있다.

n	n^2	$2n$	$T(n)$	n^2 의 비율
10	100	20	120	83.33%
100	10,000	200	10,200	98.04%
1,000	1,000,000	2,000	1,002,000	99.80%
10,000	100,000,000	20,000	100,020,000	99.98%
100,000	10,000,000,000	200,000	10,000,200,000	99.99%

- ☞ $2n$ 도 근사치 식의 구성에서 제외시킬 수 있음을 보인 결과!
 n 이 증가함에 따라 $2n+10$ 이 차지하는 비율은 미미해진다.



단순하게 빅-오 구하기

☞ 빅-오는?

- $T(n)$ 이 다항식으로 표현이 된 경우, 최고차항의 차수가 빅-오가 된다.

☞ 빅-오 결정의 예

최고차항의 차수를 빅-오로 결정!

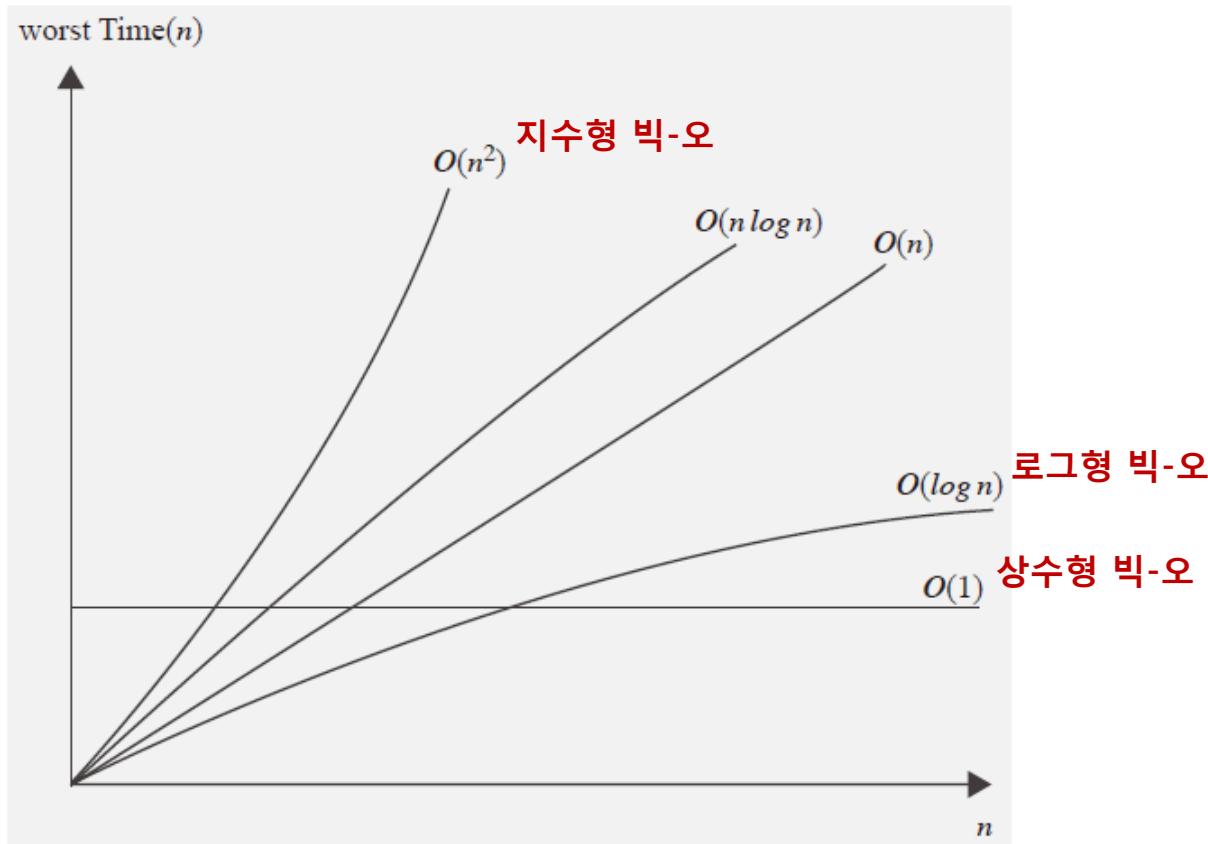
- $T(n) = \underline{n^2} + 2n + 9 \quad \blacktriangleright O(n^2)$
- $T(n) = \underline{n^4} + n^3 + n^2 + 1 \quad \blacktriangleright O(n^4)$
- $T(n) = 5\underline{n^3} + 3n^2 + 2n + 1 \quad \blacktriangleright O(n^3)$

☞ 빅-오 결정의 일반화

- $T(n) = a_m \underline{n^m} + a_{m-1}n^{m-1} + \dots + a_1n^1 + a_0 \quad \blacktriangleright O(n^m)$



대표적인 빅-오



$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



순차 탐색 알고리즘 vs. 이진 탐색 알고리즘

$$\bullet T(n) = n$$



$$O(n)$$

순차 탐색의 최악의 경우 시간 복잡도

순차 탐색의 빅-오

$$\bullet T(n) = \log_2 n + 1$$



$$O(\log n)$$

이진 탐색의 최악의 경우 시간 복잡도

이진 탐색의 빅-오

n	순차 탐색 연산 횟수	이진 탐색 연산 횟수
500	500	9
5,000	5,000	13
50,000	50,000	16

이진 탐색의 연산 횟수는
예제 *BSTWorstOpCount.c*에서
확인한 결과

최악의 경우에 진행이 되는 연산의 횟수



빅-오에 대한 수학적 접근1

☞ 빅-오의 수학적 정의

두 개의 함수 $f(n)$ 과 $g(n)$ 이 주어졌을 때, 모든 $n \geq K$ 에 대하여 $f(n) \leq Cg(n)$ 을 만족하는 두 개의 상수 C 와 K 가 존재하면, $f(n)$ 의 빅-오는 $O(g(n))$ 이다.

☞ 빅-오의 실질적인 의미

$5n^2 + 100$ 의 빅-오는 $O(n^2)$ 이다.

$5n^2 + 100$ 이 아무리 연산횟수의 증가율이 크다 한들 증가율의 패턴이 n^2 을 넘지 못한다!



빅-오에 대한 수학적 접근2

☞ 빅-오 판별의 예

"두 개의 함수 $f(n)$ 과 $g(n)$ 이 주어졌을 때"

- ▶ 두 개의 함수 $f(n) = 5n^2 + 100$, $g(n) = n^2$ 이 주어졌을 때

"모든 $n \geq K$ 에 대하여"

n 의 값이 점차 증가하여 어느 순간 이후부터

- ▶ 모든 $n \geq 12$ 에 대하여, $f(n) \leq Cg(n)$ 을 만족하게 하는 두 개의 상수 C 와 K 가 존재한다면,

" $f(n) \leq Cg(n)$ 을 만족하는 두 개의 상수 C 와 K 가 존재하면"

- ▶ $5n^2 + 100 \leq 3500n^2$ 을 만족하는 $3500(C)$ 과 $12(K)$ 가 존재하니,

" $f(n)$ 의 빅-오는 $O(g(n))$ 이다."

- ▶ $5n^2 + 100$ 의 빅-오는 $O(n^2)$ 이다.



수고하셨습니다~



Chapter 01에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 02. 재귀(Recursion)

Introduction To Data Structures Using C

Chapter 02. 재귀(Recursion)

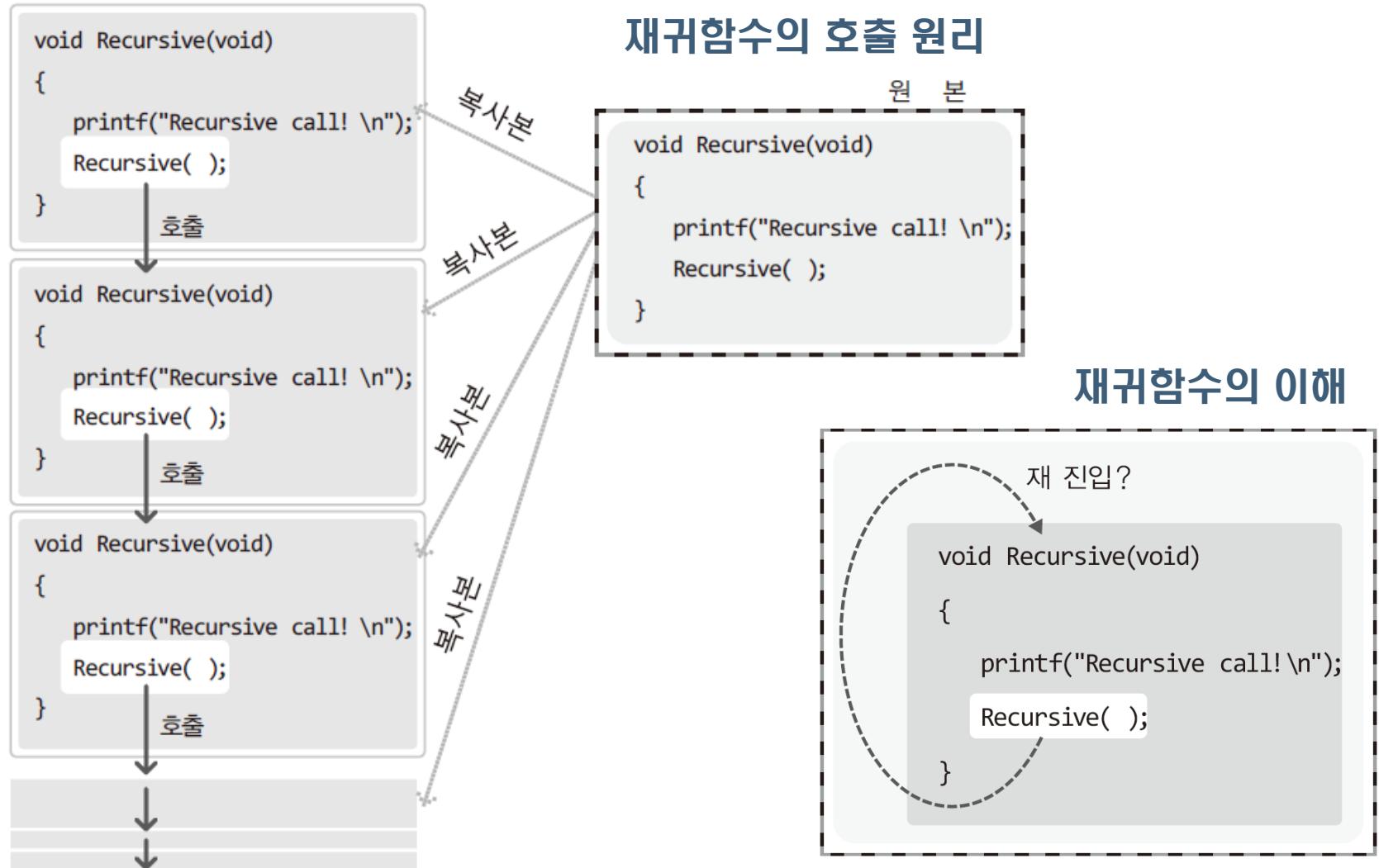


Chapter 02-1:

함수의 재귀적 호출의 이해



재귀함수의 기본적인 이해 1



재귀함수의 기본적인 이해 2

```
void Recursive(int num)
{
    if(num <= 0)          // 재귀의 탈출조건
        return;            //재귀의 탈출!
    printf("Recursive call! %d \n", num);
    Recursive(num-1);
}

int main(void)
{
    Recursive(3);
    return 0;
}
```

재귀함수의 간단한 예

실행결과

```
Recursive call! 3
Recursive call! 2
Recursive call! 1
```



재귀함수의 디자인 사례

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 2 \times 1$$

$(n-1)!$



$$n! = n \times (n-1)!$$

$$f(n) = \begin{cases} n \times f(n-1) & \dots n \geq 1 \\ 1 & \dots n = 0 \end{cases}$$

```
if(n == 0)
    return 1;
```

```
else
    return n * Factorial(n-1);
```



팩토리얼의 재귀적 구현

```
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}

int main(void)
{
    printf("1! = %d \n", Factorial(1));
    printf("2! = %d \n", Factorial(2));
    printf("3! = %d \n", Factorial(3));
    printf("4! = %d \n", Factorial(4));
    printf("9! = %d \n", Factorial(9));
    return 0;
}
```

팩토리얼 구현 결과

실행결과

```
1! = 1
2! = 2
3! = 6
4! = 24
9! = 362880
```



Chapter 02. 재귀(Recursion)



Chapter 02-2:

재귀의 활용



피보나치 수열 1: 이해

피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 . . .

피보나치 수열의 구성

수열의 n 번째 값 = 수열의 $n-1$ 번째 값 + 수열의 $n-2$ 번째 값

피보나치 수열의 표현

$$fib(n) = \begin{cases} 0 & \dots n=1 \\ 1 & \dots n=2 \\ fib(n-1) + fib(n-2) & \dots \text{otherwise} \end{cases}$$



피보나치 수열 2: 코드의 구현

$$fib(n) = \begin{cases} 0 & \dots n=1 \\ 1 & \dots n=2 \\ fib(n-1) + fib(n-2) & \dots \text{otherwise} \end{cases}$$

```
int Fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    else
        return Fibo(n-1) + Fibo(n-2);
}
```

피보나치 수열 3: 완성된 예제

```
int Fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    else
        return Fibo(n-1) + Fibo(n-2);
}

int main(void)
{
    int i;
    for(i=1; i<15; i++)
        printf("%d ", Fibo(i));

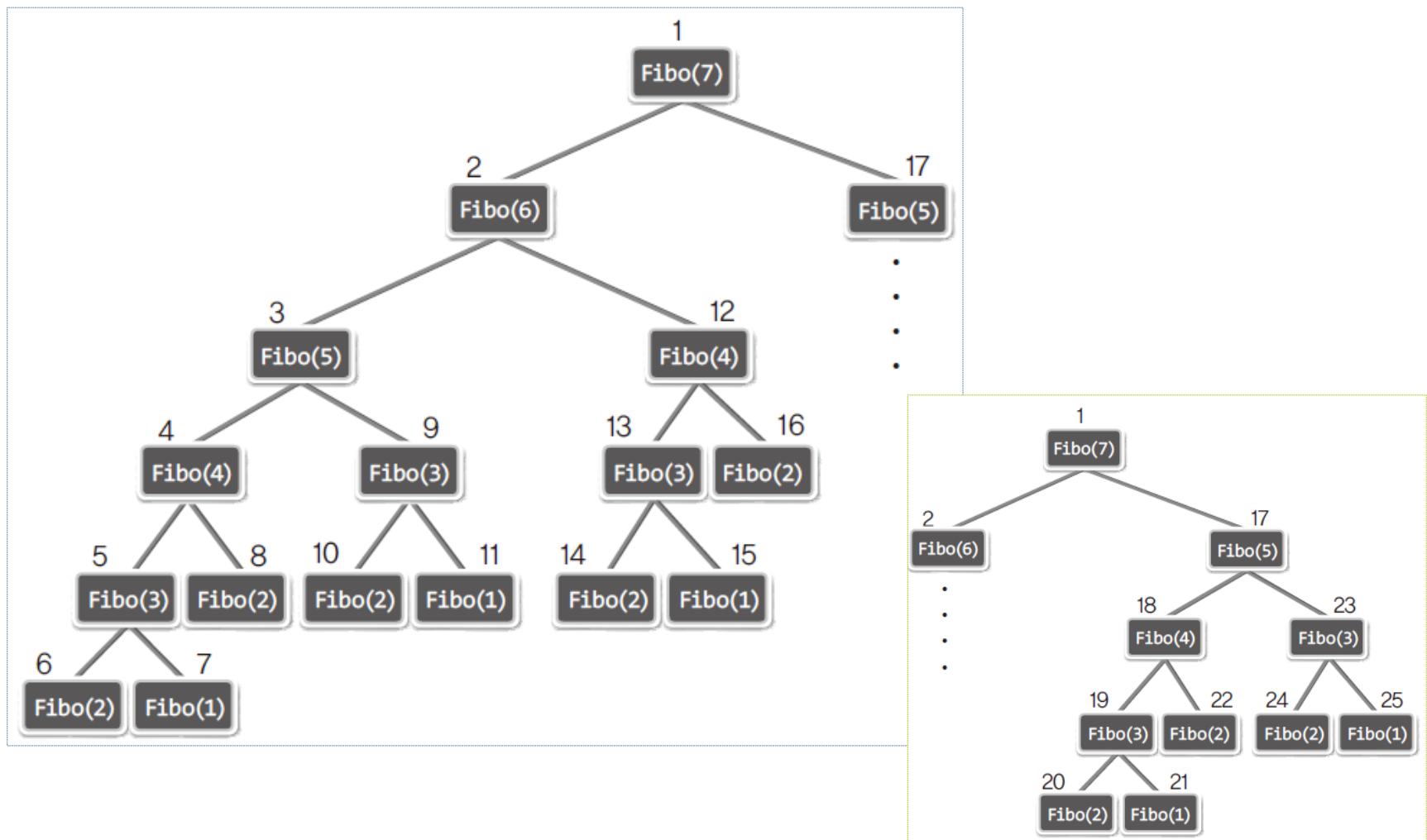
    return 0;
}
```

실행결과

0	1	1	2	3	5	8	13	21	34	55	89	144	233
---	---	---	---	---	---	---	----	----	----	----	----	-----	-----



피보나치 수열 4: 함수의 흐름



이진 탐색 알고리즘의 재귀구현 1

이진 탐색의 알고리즘의 핵심

1. 탐색 범위의 중앙에 목표 값이 저장되었는지 확인
2. 저장되지 않았다면 탐색 범위를 반으로 줄여서 다시 탐색 시작

이진 탐색의 종료에 대한 논의

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    if(first > last)          first와 last는 각각 탐색의 시작과 끝을 나타내는 변수
        return -1;           // -1의 반환은 탐색의 실패를 의미
    . . .
}
```



이진 탐색 알고리즘의 재귀구현 2

탐색 대상의 확인!

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;
    탐색의 대상에서 중심에 해당하는 인덱스 값 계산
    mid = (first+last) / 2;
    if(ar[mid] == target) 타겟이 맞는지 확인!
        return mid;
    . . .
}
```



이진 탐색 알고리즘의 재귀구현 3

계속되는 탐색

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;
    mid = (first+last) / 2;

    if(ar[mid] == target)
        return mid;
    else if(target < ar[mid])
        return BSearchRecur(ar, first, mid-1, target); 앞부분을 대상으로 재 탐색
    else
        return BSearchRecur(ar, mid+1, last, target); 뒷부분을 대상으로 재 탐색
}
```



Chapter 02. 재귀(Recursion)



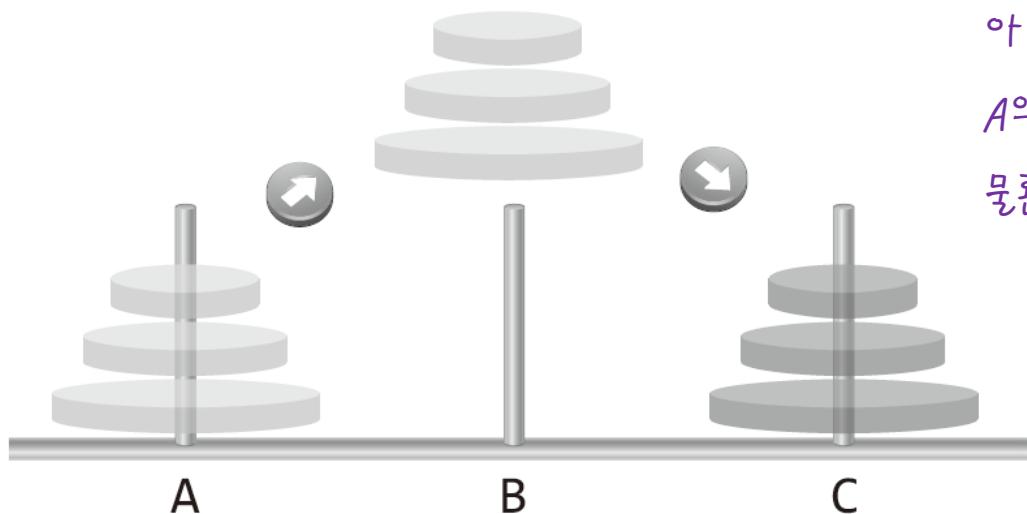
Chapter 02-3:

하노이 타워



하노이 타워 문제의 이해

원반을 A에서 C로 이동하기



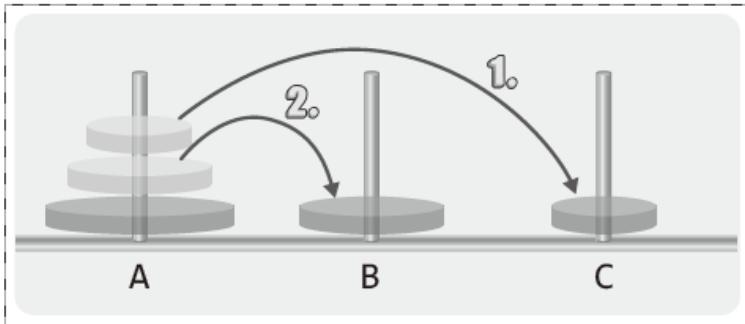
아래의 제약사항을 만족하면서
A의 모든 원반을 C로 옮기는 문제!
물론 옮기는 과정에서 B를 활용한다.

제약사항

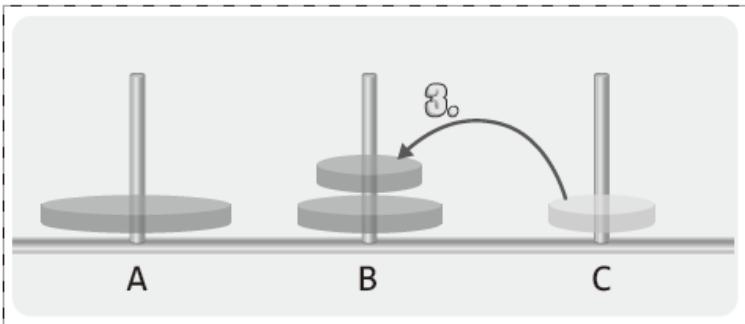
- 원반은 한 번에 하나씩만 옮길 수 있습니다.
- 옮기는 과정에서 작은 원반의 위에 큰 원반이 올려져서는 안됩니다.



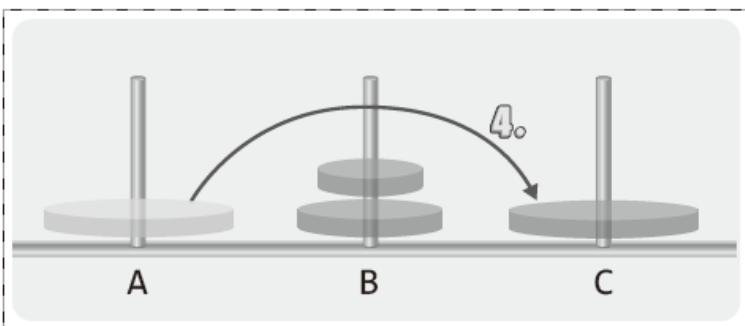
하노이 타워 문제 해결의 예 1



▶ [그림 02-8: 문제해결 1/5]



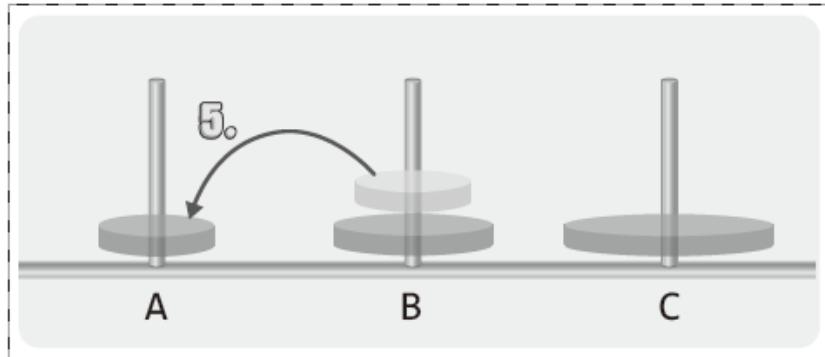
▶ [그림 02-9: 문제해결 2/5]



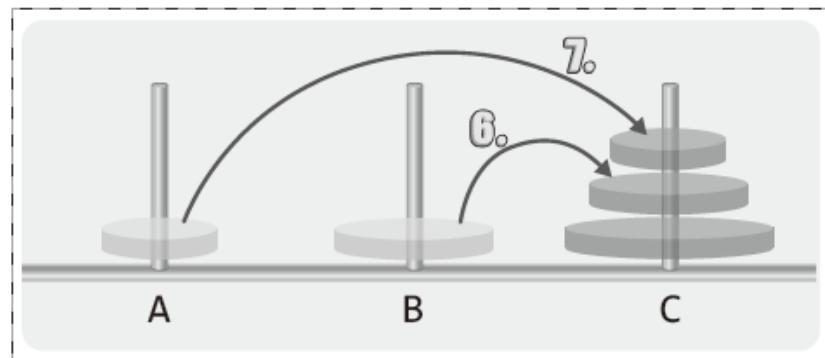
▶ [그림 02-10: 문제해결 3/5]



하노이 타워 문제 해결의 예 2



▶ [그림 02-11: 문제해결 4/5]

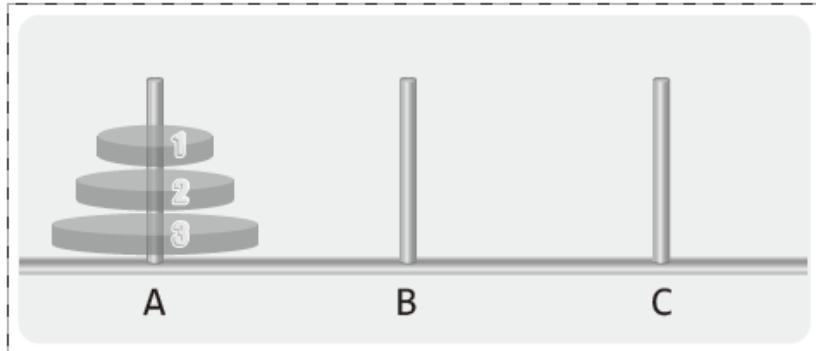


▶ [그림 02-12: 문제해결 5/5]

지금 보인 과정에서 반복의 패턴을 찾아야 문제의 해결을 위한 코드를 작성할 수 있다!



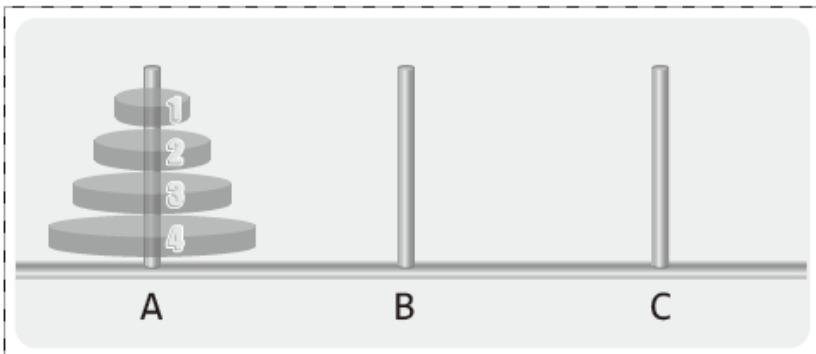
반복되는 일련의 과정을 찾기 위한 힌트



A의 세 원반을 C로 옮기기 위해서는 원반 3을 C로 옮겨야 한다. 그리고 이를 위해서는 원반 1과 2를 우선 원반 B로 옮겨야 한다.

▶ [그림 02-13: 원반이 3개인 하노이 타워]

위와 아래의 두 예를 통해서 문제의 해결에 있어서 반복이 되는 패턴이 있음을 알 수 있다.

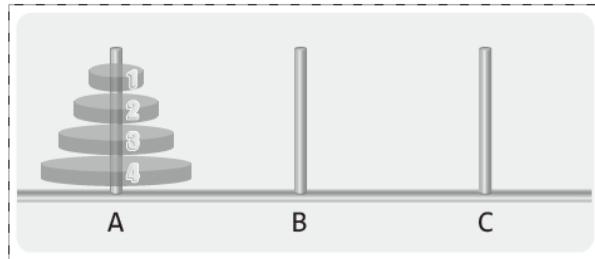


A의 네 원반을 C로 옮기기 위해서는 원반 4를 C로 옮겨야 한다. 그리고 이를 위해서는 원반 1과 2와 3을 우선 원반 B로 옮겨야 한다.

▶ [그림 02-14: 원반이 4개인 하노이 타워]

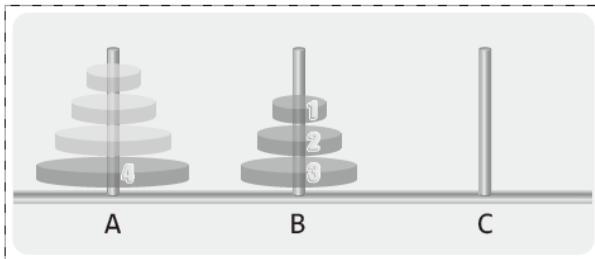


하노이 타워의 반복패턴 연구 1



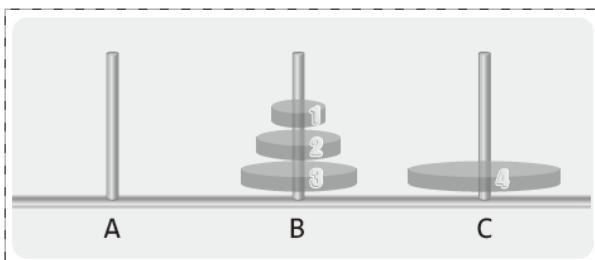
0. 원반 4개를 A에서 C로 이동

▶ [그림 02-14: 원반이 4개인 하노이 타워]



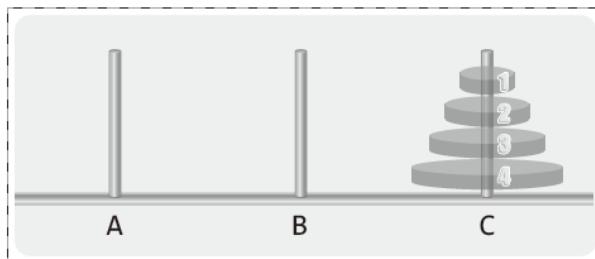
1. 작은 원반 3개를 A에서 B로 이동

▶ [그림 02-15: 반복패턴 1/3]



2. 큰 원반 1개를 A에서 C로 이동

▶ [그림 02-16: 반복패턴 2/3]



3. 작은 원반 3개를 B에서 C로 이동

▶ [그림 02-17: 반복패턴 3/3]



하노이 타워의 반복패턴 연구 2

0. 원반 4개를 A에서 C로 이동

0. 큰 원반 n개를 A에서 C로 이동

1. 작은 원반 3개를 A에서 B로 이동

1. 작은 원반 n-1개를 A에서 B로 이동

2. 큰 원반 1개를 A에서 C로 이동

2. 큰 원반 1개를 A에서 C로 이동

3. 작은 원반 3개를 B에서 C로 이동

3. 작은 원반 n-1개를 B에서 C로 이동



하노이 타워 문제의 해결 1

하노이 타워 함수의 기본 골격

```
void HanoiTowerMove(int num, char from, char by, char to)
```

```
{
```

원반 num의 수에 해당하는 원반을 from에서 to로
이동을 시키되 그 과정에서 by를 활용한다.

```
}
```

0. 큰 원반 n개를 A에서 C로 이동 `HanoiTowerMove(num, from, by, to);`
1. 작은 원반 n-1개를 A에서 B로 이동 `HanoiTowerMove(num-1, from, to, by);`
2. 큰 원반 1개를 A에서 C로 이동 `printf(. . .);`
3. 작은 원반 n-1개를 B에서 C로 이동 `HanoiTowerMove(num-1, by, from, to);`



하노이 타워 문제의 해결 2

- | | |
|--------------------------|--|
| 0. 큰 원반 n개를 A에서 C로 이동 | HanoiTowerMove(num, from , by , to); |
| 1. 작은 원반 n-1개를 A에서 B로 이동 | HanoiTowerMove(num-1, from , to , by); |
| 2. 큰 원반 1개를 A에서 C로 이동 | printf(. . .); |
| 3. 작은 원반 n-1개를 B에서 C로 이동 | HanoiTowerMove(num-1, by , from , to); |

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)          // 이동할 원반의 수가 1개라면
    {
        printf("원반1을 %c에서 %c로 이동 \n", from, to);
    }
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반%d을(를) %c에서 %c로 이동 \n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}
```



하노이 타워 문제의 해결 3

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)          // 이동할 원반의 수가 1개라면
    {
        printf("원반1을 %c에서 %c로 이동 \n", from, to);
    }
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반%d을(를) %c에서 %c로 이동 \n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}

int main(void)
{
    // 막대A의 원반 3개를 막대B를 경유하여 막대C로 옮기기
    HanoiTowerMove(3, 'A', 'B', 'C');
    return 0;
}
```

실행결과

```
원반1을 A에서 C로 이동
원반2을(를) A에서 B로 이동
원반1을 C에서 B로 이동
원반3을(를) A에서 C로 이동
원반1을 B에서 A로 이동
원반2을(를) B에서 C로 이동
원반1을 A에서 C로 이동
```



수고하셨습니다~



Chapter 02에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 03. 연결 리스트(Linked List) 1

Introduction To Data Structures Using C

Chapter 03. 연결 리스트(Linked List) 1



Chapter 03-1:

추상 자료형: Abstract Data Type



추상 자료형(ADT)의 이해

추상 자료형이란?

구체적인 기능의 완성과정을 언급하지 않고, 순수하게 기능이 무엇인지를 나열한 것



지갑의 추상 자료형

- 카드의 삽입
- 카드의 추출(카드를 빼냄)
- 동전의 삽입
- 동전의 추출(동전을 빼냄)
- 지폐의 삽입
- 지폐의 추출(지폐를 빼냄)

기능의 명세를 가리켜 왜 자료형이라 하는 것일까?



지갑을 의미하는 구조체 Wallet의 정의

자료형 Wallet의 정의

```
typedef struct _wallet
{
    int coin100Num;      // 100원짜리 동전의 수
    int bill5000Num;     // 5,000원짜리 지폐의 수
} Wallet;
```

자료형 *int*와 관련해서 우리가 아는것을 나열해 보자!
어떻게 생겼는지 아는 것이 아니라 어떠한 연산이 가능한 한지를 알고 있지 않은가?

완전한 자료형의 정의로 인식되기 위해서는
해당 자료형과 관련이 있는 연산이 함께 정의되어야 한다!

자료형 Wallet 정의의 일부

```
int TakeOutMoney(Wallet * pw, int coinNum, int billNum); // 돈 꺼내는 연산
void PutMoney(Wallet * pw, int coinNum, int billNum);    // 돈 넣는 연산
```



구조체 Wallet의 추상 자료형 정의

Wallet 기반의 main

```
int main(void)
{
    Wallet myWallet;      // 지갑 하나 장만 했음!
    ...
    PutMoney(&myWallet, 5, 10);    // 돈을 넣는다.
    ...
    ret = TakeOutMoney(&myWallet, 2, 5); // 돈을 꺼낸다.
    ...
}
```

 Wallet의 ADT

Operations:

- int TakeOutMoney(Wallet * pw, int coinNum, int billNum)
 - 첫 번째 인자로 전달된 주소의 지갑에서 돈을 꺼낸다.
 - 두 번째 인자로 꺼낼 동전의 수, 세 번째 인자로 꺼낼 지폐의 수를 전달한다.
 - 꺼내고자 하는 돈의 총액이 반환된다. 그리고 그만큼 돈은 차감된다.
- void PutMoney(Wallet * pw, int coinNum, int billNum)
 - 첫 번째 인자로 전달된 주소의 지갑에 돈을 넣는다.
 - 두 번째 인자로 넣을 동전의 수, 세 번째 인자로 넣을 지폐의 수를 전달한다.
 - 넣은 만큼 동전과 지폐의 수가 증가한다.

구조체 Wallet의 정의를 ADT에 포함시키지 않아도 됨을 보인다.



자료구조의 학습에 ADT의 정의를 포함합니다.

자료구조 학습의 옳은 순서

1. 리스트 자료구조의 ADT를 정의한다.
2. ADT를 근거로 리스트 자료구조를 활용하는 main 함수를 정의한다.
3. ADT를 근거로 리스트를 구현한다.

자료구조의 내부 구현을 모르고도 해당 자료구조의 활용이 가능하도록 ADT를 정의하는 것이 옳다.

main 함수를 먼저 접하게 되면, 구현할 자료구조를 구성하는 함수들을 잘 이해 할 수 있다.



Chapter 03. 연결 리스트(Linked List) 1



Chapter 03-2:

배열을 이용한 리스트의 구현



리스트의 이해

리스트의 구분

- 순차 리스트 배열을 기반으로 구현된 리스트
- 연결 리스트 메모리의 동적 할당을 기반으로 구현된 리스트

이는 구현 방법을 기준으로 한 구분이다. 따라서 이 두 리스트의 ADT가 동일하다고 해서 문제가 되지는 않는다.

리스트의 특징

- 저장 형태 데이터를 나란히(하나의 열로) 저장한다.
- 저장 특성 중복이 되는 데이터의 저장을 허용한다.

이것이 리스트의 ADT를 정의하는데 있어서 고려해야 할 유일한 내용이다.



리스트 자료구조의 ADT 1

- `void ListInit(List * plist);`

- 초기화할 리스트의 주소 값을 인자로 전달한다.
- 리스트 생성 후 제일 먼저 호출되어야 하는 함수이다.

리스트의 초기화

- `void LInsert(List * plist, LData data);`

- 리스트에 데이터를 저장한다. 매개변수 data에 전달된 값을 저장한다.

데이터 저장

- `int LFirst(List * plist, LData * pdata);`

- 첫 번째 데이터가 pdata가 가리키는 메모리에 저장된다.
- 데이터의 참조를 위한 초기화가 진행된다.
- 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

저장된 데이터의
탐색 및 탐색 초기화

LData는 저장 대상의 자료형을 결정할 수 있도록 `typedef`로 선언된 자료형의 이름이다.



리스트 자료구조의 ADT 2

- int LNext(List * plist, LData * pdata);

- 참조된 데이터의 다음 데이터가 pdata가 가리키는 메모리에 저장된다.
- 순차적인 참조를 위해서 반복 호출이 가능하다.
- 참조를 새로 시작하려면 먼저 LFirst 함수를 호출해야 한다.
- 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

다음 데이터의

참조(반환)을 목적으로 호출

- LData LRemove(List * plist);

- LFirst 또는 LNext 함수의 마지막 반환 데이터를 삭제한다.
- 삭제된 데이터는 반환된다.
- 마지막 반환 데이터를 삭제하므로 연이은 반복 호출을 허용하지 않는다.

바로 이전에 참조(반환)이

이뤄진 데이터의 삭제

- int LCount(List * plist);

- 리스트에 저장되어 있는 데이터의 수를 반환한다.

현재 저장되어 있는

데이터의 수를 반환



리스트의 ADT를 기반으로 정의된 main 함수

예제 ListMain.c를 봅니다.

- ArrayList.h 리스트 자료구조의 헤더파일
- ArrayList.c 리스트 자료구조의 소스파일
- ListMain.c 리스트 관련 main 함수가 담긴 소스파일

실행을 위해
필요한 파일들

실행결과

현재 데이터의 수: 5

11 11 22 22 33

현재 데이터의 수: 3

11 11 33



리스트의 초기화와 데이터 저장 과정

리스트의 초기화

```
int main(void)
{
    List list;          // 리스트의 생성
    ....
    ListInit(&list);   // 리스트의 초기화
    ....
}
```

```
int main(void)
{
    ....
    LInsert(&list, 11); // 리스트에 11을 저장
    LInsert(&list, 22); // 리스트에 22를 저장
    LInsert(&list, 33); // 리스트에 33을 저장
    ....
}
```

초기화된 리스트에 데이터 저장



리스트의 데이터 참조 과정

```
int main(void)
{
    ...
    if( LFirst(&list, &data) )    // 첫 번째 데이터 변수 data에 저장
    {
        printf("%d ", data);
        while( LNext(&list, &data) ) // 두 번째 이후 데이터 변수 data에 저장
            printf("%d ", data);
    }
    ...
}
```

데이터 참조의 새로운 시작을 위해서
LFirst 함수의 호출을 요구한다!

✓ 데이터 참조 일련의 과정

LFirst → LNext → LNext → LNext → LNext . . .



리스트의 데이터 삭제 방법

```
int main(void)
{
    . . .
    if( LFirst(&list, &data) )           LRemove 함수는 연이은 호출을 허용하
    {
        if(data == 22)                지 않는다!
            LRemove(&list);          // 위의 LFirst 함수를 통해 참조한 데이터 삭제!

        while( LNext(&list, &data) )
        {
            if(data == 22)
                LRemove(&list); // 위의 LNext 함수를 통해 참조한 데이터 삭제!
        }
    }
    . . .
}
```

프로그램의 논리상 LRemove 함수의 연이은 호출이 불필요함을 이해하는 것도
중요하다!



배열 기반 리스트의 헤더파일 정의 1

```
#ifndef __ARRAY_LIST_H__
#define __ARRAY_LIST_H__

#define TRUE      1      // '참'을 표현하기 위한 매크로 정의
#define FALSE     0      // '거짓'을 표현하기 위한 매크로 정의

#define LIST_LEN   100

typedef int LData;    저장할 대상의 자료형을 변경을 위한 typedef 선언

typedef struct __ArrayList // 배열기반 리스트를 정의한 구조체
{
    LData arr[LIST_LEN];    // 리스트의 저장소인 배열
    int numOfData;          // 저장된 데이터의 수
    int curPosition;        // 데이터 참조위치를 기록
} ArrayList;    배열 기반 리스트를 의미하는 구조체

typedef ArrayList List;    리스트의 변경을 용이하게 하기 위한 typedef 선언
```

위의 내용 중 일부는 리스트를 배열 기반으로 구현하기 위한 선언을 담고 있다.



배열 기반 리스트의 헤더파일 정의 2

```
void ListInit(List * plist);           // 초기화
void LInsert(List * plist, LData data); // 데이터 저장

int LFirst(List * plist, LData * pdata); // 첫 데이터 참조
int LNext(List * plist, LData * pdata); // 두 번째 이후 데이터 참조

LData LRemove(List * plist);           // 참조한 데이터 삭제
int LCount(List * plist);             // 저장된 데이터의 수 반환

#endif
```

위의 함수들은 리스트 ADT를 기반으로 선언된 함수들이다.

따라서 배열 기반 리스트로 선언된 함수들의 내용을 제한할 필요가 없다.



배열 기반 리스트의 초기화

```
typedef struct __ArrayList
{
    LData arr[LIST_LEN];
    int numOfData;    리스트에 저장된 데이터의 수
    int curPosition; 마지막 참조 위치에 대한 정보 저장
} ArrayList;
```

배열 기반 리스트의 초기화

```
void ListInit(List * plist)
{
    (plist->numOfData) = 0;
    (plist->curPosition) = -1;
}
```

-1은 아무런 위치도 참조하지 않음을 의미함!

실제로 초기화할 대상은 구조체 변수의 멤버이다. 따라서 초기화 함수의 구성은 구조체의 정의를 기반으로 한다.



배열 기반 리스트의 삽입

```
typedef struct __ArrayList
{
    LData arr[LIST_LEN];
    int numOfData;
    int curPosition;
} ArrayList;
```

배열에 데이터 저장!

```
void LInsert(List * plist, LData data)
{
    if(plist->numOfData > LIST_LEN)      // 더 이상 저장할 공간이 없다면
    {
        puts("저장이 불가능합니다.");
        return;
    }

    plist->arr[plist->numOfData] = data;    // 데이터 저장
    (plist->numOfData)++;                  // 저장된 데이터의 수 증가
}
```



배열 기반 리스트의 조회

```
int LFirst(List * plist, LData * pdata)           초기화! 및 첫 번째 데이터 참조
{
    if(plist->numOfData == 0)          // 저장된 데이터가 하나도 없다면
        return FALSE;
    (plist->curPosition) = 0;          // 참조 위치 초기화! 첫 번째 데이터의 참조를 의미!
    *pdata = plist->arr[0];          // pdata가 가리키는 공간에 데이터 저장
    return TRUE;
}                                               만약에 중간 지점에서 다시 처음부터 참조하기 원한다면?
```

```
int LNext(List * plist, LData * pdata)           그 다음 데이터 참조
{
    if(plist->curPosition >= (plist->numOfData)-1)      // 더 이상 참조할 데이터가 없다면
        return FALSE;
    (plist->curPosition)++;
    *pdata = plist->arr[plist->curPosition];
    return TRUE;
}
```

값의 반환은 매개변수를 통해서!
함수의 반환은 성공여부를 알리기 위해서!



배열 기반 리스트의 삭제

삭제되는 데이터는 반환의 과정을 통해서 되돌려주는 것이 옳다!

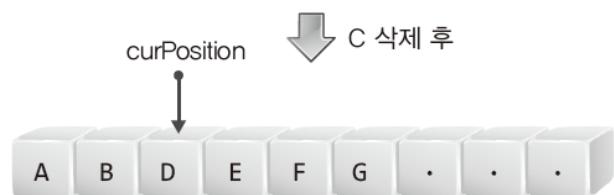
```
LData LRemove(List * plist)
{
    int rpos = plist->curPosition;          // 삭제할 데이터의 인덱스 값 참조
    int num = plist->numOfData;
    int i;
    LData rdata = plist->arr[rpos];         // 삭제할 데이터를 임시로 저장

    // 삭제를 위한 데이터의 이동을 진행하는 반복문
    for(i=rpos; i<num-1; i++)
        plist->arr[i] = plist->arr[i+1];

    (plist->numOfData)--;                  // 데이터의 수 감소
    (plist->curPosition)--;               // 참조위치를 하나 되돌린다.
    return rdata;                         // 삭제된 데이터의 반환
}
```



삭제 기본 모델



참조 위치를 하나 되돌려야 하는 이유



리스트에 구조체 변수 저장하기1:

리스트에 저장할 Point 구조체 변수의 헤더파일 선언

```
typedef struct _point
{
    int xpos;
    int ypos;
} Point;

// Point 변수의 xpos, ypos 값 설정
void SetPointPos(Point * ppos, int xpos, int ypos);

// Point 변수의 xpos, ypos 정보 출력
void ShowPointPos(Point * ppos);

// 두 Point 변수의 비교
int PointComp(Point * pos1, Point * pos2);
```



리스트에 구조체 변수 저장하기2:

구조체 Point 관련 소스파일

```
void SetPointPos(Point * ppos, int xpos, int ypos) {  
    ppos->xpos = xpos;  
    ppos->ypos = ypos;  
}  
  
void ShowPointPos(Point * ppos) {  
    printf("[%d, %d] \n", ppos->xpos, ppos->ypos);  
}  
  
int PointComp(Point * pos1, Point * pos2) {  
    if(pos1->xpos == pos2->xpos && pos1->ypos == pos2->ypos)  
        return 0;  
    else if(pos1->xpos == pos2->xpos)  
        return 1;  
    else if(pos1->ypos == pos2->ypos)  
        return 2;  
    else  
        return -1;  
}
```



리스트에 구조체 변수 저장하기3:

ArrayList.h의 변경 사항 두 가지

```
typedef int LData;      (typedef 선언변경)▶     typedef Point * LData;  
#include "Point.h"    // ArrayList.h에 추가할 헤더파일 선언문
```

typedef 선언에서 보이듯이 실제로 저장을 하는 것은 Point 구조체 변수의 주소 값이다!

실행을 위한 파일 구성

Point.h, Point.c

구조체 Point를 위한 파일

ArrayList.h, ArrayList.c

배열 기반 리스트

PointListMain.c

구조체 Point의 변수 저장



리스트에 구조체 변수 저장하기4:

예제 PointListMain.c를 봅니다.

실행결과

현재 데이터의 수: 4

[2, 1]

[2, 2]

[3, 1]

[3, 2]

현재 데이터의 수: 2

[3, 1]

[3, 2]



PointListMain.c의 일부: 초기화 및 저장

```
int main(void)
{
    List list;
    Point compPos;
    Point * ppos;

    ListInit(&list);

    /*** 4개의 데이터 저장 ***/
    ppos = (Point*)malloc(sizeof(Point));
    SetPointPos(ppos, 2, 1);
    LInsert(&list, ppos);

    ppos = (Point*)malloc(sizeof(Point));
    SetPointPos(ppos, 2, 2);
    LInsert(&list, ppos);

    . . .
}
```



PointListMain.c의 일부: 참조 및 조회

```
int main(void)
{
    .....
    /*** 저장된 데이터의 출력 ***/
    printf("현재 데이터의 수: %d \n", LCount(&list));

    if(LFirst(&list, &ppos))
    {
        ShowPointPos(ppos);

        while(LNext(&list, &ppos))
            ShowPointPos(ppos);
    }
    printf("\n");
    .....
}
```



PointListMain.c의 일부: 삭제

```
int main(void)
{
    .....
    /*** xpos가 2인 모든 데이터 삭제 ***/
    compPos.xpos=2;
    compPos.ypos=0;

    if(LFirst(&list, &ppos)) {
        if(PointComp(ppos, &compPos)==1) {
            ppos=LRemove(&list);
            free(ppos);
        }
        while(LNext(&list, &ppos)) {
            if(PointComp(ppos, &compPos)==1) {
                ppos=LRemove(&list);
                free(ppos);
            }
        }
    }
    .....
}
```



배열 기반 리스트의 장점과 단점

✓ 배열 기반 리스트의 단점

- 배열의 길이가 초기에 결정되어야 한다. 변경이 불가능하다.
- 삭제의 과정에서 데이터의 이동(복사)가 매우 빈번히 일어난다.

배열의 일반적인 단점

✓ 배열 기반 리스트의 장점

- 데이터 참조가 쉽다. 인덱스 값 기준으로 어디든 한 번에 참조 가능!

배열의 일반적인 장점



수고하셨습니다~



Chapter 03에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 04. 연결 리스트(Linked List) 2

Introduction To Data Structures Using C

Chapter 04. 연결 리스트(Linked List) 2



Chapter 04-1:

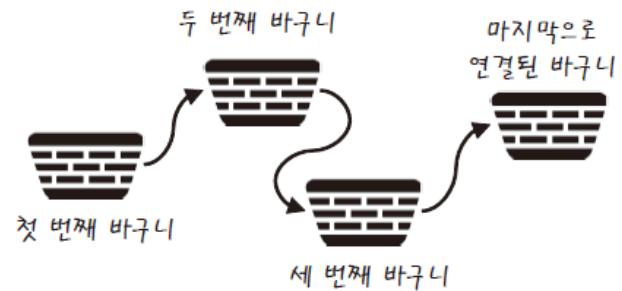
연결 리스트의 개념적인 이해



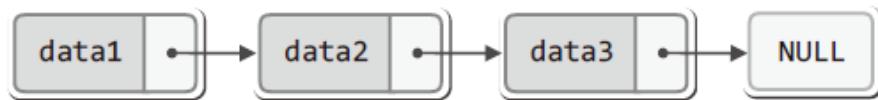
Linked! 무엇을 연결하겠다는 뜻인가!

```
typedef struct _node
{
    int data;      // 데이터를 담을 공간
    struct _node * next;    // 연결의 도구!
} Node;
```

일종의 바구니, 연결이 가능한 바구니



▶ [그림 04-1: 노드의 표현]



▶ [그림 04-2: 노드의 연결]

예제 *LinkedRead.c*의 분석을 시도 바랍니다!



예제 LinkedRead.c의 분석: 초기화

```
typedef struct _node
{
    int data;
    struct _node * next;
} Node;

int main(void)
{
    Node * head = NULL;
    Node * tail = NULL;
    Node * cur = NULL;

    Node * newNode = NULL;
    int readData;
    ....
}
```

LinkedRead.c의 일부

- head, tail, cur이 연결 리스트의 핵심!
- head와 tail은 연결을 추가 및 유지하기 위한것
- cur은 참조 및 조회를 위한것



예제 LinkedRead.c의 분석: 삽입 1회전

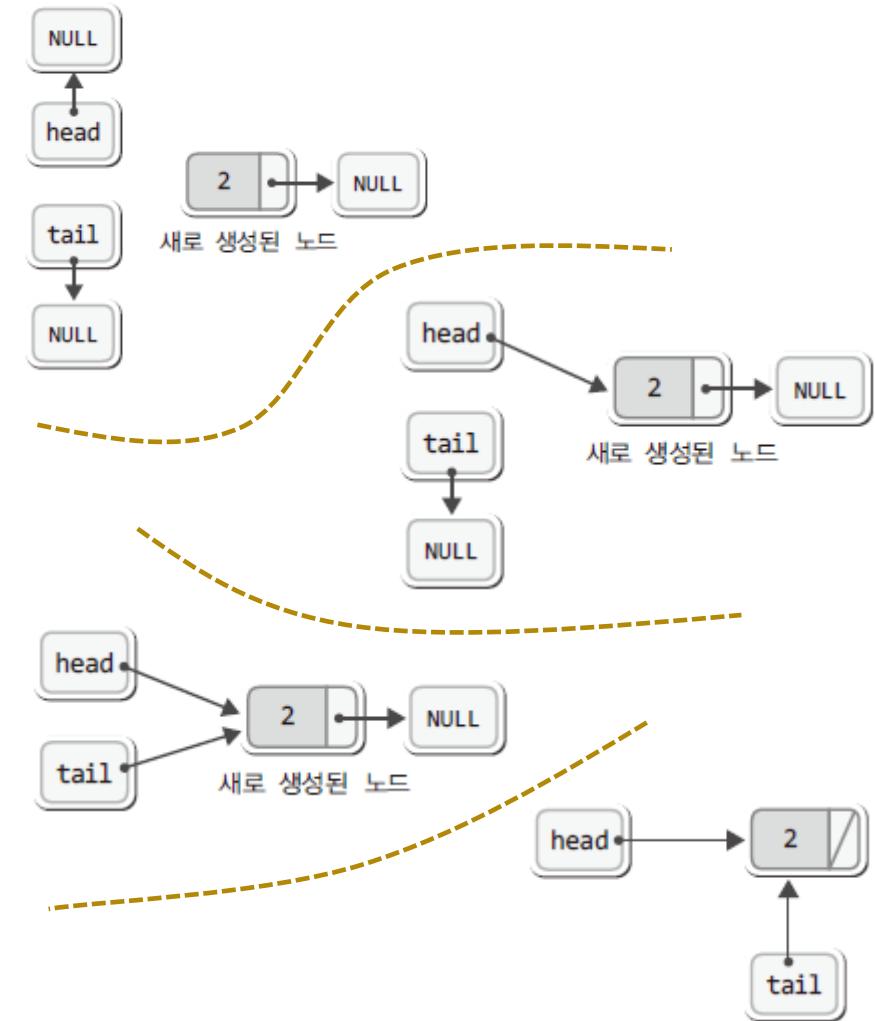
```
while(1)
{
    printf("자연수 입력: ");
    scanf("%d", &readData);
    if(readData < 1)
        break;

    // 노드의 추가과정
    newNode = (Node*)malloc(sizeof(Node));
    newNode->data = readData;
    newNode->next = NULL;

    if(head == NULL)
        head = newNode;
    else
        tail->next = newNode;

    tail = newNode;
}
```

LinkedRead.c의 일부



예제 LinkedRead.c의 분석: 삽입 2회전

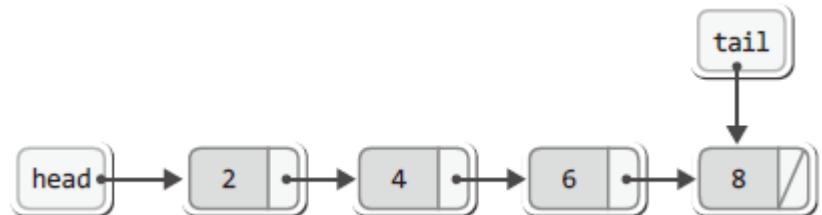
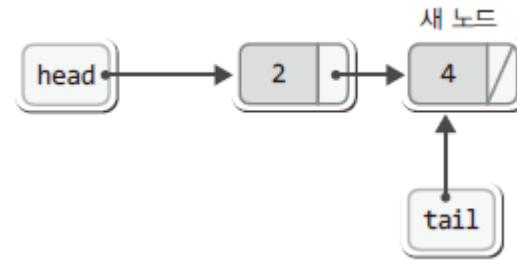
```
while(1)
{
    printf("자연수 입력: ");
    scanf("%d", &readData);
    if(readData < 1)
        break;

    // 노드의 추가과정
    newNode = (Node*)malloc(sizeof(Node));
    newNode->data = readData;
    newNode->next = NULL;

    if(head == NULL)
        head = newNode;
    else
        tail->next = newNode;

    tail = newNode;
}
```

LinkedRead.c의 일부



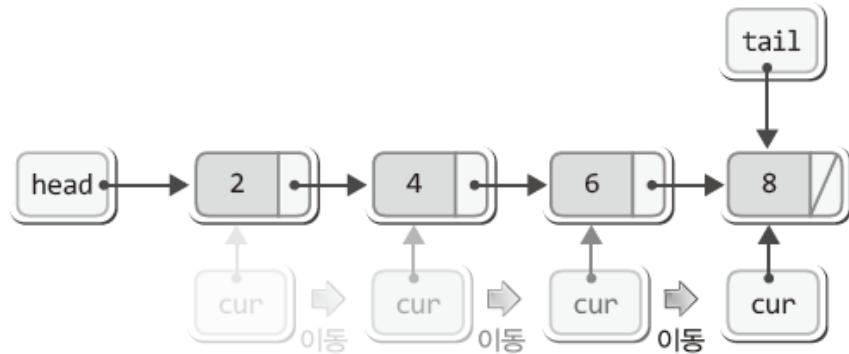
다수의 노드를 저장한 결과



예제 LinkedRead.c의 분석: 데이터 조회

전체 데이터의 출력 과정

```
if(head == NULL)
{
    printf("저장된 자연수가 존재하지 않습니다. \n");
}
else
{
    cur = head;
    printf("%d ", cur->data);
    while(cur->next != NULL)
    {
        cur = cur->next;
        printf("%d ", cur->data);
    }
}
```



LinkedRead.c의 일부

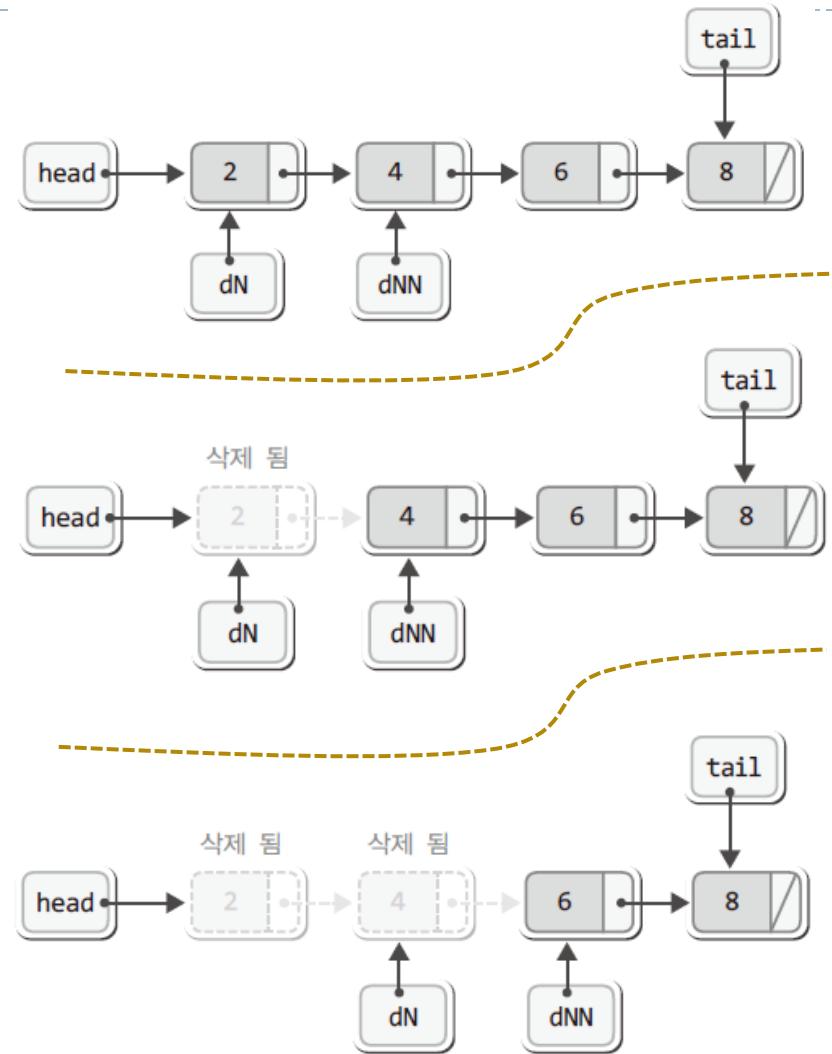


예제 LinkedRead.c의 분석: 데이터 삭제

```
if(head == NULL)    전체 노드의 삭제 과정
{
    return 0;
}
else
{
    Node * delNode = head;
    Node * delNextNode = head->next;
    printf("%d을 삭제\n", head->data);
    free(delNode);

    while(delNextNode != NULL)
    {
        delNode = delNextNode;
        delNextNode = delNextNode->next;
        printf("%d을 삭제\n", delNode->data);
        free(delNode);
    }
}
```

LinkedRead.c의 일부



Chapter 04. 연결 리스트(Linked List) 2



Chapter 04-2:

단순 연결 리스트의 ADT와 구현



정렬 기능 추가된 연결 리스트의 ADT1

- void ListInit(List * plist);
 - 초기화할 리스트의 주소 값을 인자로 전달한다.
 - 리스트 생성 후 제일 먼저 호출되어야 하는 함수이다.

이전과 동일

- void LInsert(List * plist, LData data);
 - 리스트에 데이터를 저장한다. 매개변수 data에 전달된 값을 저장한다.

이전과 동일

- int LFirst(List * plist, LData * pdata);
 - 첫 번째 데이터가 pdata가 가리키는 메모리에 저장된다.
 - 데이터의 참조를 위한 초기화가 진행된다.
 - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

이전과 동일

- int LNext(List * plist, LData * pdata);
 - 참조된 데이터의 다음 데이터가 pdata가 가리키는 메모리에 저장된다.
 - 순차적인 참조를 위해서 반복 호출이 가능하다.
 - 참조를 새로 시작하려면 먼저 LFirst 함수를 호출해야 한다.
 - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

이전과 동일



정렬 기능 추가된 연결 리스트의 ADT2

- LData LRemove(List * plist);

- LFirst 또는 LNext 함수의 마지막 반환 데이터를 삭제한다.
- 삭제된 데이터는 반환된다.
- 마지막 반환 데이터를 삭제하므로 연이은 반복 호출을 허용하지 않는다.

이전과 동일

- int LCount(List * plist);

- 리스트에 저장되어 있는 데이터의 수를 반환한다.

이전과 동일

- void SetSortRule(List * plist, int (*comp)(LData d1, LData d2));

- 리스트에 정렬의 기준이 되는 함수를 등록한다.

새로 추가된 함수

SetSortRule 함수는 정렬의 기준을 설정하기 위해 정의된 함수! 이 함수의 선언 및 정의를 이해하기 위해서는 '함수 포인터'의 대한 이해가 필요하다.



새 노드의 추가 위치에 따른 장점과 단점

새 노드를 연결 리스트의 머리에 추가하는 경우

- 장점 포인터 변수 tail이 불필요하다.
- 단점 저장된 순서를 유지하지 않는다.

새 노드를 연결 리스트의 꼬리에 추가하는 경우

- 장점 저장된 순서가 유지된다.
- 단점 포인터 변수 tail이 필요하다.

두 가지 다 가능한 방법이다. 다만 tail의 관리를 생략하기 위해서 머리에 추가하는 것을 원칙으로 하자!



SetSortRule 함수 선언에 대한 이해

```
void SetSortRule ( List * plist, int (*comp)(LData d1, LData d2) );
```

- ✓ 반환형이 int이고, int (*comp)(LData d1, LData d2)
- ✓ LData형 인자를 두 개 전달받는, int (*comp)(LData d1, LData d2)
- ✓ 함수의 주소 값을 전달해라! int (*comp)(LData d1, LData d2)

```
int WhoIsPrecede(LData d1, LData d2) // typedef int LData;  
{  
    if(d1 < d2)  
        return 0;                                  // d1이 정렬 순서상 앞선다.  
    else  
        return 1;                                  // d2가 정렬 순서상 앞서거나 같다.  
}
```

인자로 전달이 가능한
함수의 예



정렬의 기준을 결정하는 함수에 대한 약속!

```
int WhoIsPrecede(LData d1, LData d2)
{
    if(d1 < d2)           // d1이 정렬 순서상 앞선다.
        return 0;
    else                  // d2가 정렬 순서상 앞서거나 같다.
        return 1;
}
```

이렇듯 결정된 약속을 근거로 함수가 정의되어야 하며, 연결 리스트 또한 이를 근거로 구현되어야 한다.

약속에 근거한 함수 정의의 예

```
int cr = WhoIsPrecede(D1, D2);
```

Cr에 저장된 값이 0이라면

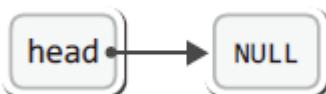
head . . . D1 . . D2 . . . tail D1이 head에 더 가깝다.

Cr에 저장된 값이 1이라면

head . . . D2 . . D1 . . . tail D2가 head에 더 가깝다.



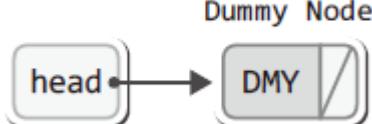
우리가 구현할 더미 노드 기반 연결 리스트



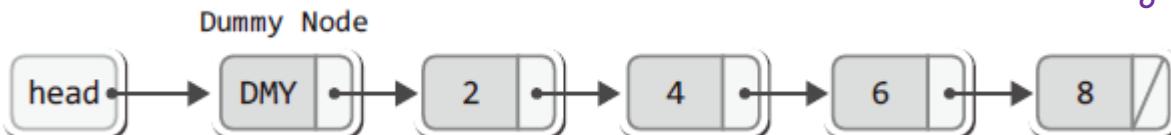
머리에 새 노드를 추가하되 더미 노드 없는 경우



첫 번째 노드와 두 번째 이후의 노드
추가 및 삭제 방식이 다를 수 있다.



머리에 새 노드를 추가하되 더미 노드 있는 경우



노드의 추가 및 삭제 방식이 향
상 일정하다.

LinkedRead.c와 문제 04-2의 답안인 DLinkedRead.c를 비교해보자!



정렬 기능 추가된 연결 리스트의 구조체

```
typedef struct _node
{
    LData data;          // typedef int LData
    struct _node * next;
} Node;
```

노드의 구조체 표현

연결 리스트에 필요한 변수들을 구조체로 묶지 않는 것은 옳지 못하다.

연결 리스트의 구조체 표현

```
typedef struct _linkedList
{
    Node * head;           // 더미 노드를 가리키는 멤버
    Node * cur;            // 참조 및 삭제를 돋는 멤버
    Node * before;         // 삭제를 돋는 멤버
    int numOfData;         // 저장된 데이터의 수를 기록하기 위한 멤버
    int (*comp)(LData d1, LData d2); // 정렬의 기준을 등록하기 위한 멤버
} LinkedList;
```



정렬 기능 추가된 연결 리스트 헤더파일

```
#define TRUE 1
#define FALSE 0

typedef int LData;
typedef struct _node
{
    LData data;
    struct _node * next;
} Node;

typedef struct _linkedList
{
    Node * head;
    Node * cur;
    Node * before;
    int numOfData;
    int (*comp)(LData d1, LData d2);
} LinkedList;
```

```
typedef LinkedList List;

void ListInit(List * plist);
void LInsert(List * plist, LData data);

int LFirst(List * plist, LData * pdata);
int LNext(List * plist, LData * pdata);
LData LRemove(List * plist);
int LCount(List * plist);
void SetSortRule(List * plist, int (*comp)(LData d1, LData d2));
```

뒷부분

앞부분



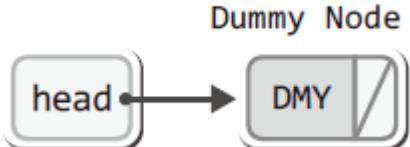
더미 노드 연결 리스트 구현: 초기화

```
typedef struct _linkedList
{
    Node * head;
    Node * cur;
    Node * before;
    int numOfData;
    int (*comp)(LData d1, LData d2);
} LinkedList;
```

초기화 함수의 정의를 위해서
살펴봐야 하는 구조체의 정의

초기화 함수의 정의

```
void ListInit(List * plist)
{
    plist->head = (Node*)malloc(sizeof(Node)); // 더미 노드의 생성
    plist->head->next = NULL;
    plist->comp = NULL;
    plist->numOfData = 0;
}
```



더미 노드 연결 리스트 구현: 삽입1

```
void LInsert(List * plist, LData data)
{
    if(plist->comp == NULL)          // 정렬기준이 마련되지 않았다면,
        FInsert(plist, data);       // 머리에 노드를 추가!

    else                            // 정렬기준이 마련되었다면,
        SInsert(plist, data);     // 정렬기준에 근거하여 노드를 추가!
}
```

```
void FInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node)); // 새 노드 생성
    newNode->data = data;                      // 새 노드에 데이터 저장

    newNode->next = plist->head->next;        // 새 노드가 다른 노드를 가리키게 함
    plist->head->next = newNode;                // 더미 노드가 새 노드를 가리키게 함

    (plist->numOfData)++;
}
```



더미 노드 연결 리스트 구현: 삽입2

```
void FInsert(List * plist, LData data)
```

```
{
```

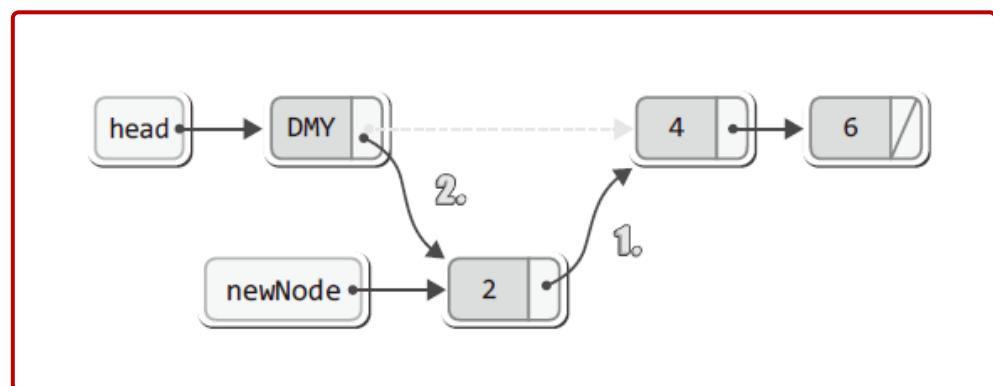
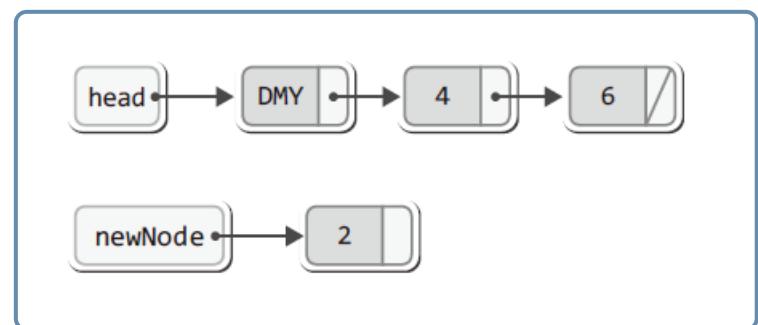
```
    Node * newNode = (Node*)malloc(sizeof(Node));  
    newNode->data = data;
```

```
    newNode->next = plist->head->next;  
    plist->head->next = newNode;
```

```
(plist->numOfData)++;
```

```
}
```

모든 경우에 있어서 동일한 삽입과정을
거친다는 것이 더미 노드 기반 연결
리스트의 장점!

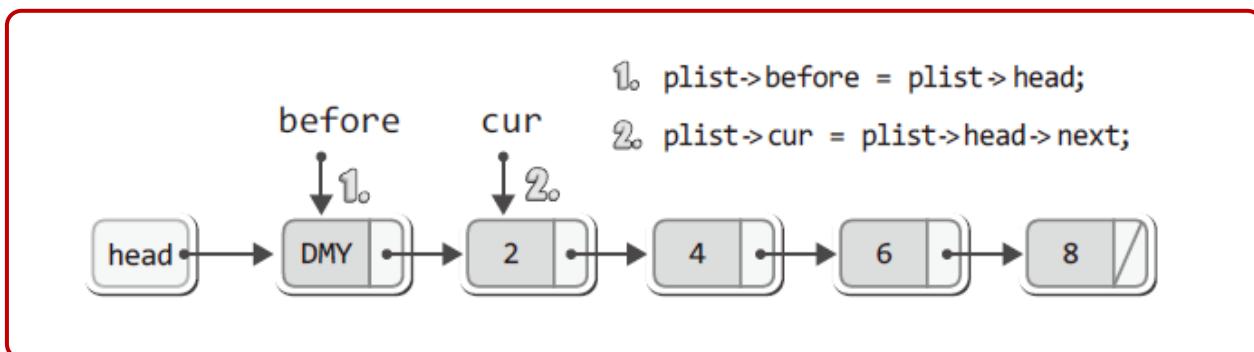


더미 노드 연결 리스트 구현: 참조1

```
int LFirst(List * plist, LData * pdata)
{
    if(plist->head->next == NULL)          // 더미 노드가 NULL을 가리킨다면,
        return FALSE;                      // 반환할 데이터가 없다!

    plist->before = plist->head;           // before는 더미 노드를 가리키게 함
    plist->cur = plist->head->next;         // cur은 첫 번째 노드를 가리키게 함

    *pdata = plist->cur->data;            // 첫 번째 노드의 데이터를 전달
    return TRUE;                          // 데이터 반환 성공!
}
```

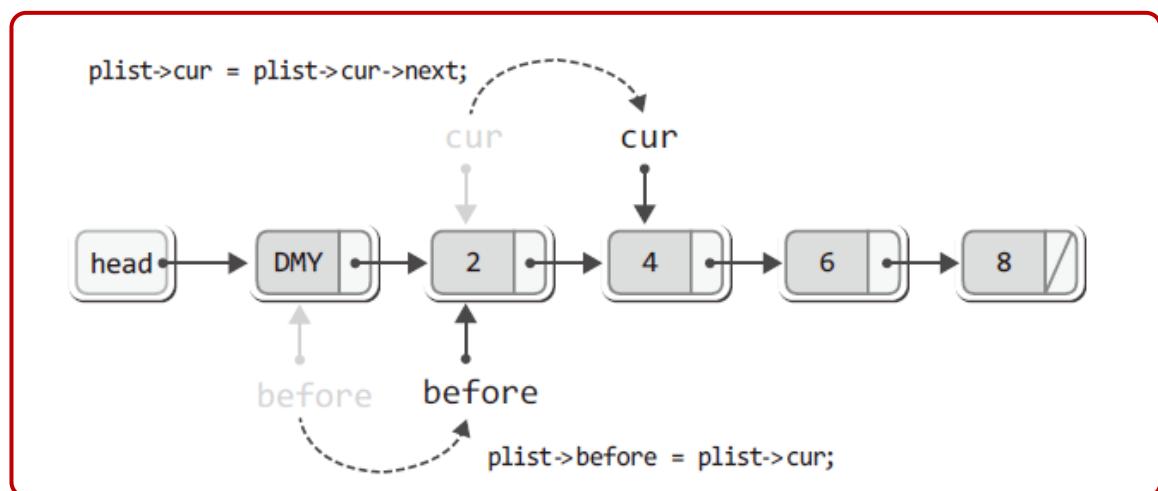


더미 노드 연결 리스트 구현: 참조2

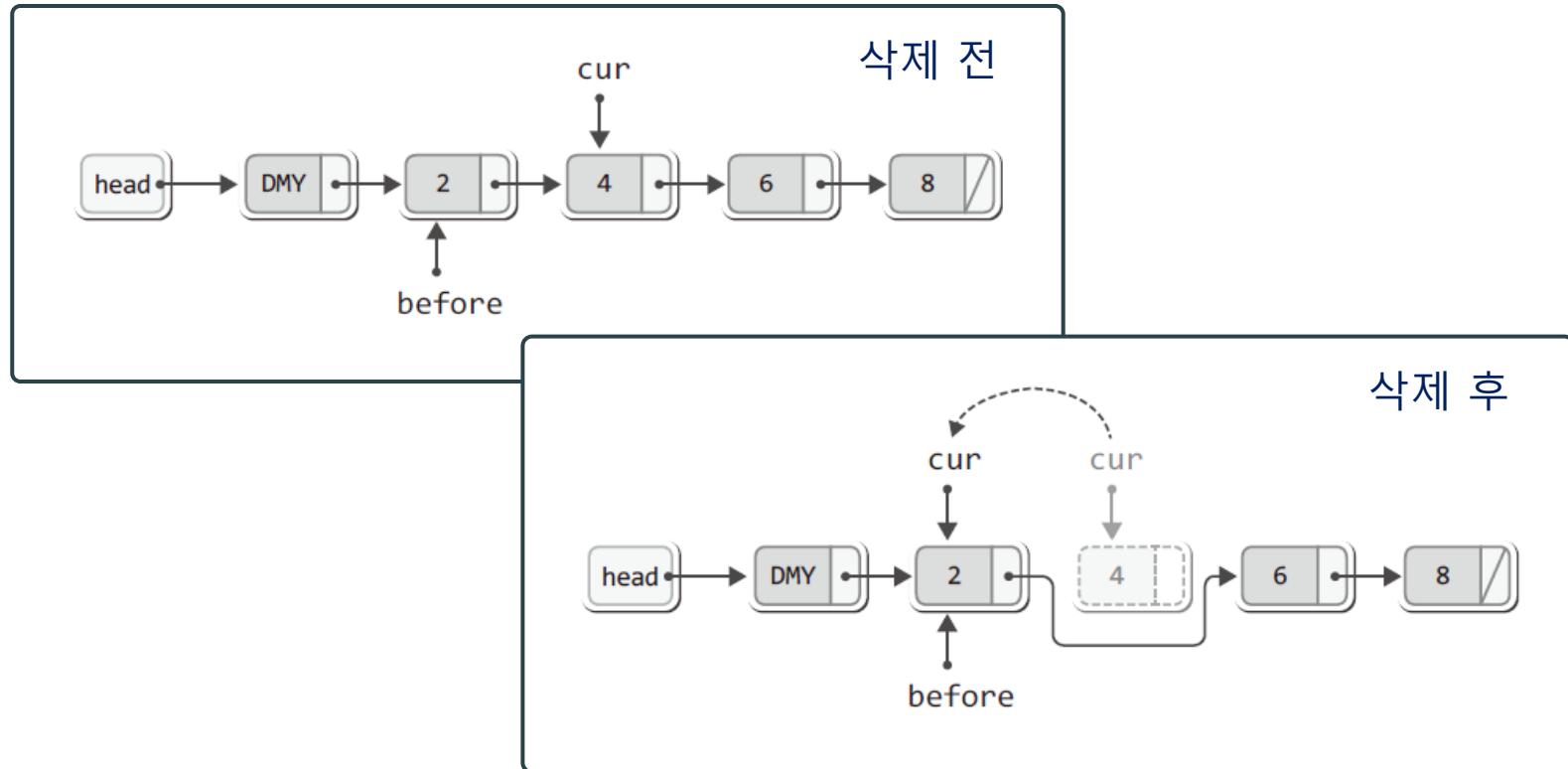
```
int LNext(List * plist, LData * pdata)
{
    if(plist->cur->next == NULL)          // 더미 노드가 NULL을 가리킨다면,
        return FALSE;                      // 반환할 데이터가 없다!

    plist->before = plist->cur;           // cur이 가리키던 것을 before가 가리킴
    plist->cur = plist->cur->next;         // cur은 그 다음 노드를 가리킴

    *pdata = plist->cur->data;            // cur이 가리키는 노드의 데이터 전달
    return TRUE;                          // 데이터 반환 성공!
}
```



더미 노드 연결 리스트 구현: 삭제1



`cur`은 삭제 후 제조정의 과정을 거쳐야 하지만 `before`는 LFirst or LNext 호출 시 재설정되므로 제조정의 과정이 불필요하다.

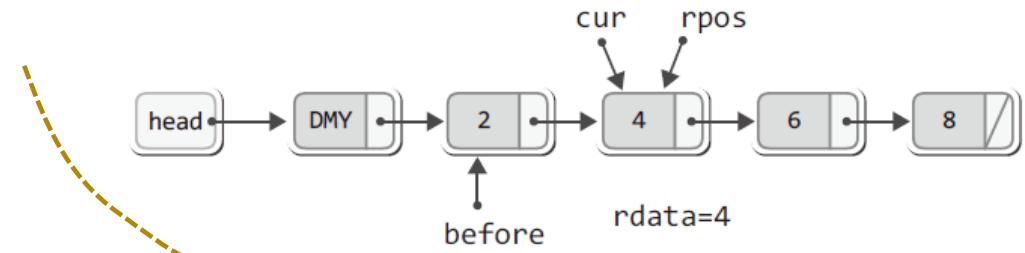


더미 노드 연결 리스트 구현: 삭제2

```
LData LRemove(List * plist)
```

```
{
```

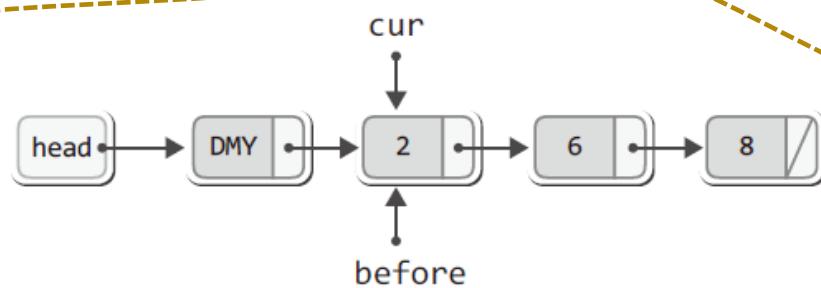
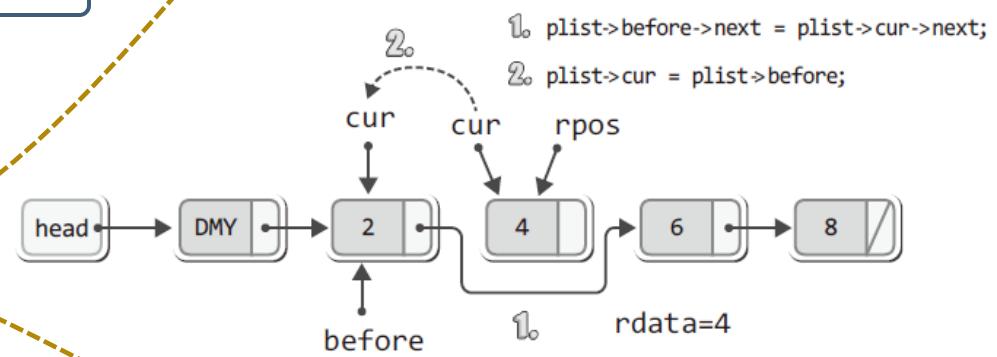
```
    Node * rpos = plist->cur;  
    LData rdata = rpos->data;
```



```
    plist->before->next = plist->cur->next;  
    plist->cur = plist->before;
```

```
}
```

```
    free(rpos);  
    (plist->numOfData)--;  
    return rdata;
```



더미 기반 단순 연결 리스트 한데 묶기

DLinkedList.c
DLinkedList.h
DLinkedListMain.c

실행결과

```
현재 데이터의 수: 5  
33 22 22 11 11
```

```
현재 데이터의 수: 3  
33 11 11
```

Chapter 03의 ListMain.c의 main 함수와 완전히 동일하다.

다만 노드를 머리에 추가하는 방식이기 때문에 실행결과에서는 차이가 난다.



Chapter 04. 연결 리스트(Linked List) 2



Chapter 04-3:

연결 리스트의 정렬 삽입의 구현



정렬기준 설정과 관련된 부분

단순 연결 리스트의 정렬 관련 요소 세 가지

- 정렬기준이 되는 함수를 등록하는 **SetSortRule** 함수
- **SetSortRule** 함수 통해 전달된 함수정보 저장을 위한 **LinkedList**의 멤버 **comp**
- **comp**에 등록된 정렬기준을 근거로 데이터를 저장하는 **SInsert** 함수



하나의 문장으로 구성한 결과

“**SetSortRule** 함수가 호출되면서 정렬의 기준이 리스트의 멤버 **comp**에 등록되면, **SInsert** 함수 내에서는 **comp**에 등록된 정렬의 기준을 근거로 데이터를 정렬하여 저장한다.”



SetSortRule 함수와 멤버 comp

1. SetSortRule 함수의 호출을 통해서 . . .

```
void SetSortRule(List * plist, int (*comp)(LData d1, LData d2))
{
    plist->comp = comp;
}
```

```
typedef struct _linkedList
{
    Node * head;
    Node * cur;
    Node * before;
    int numOfData;
    int (*comp)(LData d1, LData d2);
} LinkedList;
```

2. 멤버 comp가 초기화되면 . . .

```
void LInsert(List * plist, LData data)
{
    if(plist->comp == NULL)
        FInsert(plist, data);
    else
        SInsert(plist, data);
}
```

3. 정렬 관련 SInsert 함수가 호출된다.



SInsert 함수1

```
void SInsert(List * plist, LData data)
```

```
{  
    Node * newNode = (Node*)malloc(sizeof(Node));  
    Node * pred = plist->head;  
    newNode->data = data;
```

// 새 노드가 들어갈 위치를 찾기 위한 반복문!

```
while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)  
{
```

```
    pred = pred->next; // 다음 노드로 이동
```

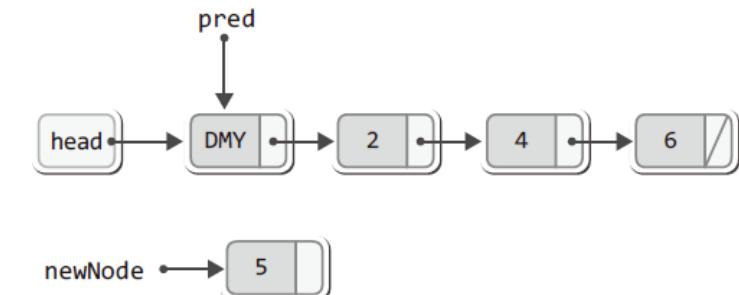
```
}
```

```
newNode->next = pred->next;
```

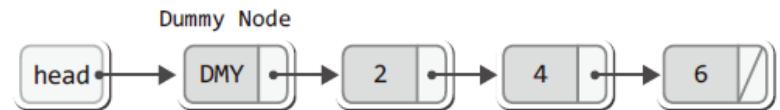
```
pred->next = newNode;
```

```
(plist->numOfData)++;
```

```
}
```



▶ [그림 04-31: SInsert 함수에서의 초기화]



▶ [그림 04-30: 값의 대소가 정렬의 기준인 연결 리스트]

위 상황에서 다음 문장이 실행되었다고 가정!

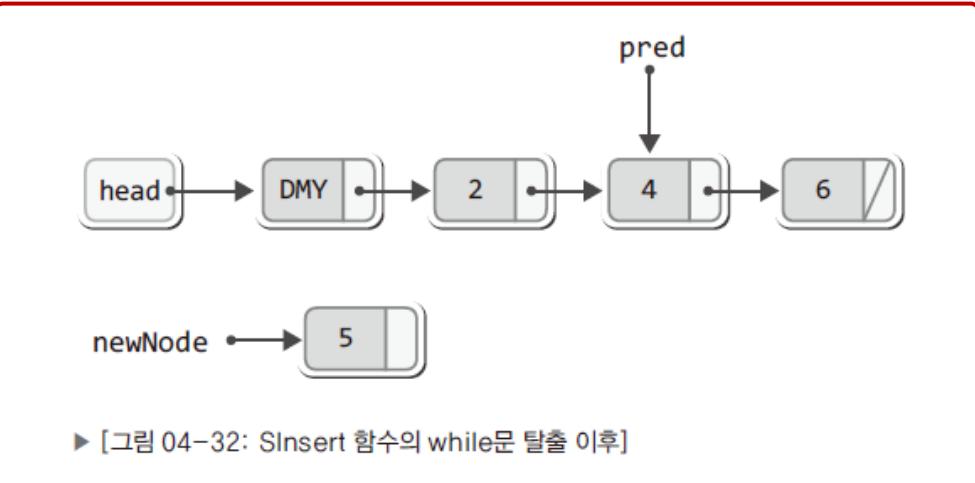
```
SInsert(&slist, 5);
```

SInsert 함수2

```
void SInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    Node * pred = plist->head;
    newNode->data = data;

    // 새 노드가 들어갈 위치를 찾기 위한 반복문!
    while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
    {
        pred = pred->next; // 다음 노드로 이동
    }
    newNode->next = pred->next;
    pred->next = newNode;
    (plist->numOfData)++;
}
```

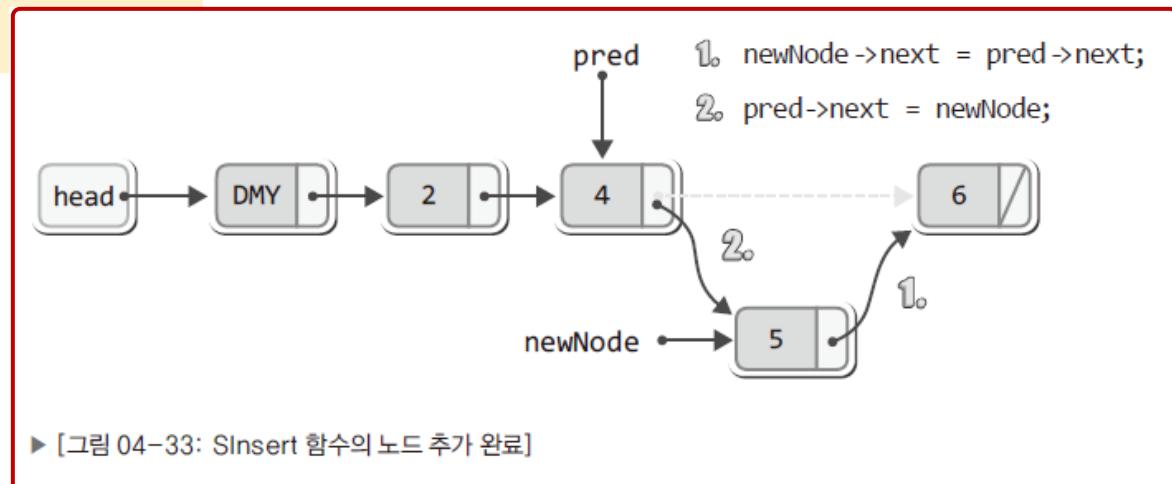
comp가 0을 반환한다는 것은 첫 번째 인자인 data가 정렬 순서상 앞서기 때문에 head에 가까워야 한다는 의미!



SInsert 함수3

```
void SInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    Node * pred = plist->head;
    newNode->data = data;

    // 새 노드가 들어갈 위치를 찾기 위한 반복문!
    while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
    {
        pred = pred->next; // 다음 노드로 이동
    }
    newNode->next = pred->next;
    pred->next = newNode;
    (plist->numOfData)++;
}
```



정렬의 핵심인 while 반복문

- 반복의 조건 1 pred->next != NULL
pred가 리스트의 마지막 노드를 가리키는지 묻기 위한 연산
- 반복의 조건 2 plist->comp(data, pred->next->data) != 0
새 데이터와 pred의 다음 노드에 저장된 데이터의 우선순위 비교를 위한 함수호출

```
while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
{
    pred = pred->next;           // 다음 노드로 이동
}
```



comp의 반환 값과 그 의미

```
while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
{
    pred = pred->next;          // 다음 노드로 이동
}
```

우리의 결정 내용! 이 내용을 근거로 *SInsert* 함수를 정의하였다.

- comp가 0을 반환

첫 번째 인자인 data가 정렬 순서상 앞서서 head에 더 가까워야 하는 경우

- comp가 1을 반환

두 번째 인자인 pred->next->data가 정렬 순서상 앞서서 head에 더 가까워야 하는 경우



정렬의 기준을 설정하기 위한 함수의 정의

- 두 개의 인자를 전달받도록 함수를 정의한다.
- 첫 번째 인자의 정렬 우선순위가 높으면 0을, 그렇지 않으면 1을 반환한다.

함수의 정의 기준

```
int WhoIsPrecede(int d1, int d2)
{
    if(d1 < d2)
        return 0;      // d1이 정렬 순서상 앞선다.
    else
        return 1;      // d2가 정렬 순서상 앞서거나 같다.
}
```

오름차순 정렬을 위한 함수의 정의

정렬 관련된 함수를 *DLinkedListSortMain.c*에 포함시켜야 함을 이해한다.

DLinkedList.c

DLinkedList.h

DLinkedListSortMain.c

현재 데이터의 수: 5

11 11 22 22 33

현재 데이터의 수: 3

11 11 33

실행결과



수고하셨습니다~



Chapter 04에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 05. 연결 리스트(Linked List) 3

Introduction To Data Structures Using C

Chapter 05. 연결 리스트(Linked List) 3



Chapter 05-1:

원형 연결 리스트

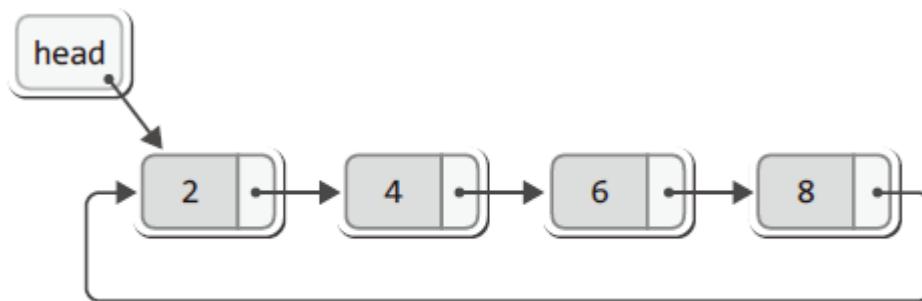


원형 연결 리스트의 이해



▶ [그림 05-1: 단순 연결 리스트]

단순 연결 리스트의 마지막 노드는 NULL을 가리킨다.

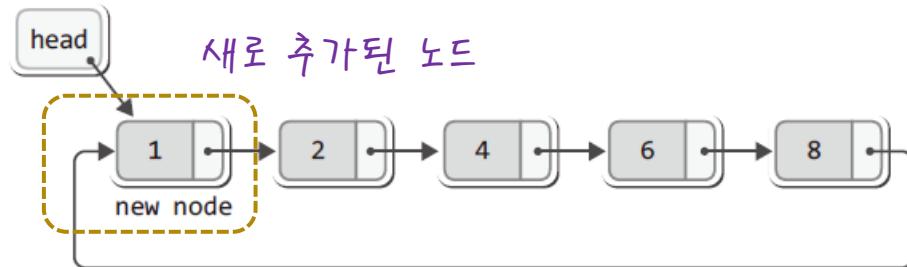


▶ [그림 05-2: 원형 연결 리스트]

원형 연결 리스트의 마지막 노드는 첫 번째 노드를 가리킨다.

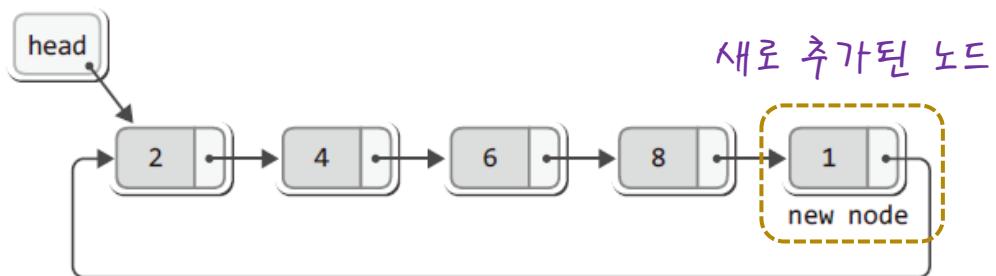


원형 연결 리스트의 노드 추가



▶ [그림 05-3: 원형 연결 리스트의 머리에 노드 추가]

이렇듯 모든 노드가 원의 형태를 이루면서 연결되어 있기 때문에 원형 연결 리스트에서는 사실상 머리와 꼬리의 구분이 없다.



▶ [그림 05-4: 원형 연결 리스트의 꼬리에 노드 추가]

같고 또 다른 점!

- 같다
- 다르다

두 경우의 노드 연결 순서가 같다.
head가 가리키는 노드가 다르다.

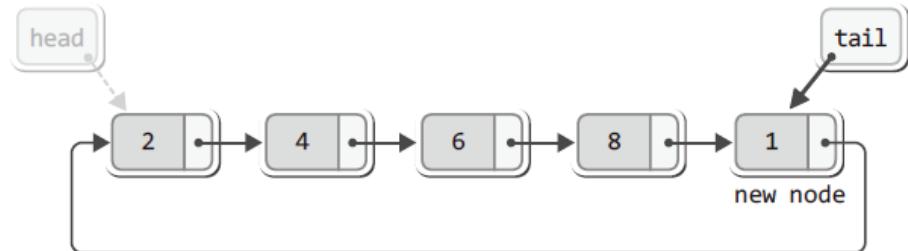


원형 연결 리스트의 대표적인 장점

원형 연결 리스트에서 놓칠 수 없는 장점

“단순 연결 리스트처럼 머리와 꼬리를 가리키는 포인터 변수를 각각 두지 않아도, 하나의 포인터 변수만 있어도 머리 또는 꼬리에 노드를 간단히 추가할 수 있다.”

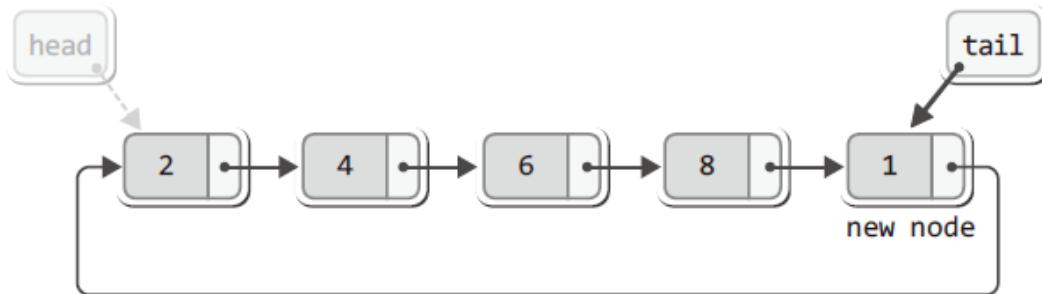
앞서 소개한 모델을 기반으로는 위의 장점을 살리기 어렵다. 그래서 우리는 변형된, 그러나 보다 일반적이라고 인식이 되고 있는 **변경된 원형 연결 리스트**를 구현한다.



▶ [그림 05-6: 변경된 원형 연결 리스트]



변형된 원형 연결 리스트



▶ [그림 05-6: 변형된 원형 연결 리스트]

- **꼬리**를 가리키는 포인터 변수는? **tail** 입니다!
- **머리**를 가리키는 포인터 변수는? **tail->next** 입니다!

이렇듯 리스트의 꼬리와 머리의 주소 값을 쉽게 확인할 수 있기 때문에 연결 리스트를 가리키는 포인터 변수는 하나만 있으면 된다.



변형된 원형 연결 리스트의 구현 범위

원형 연결 리스트는 그 구조상 끝이 존재하지 않는다. 따라서 `LNext` 함수는 계속해서 호출이 가능하고, 이로 인해서 리스트를 순환하면서 저장된 값을 반환하도록 구현한다.



원형 연결 리스트의 헤더파일과 초기화 함수

```
typedef int Data;  
  
typedef struct _node  
{  
    Data data;  
    struct _node * next;  
} Node;  
  
typedef struct _CLL  
{  
    Node * tail;  
    Node * cur;  
    Node * before;  
    int numOfData;  
} CLList;
```

```
typedef CLList List;  
  
void ListInit(List * plist);  
void LInsert(List * plist, Data data); 노드를 꼬리에 추가!  
void LInsertFront(List * plist, Data data); 노드를 머리에 추가!  
  
int LFirst(List * plist, Data * pdata);  
int LNext(List * plist, Data * pdata);  
Data LRemove(List * plist);  
int LCount(List * plist);
```

```
void ListInit(List * plist)  
{  
    plist->tail = NULL;  
    plist->cur = NULL;  
    plist->before = NULL;  
    plist->numOfData = 0;  
}
```

모든 멤버를 NULL과 0으로
초기화한다.

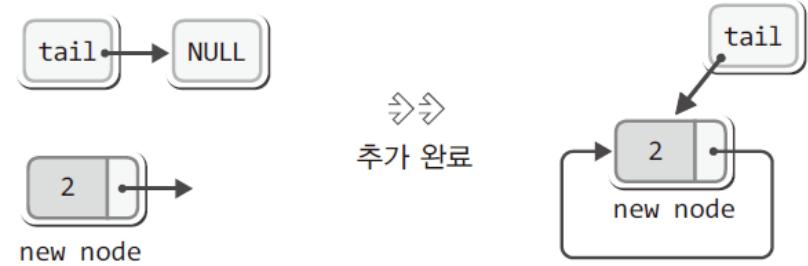


원형 연결 리스트 구현: 첫 번째 노드 삽입

```
// LInsert & LInsertFront의 공통 부분
void LInser~(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    첫 번째 노드라면

    if(plist->tail == NULL)
    {
        plist->tail = newNode;
        newNode->next = newNode;
    }
    else
        두 번째 이후의 노드라면
        . . . . 차이가 나는 부분 . . .
    }

    (plist->numOfData)++;
}
```



첫 번째 노드는 그 자체로 머리이자 꼬리이기 때문에 노드를 뒤에 추가하건 앞에 추가하건 그 결과가 동일하다!



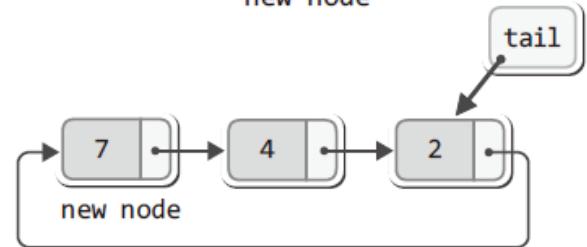
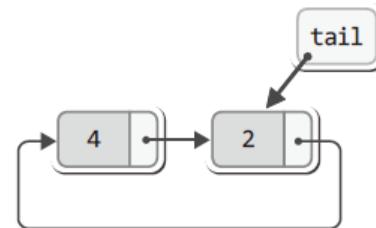
원형 연결 리스트 구현: 두 번째 이후 노드 머리로 삽입

두 번째 이후의 노드 머리에 추가!

```
void LInsertFront(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

    if(plist->tail == NULL)
    {
        plist->tail = newNode;
        newNode->next = newNode;
    }
    else
    {
        newNode->next = plist->tail->next;
        plist->tail->next = newNode;
    }

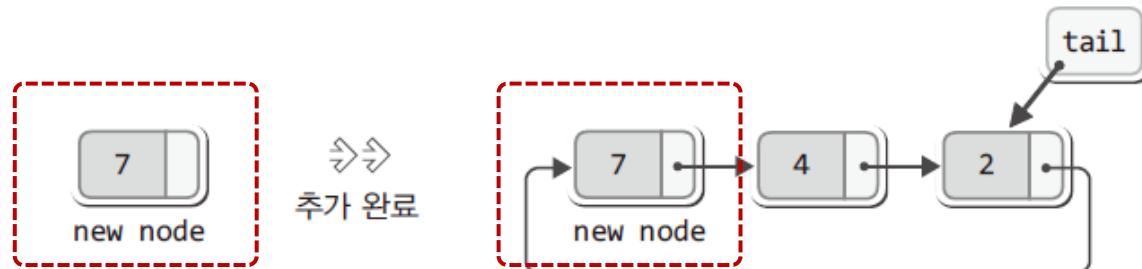
    (plist->numOfData)++;
}
```



두 번째 이후 노드를 머리에 추가

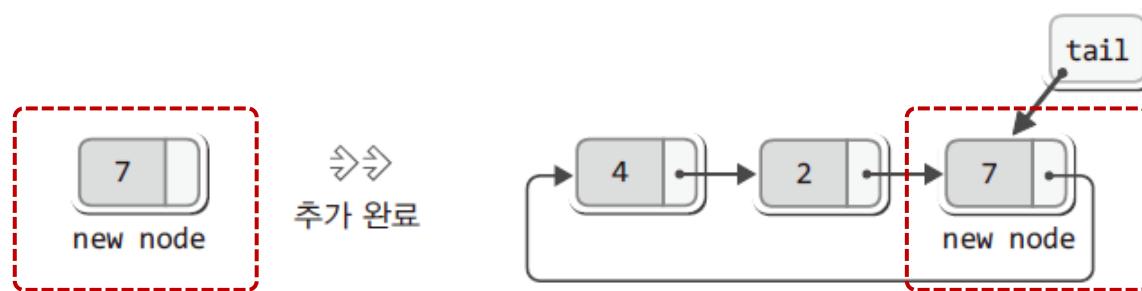


원형 연결 리스트 구현: 앞과 뒤의 삽입 과정 비교1



노드를 머리에 추가한 결과

그림상에서 보았을 때 이 둘의
실질적인 차이점은?



노드를 꼬리에 추가한 결과



원형 연결 리스트 구현: 앞과 뒤의 삽입 과정 비교2

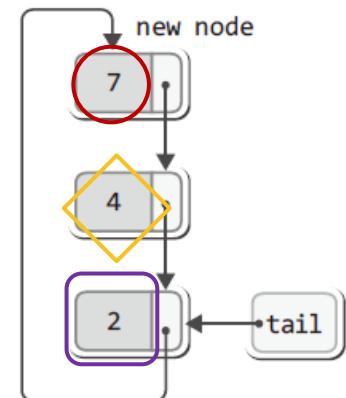
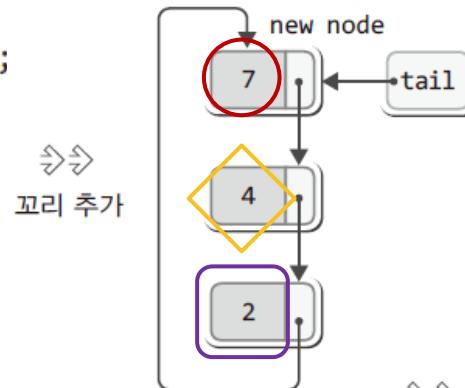
두 번째 이후의 노드 꼬리에 추가!

```
void LInsert(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

    if(plist->tail == NULL)
    {
        plist->tail = newNode;
        newNode->next = newNode;
    }
    else
    {
        newNode->next = plist->tail->next;
        plist->tail->next = newNode;
        plist->tail = newNode;
    }
}

(plist->numOfData)++;
}
```

tail의 위치가 유일한 차이점이다!



LInsertFront 함수에는 없는 문장!

LInsertFront 함수와의 유일한 차이점!

원형 연결 리스트 구현: 조회 LFirst

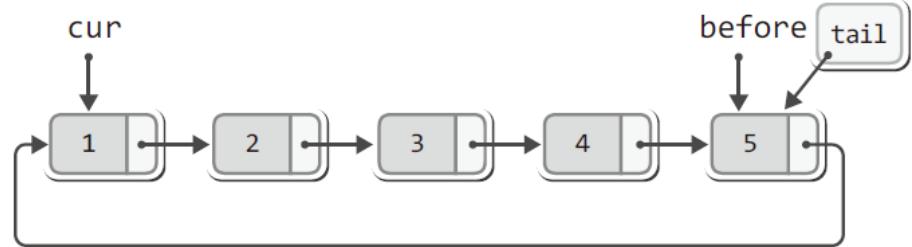
```
int LFirst(List * plist, Data * pdata)
{
    if(plist->tail == NULL)
        return FALSE;

    plist->before = plist->tail;
    plist->cur = plist->tail->next;

    *pdata = plist->cur->data;
    return TRUE;
}
```

```
typedef struct _CLL
{
    Node * tail;
    Node * cur;
    Node * before;
    int numOfData;
} CList;
```

cur이 가리키는 노드가 머리!



▶ [그림 05-12: LFirst 함수의 호출결과]



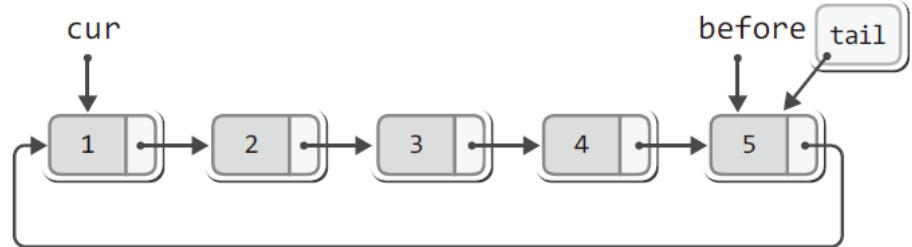
원형 연결 리스트 구현: 조회 LNext

```
int LNext(List * plist, Data * pdata)
{
    if(plist->tail == NULL)
        return FALSE;

    plist->before = plist->cur;
    plist->cur = plist->cur->next;

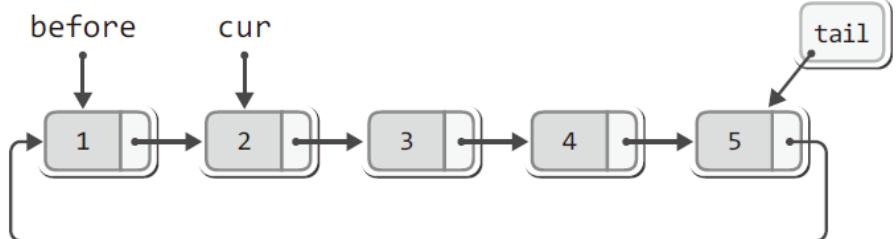
    *pdata = plist->cur->data;
    return TRUE;
}
```

원형 연결 리스트이므로 리스트의
끝을 검사하는 코드가 없다!



▶ [그림 05-12: LFirst 함수의 호출결과]

이어지는
LNext 호출결과



▶ [그림 05-13: LNext 함수의 호출결과]



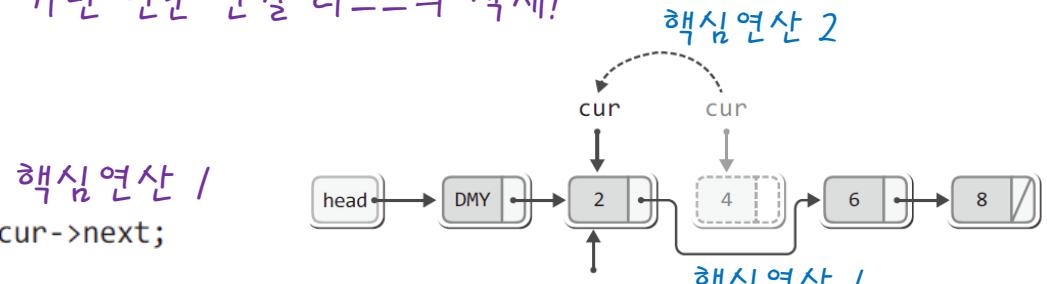
원형 연결 리스트 구현: 노드의 삭제(복습)

더미 노드 기반 연결 리스트의 삭제 과정 복습!

- 핵심연산 1.
삭제할 노드의 이전 노드가, 삭제할 노드의 다음 노드를 가리키게 한다.
- 핵심연산 2.
포인터 변수 cur을 한 칸 뒤로 이동시킨다.

LData LRemove(List * plist) 더미 기반 단순 연결 리스트의 삭제!

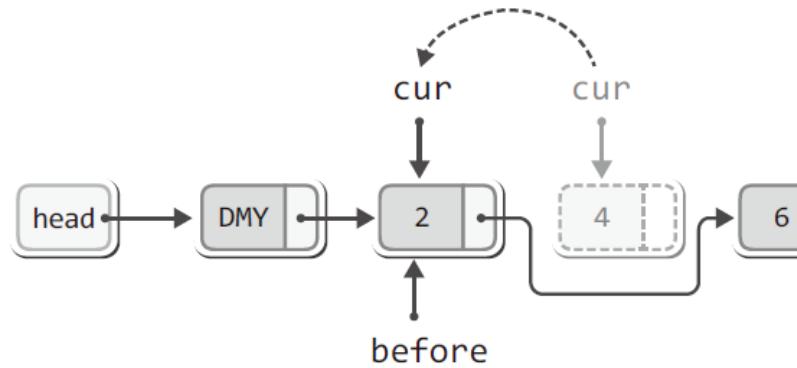
```
{  
    Node * rpos = plist->cur;  
    LData rdata = rpos->data;  
  
    plist->before->next = plist->cur->next;  
    plist->cur = plist->before;  
  
    free(rpos);  
    (plist->numOfData)--;  
    return rdata;  
}
```



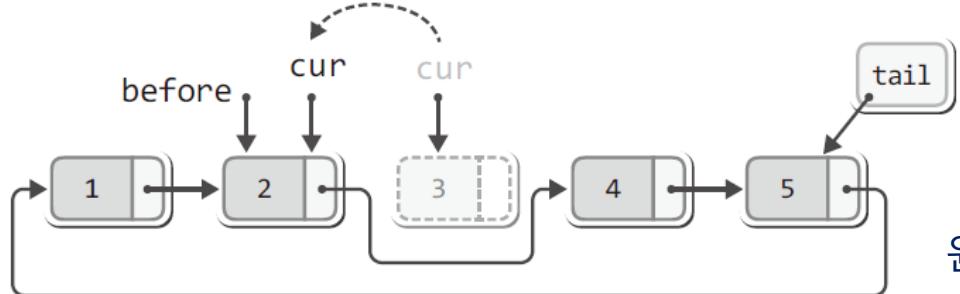
원형 연결 리스트의 삭제도 이와 별반 다르지 않다!



원형 연결 리스트 구현: 노드의 삭제(그림 비교)



더미 기반 단순 연결 리스트



원형 연결 리스트

그림상으로는 두 연결 리스트의 삭제 과정이 비슷해 보이나 원형 연결 리스트에는 더미 노드가 없기 때문에 삭제의 과정이 상황에 따라서 달라진다.



원형 연결 리스트 노드 삭제 구현

```
Data LRemove(List * plist)
{
    Node * rpos = plist->cur;
    Data rdata = rpos->data;

    if(rpos == plist->tail)
        {
            그리고 마지막 노드라면
            if(plist->tail == plist->tail->next)
                plist->tail = NULL;
            else
                plist->tail = plist->before;
        }

    plist->before->next = plist->cur->next;
    plist->cur = plist->before;

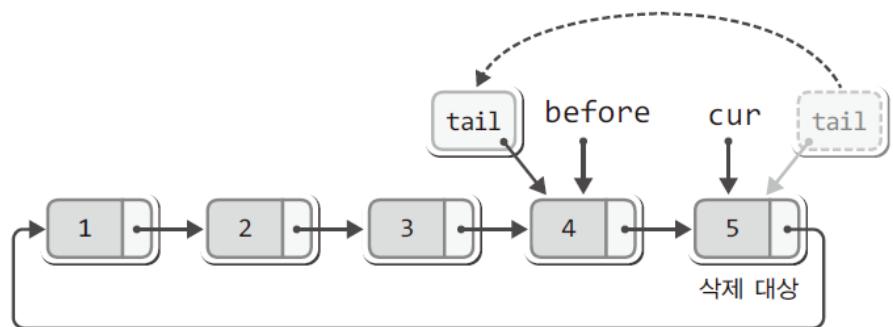
    free(rpos);
    단순 연결 리스트와 동일한 코드
    (plist->numOfData)--;
    return rdata;
}
```

• 예외적인 상황 1

삭제할 노드를 tail이 가리키는 경우

• 예외적인 상황 2

삭제할 노드를 tail이 가리키는데,
그 노드가 마지막 노드라면



▶ [그림 05-16: 삭제의 예외적인 경우]

예외적인 상황 1에 해당하는 경우



원형 연결 리스트 구현: 하나로 묶기

CLinkedList.c
CLinkedList.h
CLinkedListMain.c

실행결과

```
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5  
1 3 5
```



Chapter 05. 연결 리스트(Linked List) 3

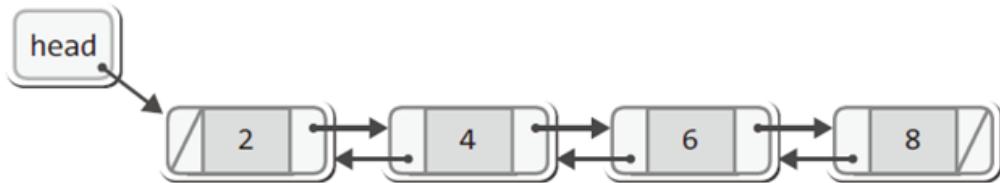


Chapter 05-2:

양방향 연결 리스트

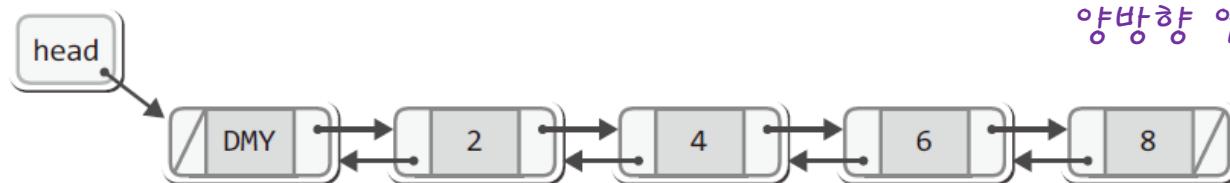


양방향 연결 리스트의 이해



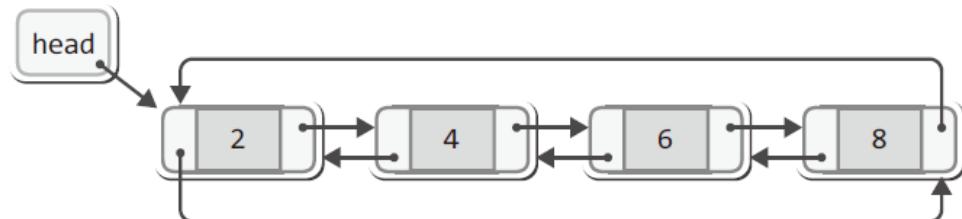
▶ [그림 05-17: 기본적인 양방향 연결 리스트]

```
typedef struct _node
{
    Data data;
    struct _node * next;
    struct _node * prev;
} Node;
```



양방향 연결 리스트를 위한 노드의 표현

▶ [그림 05-18: 더미 노드 양방향 연결 리스트]



▶ [그림 05-19: 원형 연결 기반의 양방향 연결 리스트]

양방향으로 노드를 연결하는 이유!

```
int LNext(List * plist, Data * pdata)
{
    if(plist->tail == NULL)
        return FALSE;

    plist->before = plist->cur;
    plist->cur = plist->cur->next;

    *pdata = plist->cur->data;
    return TRUE;
}
```

단순 연결 리스트의 *LNext*

```
int LNext(List * plist, Data * pdata)
{
    if(plist->cur->next == NULL)
        return FALSE;

    plist->cur = plist->cur->next;
    *pdata = plist->cur->data;

    return TRUE;
}
```

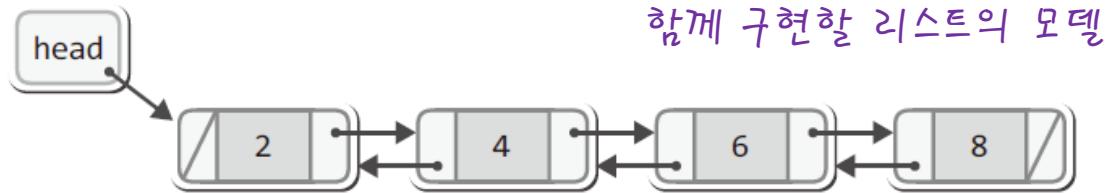
단순 연결 리스트와 같이
*before*를 유지할 필요가 없다!

양방향 연결 리스트의 *LNext*

- ✓ 오른쪽 노드로의 이동이 용이하다! 양방향으로 이동이 가능하다!
- ✓ 위의 코드에서 보이듯이 양방향으로 연결한다 하여 더 복잡한 것은 아니다!
그렇게 느낀다면 선입견이다.



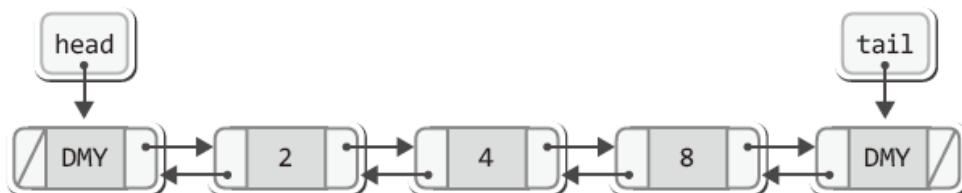
우리가 구현할 양방향 연결 리스트 모델



▶ [그림 05-20: 함께 구현할 양방향 연결 리스트의 구조]

- ✓ LRemove 함수를 ADT에서 제외시킨다.
- ✓ 대신에 왼쪽 노드의 데이터를 참조하는 LPrevious 함수를 ADT에 추가시킨다.
- ✓ 새 노드는 머리에 추가한다.

문제 05-2를 통해서 구현을 요구하는 모델



▶ [그림 05-25: 양쪽으로 더미 노드가 존재하는 양방향 연결 리스트]

문제 05-2에서는 LRemove 함수를 정의한다.
그리고 이 문제는 리스트를 마무리 하는데
있어서 여러 가지 의미를 갖는다.



양방향 연결 리스트의 헤더파일

```
typedef int Data;

typedef struct _node
{
    Data data;
    struct _node * next;
    struct _node * prev;
} Node;

typedef struct _DLinkedList
{
    Node * head;
    Node * cur;
    int numOfData;
} DBLinkedList;
```

```
typedef DBLinkedList List;

void ListInit(List * plist);
void LInsert(List * plist, Data data);

int LFirst(List * plist, Data * pdata);
int LNext(List * plist, Data * pdata);
int LPrevious(List * plist, Data * pdata);
int LCount(List * plist);
```

LPrevious 함수는 LNext 함수와 반대로 이동한다!



양방향 연결 리스트의 활용의 예

```
int main(void)
{
    // 양방향 연결 리스트의 생성 및 초기화 ///////
    List list;
    int data;
    ListInit(&list);

    // 8개의 데이터 저장 ///////
    LInsert(&list, 1); LInsert(&list, 2);
    LInsert(&list, 3); LInsert(&list, 4);
    LInsert(&list, 5); LInsert(&list, 6);
    LInsert(&list, 7); LInsert(&list, 8);
```



추가된 이후의 연결 순서

head → 8 → 7 → 6 → 5 → 4 → 3 → 2 → 1

```
// 저장된 데이터의 조회 ///////
if(LFirst(&list, &data))
{
    printf("%d ", data);

    // 오른쪽 노드로 이동하며 데이터 조회
    while(LNext(&list, &data))
        printf("%d ", data);

    // 왼쪽 노드로 이동하며 데이터 조회
    while(LPrevious(&list, &data))
        printf("%d ", data);

    printf("\n\n");
}

return 0;
```

실행결과

```
8 7 6 5 4 3 2 1 2 3 4 5 6 7 8
```



연결 리스트의 구현: 리스트의 초기화

```
typedef struct _dbLinkedList
{
    Node * head;
    Node * cur;
    int numOfData;
} DBLinkedList;
```

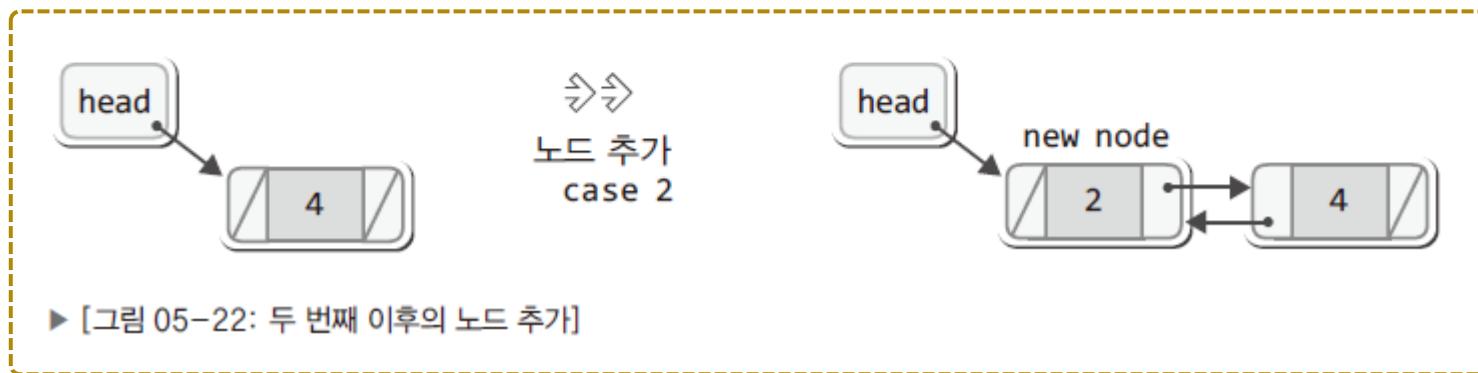
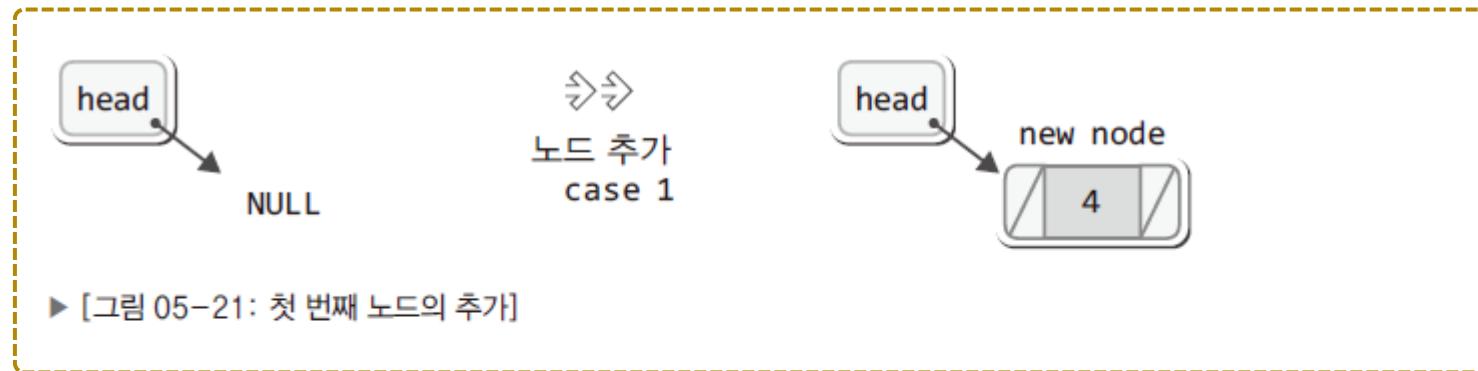
ListInit 함수의 정의에 참조해야 하는 구조체의 정의

```
void ListInit(List * plist)
{
    plist->head = NULL;
    plist->numOfData = 0;
}
```

멤버 cur은 조회의 과정에서 초기화 되는 멤버이니
head와 numOfData만 초기화 하면 된다.



연결 리스트의 구현: 노드 삽입의 구분



더미 노드를 기반으로 하지 않기 때문에 노드의 삽입 방법은 두 가지로 구분이 된다.



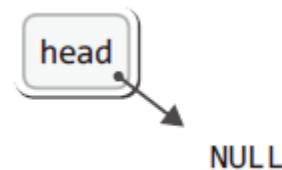
연결 리스트의 구현: 첫 번째 노드의 삽입

첫 번째 노드의 추가 과정만을 담은 결과

```
void LInsert(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

    // 아래 문장에서 plist->head는 NULL이다!
    newNode->next = plist->head;
    newNode->prev = NULL;
    plist->head = newNode;

    (plist->numOfData)++;
}
```



⇒⇒
노드 추가
case 1



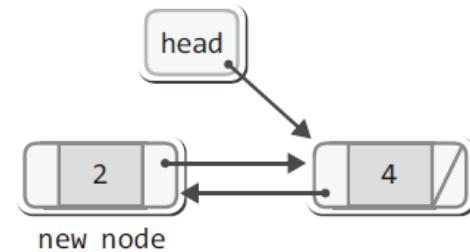
연결 리스트의 구현: 두 번째 이후 노드의 삽입

```
void LInsert(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

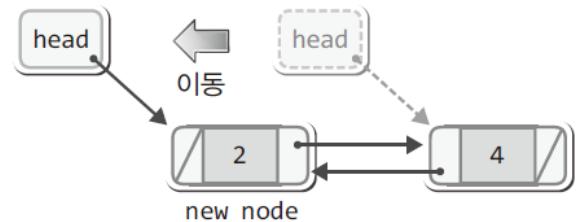
    newNode->next = plist->head;
    if(plist->head != NULL)
        plist->head->prev = newNode;

    newNode->prev = NULL;
    plist->head = newNode;

    (plist->numOfData)++;
}
```



▶ [그림 05-23: 두 번째 노드의 추가과정 1/2]



▶ [그림 05-24: 두 번째 노드의 추가과정 2/2]

첫 번째 노드의 추가 과정에 덧붙여서, if문이 포함하는 문장이 두 번째 이후의 노드 추가 과정에서는 요구가 된다.

연결 리스트의 구현: 데이터 조회

```
int LFirst(List * plist, Data * pdata)
{
    if(plist->head == NULL)
        return FALSE;

    plist->cur = plist->head;

    *pdata = plist->cur->data;
    return TRUE;
}
```

```
int LNext(List * plist, Data * pdata)
{
    if(plist->cur->next == NULL)
        return FALSE;

    plist->cur = plist->cur->next;
    *pdata = plist->cur->data;
    return TRUE;
}
```

```
int LPrevious(List * plist, Data * pdata)
{
    if(plist->cur->prev == NULL)
        return FALSE;

    plist->cur = plist->cur->prev;

    *pdata = plist->cur->data;
    return TRUE;
}
```

LFirst 함수와 LNext 함수는 사실상 단방향 연결 리스트의 경우와 차이가 없다. 그리고 LPrevious 함수는 LNext 함수와 이동 방향에서만 차이가 난다.



양방향 연결 리스트의 구현을 하나로 묶기

DBLinkedList.c
DBLinkedList.h
DBLinkedListMain.c

실행결과

```
8 7 6 5 4 3 2 1 2 3 4 5 6 7 8
```



수고하셨습니다~



Chapter 05에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 06. 스택(Stack)

Introduction To Data Structures Using C

Chapter 06. 스택(Stack)



Chapter 06-1:

스택의 이해와 ADT 정의



스택(Stack)의 이해



- 스택은 '먼저 들어간 것이 나중에 나오는 자료구조'로써 초코볼이 담겨있는 통에 비유할 수 있다.
- 스택은 'LIFO(Last-in, First-out) 구조'의 자료구조이다.

- 초코볼 통에 초코볼을 넣는다.
- 초코볼 통에서 초코볼을 꺼낸다.
- 이번에 꺼낼 초코볼의 색이 무엇인지 통 안을 들여다 본다.

push
pop
peek

스택의
기본 연산

일반적인 자료구조의 학습에서 스택의 이해와 구현은 어렵지 않다. 오히려 활용의 측면에서 생각할 것들이 많다!



스택의 ADT 정의

- void StackInit(Stack * pstack);
 - 스택의 초기화를 진행한다.
 - 스택 생성 후 제일 먼저 호출되어야 하는 함수이다.
- int SIsEmpty(Stack * pstack);
 - 스택이 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.
- void SPush(Stack * pstack, Data data);
 - 스택에 데이터를 저장한다. 매개변수 data로 전달된 값을 저장한다.
- Data SPop(Stack * pstack);
 - 마지막에 저장된 요소를 삭제한다.
 - 삭제된 데이터는 반환된다.
 - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.
- Data SPeek(Stack * pstack);
 - 마지막에 저장된 요소를 반환하되 삭제하지 않는다.
 - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.



Chapter 06. 스택(Stack)



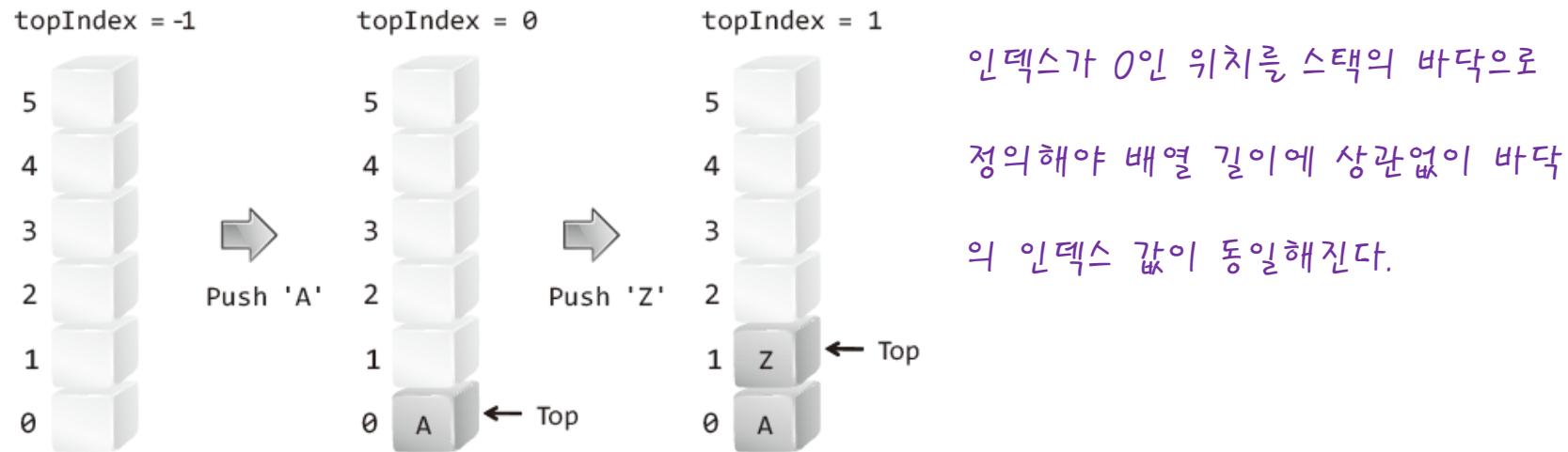
Chapter 06-2:

스택의 배열 기반 구현



구현의 논리

두 번의 PUSH 연산



▶ [그림 06-1: 배열 기반 스택의 push 연산]

- 인덱스 0의 배열 요소가 '스택의 바닥(초코볼 통의 바닥)'으로 정의되었다.
- 마지막에 저장된 데이터의 위치를 기억해야 한다.
- **push** Top을 위로 한 칸 올리고, Top이 가리키는 위치에 데이터 저장
- **pop** Top이 가리키는 데이터를 반환하고, Top을 아래로 한 칸 내림

스택의 헤더파일

```
#define TRUE      1
#define FALSE     0
#define STACK_LEN 100

typedef int Data;

typedef struct _arrayStack
{
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;

typedef ArrayStack Stack;

void StackInit(Stack * pstack);           // 스택의 초기화
int SIsEmpty(Stack * pstack);            // 스택이 비었는지 확인

void SPush(Stack * pstack, Data data);    // 스택의 push 연산
Data SPop(Stack * pstack);                // 스택의 pop 연산
Data SPeek(Stack * pstack);               // 스택의 peek 연산
```

배열 기반을 고려하여 정의된
스택의 구조체!



배열 기반 스택의 구현: 초기화 및 기타 함수

```
typedef struct _arrayStack
{
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;
```

```
void StackInit(Stack * pstack)
```

```
{
```

```
    pstack->topIndex = -1;
```

```
}
```

-1은 스택이 비었음을 의미

```
int SIsEmpty(Stack * pstack)
```

```
{
```

```
    if(pstack->topIndex == -1)
```

```
        return TRUE;
```

```
    else
```

```
        return FALSE;
```

```
}
```

빈 경우 TRUE를 반환



배열 기반 스택의 구현: PUSH, POP, PEEK

```
void SPush(Stack * pstack, Data data)
{
    pstack->topIndex += 1;
    pstack->stackArr[pstack->topIndex] = data;
}
```

```
Data SPop(Stack * pstack)
{
    int rIdx;

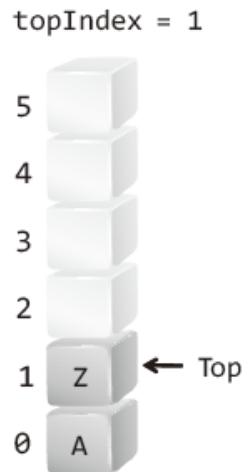
    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }

    rIdx = pstack->topIndex;
    pstack->topIndex -= 1;

    return pstack->stackArr[rIdx];
}
```

```
Data SPeek(Stack * pstack)
{
    if(SIsEmpty(pstack))
    {
        printf("Stack Memory Error!");
        exit(-1);
    }

    return pstack->stackArr[pstack->topIndex];
}
```



배열 기반 스택의 활용: main 함수

```
int main(void)
{
    // Stack의 생성 및 초기화 ///////
    Stack stack;
    StackInit(&stack);

    // 데이터 넣기 ///////
    SPush(&stack, 1); SPush(&stack, 2);
    SPush(&stack, 3); SPush(&stack, 4);
    SPush(&stack, 5);

    // 데이터 꺼내기 ///////
    while(!SIsEmpty(&stack))
        printf("%d ", SPop(&stack));

    return 0;
}
```

ArrayBaseStack.h
ArrayBaseStack.c
ArrayBaseStackMain.c

5 4 3 2 1

실행결과



Chapter 06. 스택(Stack)



Chapter 06-3:

스택의 연결 리스트 기반 구현



연결 리스트 기반 스택의 논리와 헤더파일의 정의

이렇듯 메모리 구조만 보아서는 스택임이 구분되지 않는다!



▶ [그림 06-2: 스택의 구현에 활용할 리스트 모델]

저장된 순서의 역순으로 데이터(노드)를 참조(삭제)
하는 연결 리스트가 바로 연결 기반의 스택이다!

```
typedef int Data;

typedef struct _node
{
    Data data;
    struct _node * next;
} Node;

typedef struct _listStack
{
    Node * head;
} ListStack;
```

문제 06-1에서는 CLinkedList.h와 CLinkedList.c를
단순히 활용하여 스택의 구현을 요구한다!

```
typedef ListStack Stack;

void StackInit(Stack * pstack);
int SIsEmpty(Stack * pstack);

void SPush(Stack * pstack, Data data);
Data SPop(Stack * pstack);
Data SPeek(Stack * pstack);
```



연결 리스트 기반 스택의 구현 1

```
void StackInit(Stack * pstack)
{
    pstack->head = NULL;
}
```

```
Data SPop(Stack * pstack)
{
    Data rdata;
    Node * rnode;

    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }
    rdata = pstack->head->data;
    rnode = pstack->head;
    pstack->head = pstack->head->next;
    free(rnode);
    return rdata;
}
```

```
int SIsEmpty(Stack * pstack)
{
    if(pstack->head == NULL)
        return TRUE;
    else
        return FALSE;
}
```

새 노드를 머리에 추가하고, 삭제 시 머리부
터 삭제하는 단순 연결 리스트의 코드에 지
나지 않는다.



연결 리스트 기반 스택의 구현 2

```
void SPush(Stack * pstack, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));

    newNode->data = data;
    newNode->next = pstack->head;

    pstack->head = newNode;
}
```

```
Data SPeek(Stack * pstack)
{
    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }
    return pstack->head->data;
}
```

연결 리스트 기반 스택을 직접 구현하는 것보다
의미 있는 것은 문제 06-1을 해결하는 것이다!



연결 기반 스택의 활용: main 함수

```
int main(void)
{
    // Stack의 생성 및 초기화 ///////
    Stack stack;
    StackInit(&stack);

    // 데이터 넣기 ///////
    SPush(&stack, 1); SPush(&stack, 2);
    SPush(&stack, 3); SPush(&stack, 4);
    SPush(&stack, 5);

    // 데이터 꺼내기 ///////
    while(!SIsEmpty(&stack))
        printf("%d ", SPop(&stack));

    return 0;
}
```

배열 기반 리스트 관련 main 함수와 완전히 동일하게 정의된 main 함수!

ListBaseStack.h
ListBaseStack.c
ListBaseStackMain.c

5 4 3 2 1

실행결과



Chapter 06. 스택(Stack)



Chapter 06-4:

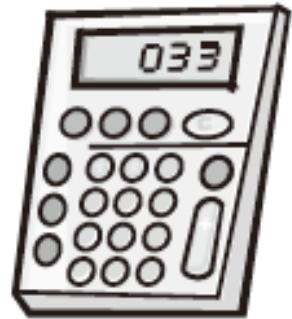
계산기 프로그램 구현



구현할 계산기 프로그램의 성격

다음과 같은 문장의 수식을 계산할 수 있어야 한다.

$$(3 + 4) * (5 / 2) + (7 + (9 - 5))$$



다음과 같은 문장의 수식계산을 위해서는 다음 두 가지를 고려해야 한다.

- 소괄호를 파악하여 그 부분을 먼저 연산한다.
- 연산자의 우선순위를 근거로 연산의 순위를 결정한다.

계산기 구현에 필요한 알고리즘은 스택과는 별개의 것이다.

다만 그 알고리즘의 구현에 있어서 스택이 매우 요긴하게 활용된다.



세 가지 수식의 표기법: 전위, 중위, 후위

- 중위 표기법(infix notation)

예) $5 + 2 / 7$

수식 내에 연산의 순서에 대한 정보가 담겨 있지 않다. 그래서 소괄호와 연산자의 우선순위라는 것을 정의하여 이를 기반으로 연산의 순서를 명시한다.

- 전위 표기법(prefix notation)

예) $+ 5 / 2 7$

수식 내에 연산의 순서에 대한 정보가 담겨 있다. 그래서 소괄호가 필요 없고 연산의 우선순위를 결정할 필요도 없다.

- 후위 표기법(postfix notation)

예) $5 2 7 / +$

전위 표기법과 마찬가지로 수식 내에 연산의 순서에 대한 정보가 담겨 있다. 그래서 소괄호가 필요 없고 연산의 우선순위를 결정할 필요도 없다.

소괄호와 연산자의 우선순위를 인식하게 하여 중위 표기법의 수식을 직접 계산하게 프로그래밍 하는 것보다 후위 표기법의 수식을 계산하도록 프로그래밍 하는 것이 더 쉽다!



중위 → 후위 : 소괄호 고려하지 않고 1



▶ [그림 06-3: 수식 변환의 과정 1/7]

수식을 이루는 왼쪽 문자부터 시작해서 하나씩 처리해 나간다.



▶ [그림 06-4: 수식 변환의 과정 2/7]

피 연산자를 만나면 무조건 변환된 수식이 위치할 자리로 이동시킨다.



중위 → 후위 : 소괄호 고려하지 않고 2



▶ [그림 06-5: 수식 변환의 과정 3/7]

연산자를 만나면 무조건 쟁반으로 이동한다.



▶ [그림 06-6: 수식 변환의 과정 4/7]

숫자를 만났으니 변환된 수식이 위치할 자리로 이동!



중위 → 후위 : 소괄호 고려하지 않고 3

/ 연산자의 우선순위가 높으므로 + 연산자 위에 올린다.



▶ [그림 06-7: 수식 변환의 과정 5/7]

쟁반에 기존 연산자가 있는 상황에서의 행동 방식!

☞ 쟁반에 위치한 연산자의 우선순위가 높다면

- 쟁반에 위치한 연산자를 꺼내서 변환된 수식이 위치할 자리로 옮긴다.
- 그리고 새 연산자는 쟁반으로 옮긴다.

☞ 쟁반에 위치한 연산자의 우선순위가 낮다면

- 쟁반에 위치한 연산자의 위에 새 연산자를 쌓는다.

우선순위가 높은 연산자는 우선순위가 낮은 연산자 위에 올라서서, 우선순위가 낮은 연산자가 먼저 자리를 잡지 못하게 하려는 목적!



중위 → 후위 : 소괄호 고려하지 않고 4



▶ [그림 06-8: 수식 변환의 과정 6/7]

피 연산자는 무조건 변환된 수식의 자리로 이동!



▶ [그림 06-9: 수식 변환의 과정 7/7]

나머지 연산자들은 쟁반에서 차례로 옮긴다!



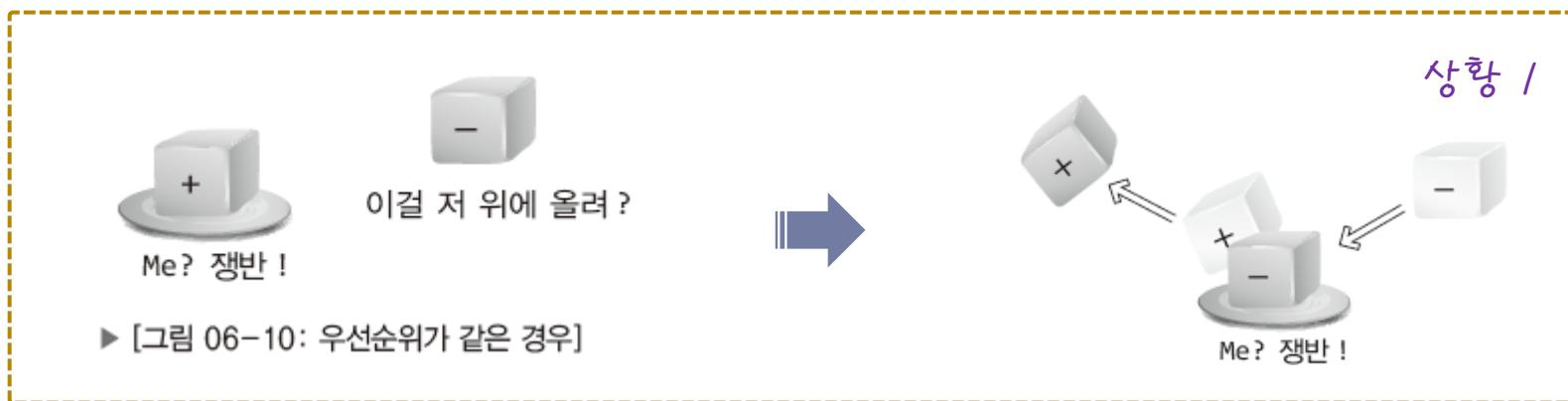
중위 → 후위 : 정리하면

변환 규칙의 정리 내용

- 피 연산자는 그냥 옮긴다.
- 연산자는 쟁반으로 옮긴다.
- 연산자가 쟁반에 있다면 우선순위를 비교하여 처리방법을 결정한다.
- 마지막까지 쟁반에 남아있는 연산자들은 하나씩 꺼내서 옮긴다.



중위 → 후위 : 고민 될 수 있는 상황



+ 연산자가 우선순위가 높다고 가정하고(+ 연산자가 먼저 등장했으므로) 일을 진행한다.
즉 + 연산자를 옮기고 그 자리에 – 연산자를 가져다 놔야 한다."



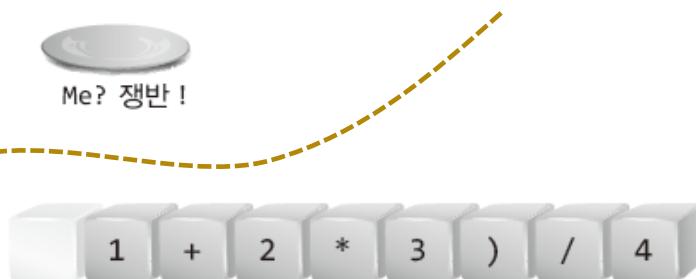
중위 → 후위 : 소괄호 고려 1

후위 표기법의 수식에서는 먼저 연산이 이뤄져야 하는 연산자가 뒤에 연산이 이뤄지는 연산자보다 앞에 위치해야 한다. 따라서 소괄호 안에 있는 연산자들이 후위 표기법의 수식에서 앞부분에 위치해야 한다.



▶ [그림 06-14: 소괄호가 포함된 수식의 변환 1/6]

(연산자의 우선순위는 그 어떤 사칙 연산자들보다 낮다고 간주! 그래서) 연산자가 등장할 때까지 쟁반에 남아 소괄호의 경계 역할을 해야 함!



▶ [그림 06-15: 소괄호가 포함된 수식의 변환 2/6]



중위 → 후위 : 소괄호 고려 2



변환된 수식이 위치할 자리



Me? 쟁반!

▶ [그림 06-16: 소괄호가 포함된 수식의 변환 3/6]



변환된 수식이 위치할 자리



Me? 쟁반!

▶ [그림 06-17: 소괄호가 포함된 수식의 변환 4/6]



변환된 수식이 위치할 자리

▶ [그림 06-18: 소괄호가 포함된 수식의 변환 5/6]



Me? 쟁반!

) 연산자를 만나면 쟁반에서 (연산자 만날 때까지 연산자
를 변환된 수식의 자리로 옮긴다!



중위 → 후위 : 소괄호 고려 3



▶ [그림 06-19: 소괄호가 포함된 수식의 변환 6/6]

조금 달리 설명하면 (연산자는 쟁반의 또 다른 바닥이다! 그리고) 연산자는 변환이 되어야 하는 작은 수식의 끝을 의미한다. 그래서) 연산자를 만나면 (연산자를 만날 때까지 연산자를 이동 시켜야 한다.

지금까지 설명한 내용에 해당하는 코드의 구현에 앞서 중위 표기법의 수식을 후위 표기법의 수식으로 바꾸는 연습을 할 필요가 있다.



중위 → 후위 : 프로그램 구현 1

```
void ConvToRPNExp(char exp[])
{
    . . .
}
```

변환 함수의 타입

함수 이름의 일부인 RPN은 후위 표기법의 또 다른 이름인 Reverse Polish Notation의 약자이다.

```
int main(void)
{
    char exp[] = "3-2+4";
    ConvToRPNExp(exp);
    . . .
}
```

중위 표기법 수식을 배열에 담아 함수의 인자로 전달한다.

호출 완료 후 exp에는 변환된 수식이 담긴다.



중위 → 후위 : 프로그램 구현 2

함수 ConvToRPNExp의 첫 번째 helper function!

```
int GetOpPrec(char op)          // 연산자의 연산 우선순위 정보를 반환한다.  
{  
    switch(op)  
    {  
        case '*':  
        case '/':  
            return 5;           // 가장 높은 연산의 우선순위  
        case '+':  
        case '-':  
            return 3;  
        case '(':  
            return 1;           // (연산자는) 연산자가 등장할 때까지 쟁반에 남아 있어야 하기 때문에 가장 낮은 우선순위를 부여!  
    }  
  
    return -1;                // 등록되지 않은 연산자임을 알림!  
}
```

-) 연산자는 소괄호의 끝을 알리는 메시지의 역할을 한다. 따라서 쟁반으로 가지 않는다.
- 때문에) 연산자에 대한 반환 값은 정의되어 있을 필요가 없다!



중위 → 후위 : 프로그램 구현 3

함수 ConvToRPNEexp의 두 번째 helper function!

```
int WhoPrecOp(char op1, char op2)      두 연산자의 우선순위 비교 결과를 반환한다.  
{  
    int op1Prec = GetOpPrec(op1);  
    int op2Prec = GetOpPrec(op2);  
  
    if(op1Prec > op2Prec)          // op1의 우선순위가 더 높다면  
        return 1;  
    else if(op1Prec < op2Prec)      // op2의 우선순위가 더 높다면  
        return -1;  
    else  
        return 0;                  // op1과 op2의 우선순위가 같다면  
}
```

ConvToRPNEexp 함수의 실질적인 Helper Function은 위의 함수 하나이다!



중위 → 후위 : 프로그램 구현 4

```
void ConvToRPNExp(char exp[])
{
    Stack stack;
    int expLen = strlen(exp);
    char * convExp = (char*)malloc(expLen+1);    변환된 수식을 담을 공간 마련

    int i, idx=0;
    char tok, popOp;

    memset(convExp, 0, sizeof(char)*expLen+1);    마련한 공간 0으로 초기화
    StackInit(&stack);

    for(i=0; i<expLen; i++) {
        ...
    }                                일련의 변환 과정을 이 반복문 안에서 수행

    while(!SIsEmpty(&stack))
        convExp[idx++] = SPop(&stack);    스택에 남아 있는 모든 연산자를 이동시키는 반복문

    strcpy(exp, convExp);    변환된 수식을 반환!
    free(convExp);
}
```



중위 → 후위 : 프로그램 구현 5

```
void ConvToRPNExp(char exp[])
{
    ...
    for(i=0; i<expLen; i++)
    {
        tok = exp[i];

        if(isdigit(tok))    tok에 저장된 문자가 피연산자라면
        {
            convExp[idx++] = tok;
        }
        else                tok에 저장된 문자가 연산자라면
        {
            switch(tok)
            {
                ...
                ...
            }
        }
    }
}
```

연산자일 때의 처리 루틴을 switch문에 담는다!



중위 → 후위 : 프로그램 구현 6

```
switch(tok)
{
    case '(': // 여는 소괄호라면,
        SPush(&stack, tok); // 스택에 쌓는다.
        break;
    case ')': // 닫는 소괄호라면,
        while(1) // 반복해서,
        {
            popOp = SPop(&stack); // 스택에서 연산자를 꺼내어,
            if(popOp == '(') // 연산자 ( 을 만날 때까지,
                break;
            convExp[idx++] = popOp; // 배열 convExp에 저장한다.
        }
        break;
    case '+': case '-':
    case '*': case '/': // tok에 저장된 연산자를 스택에 저장하기 위한 과정
        while(!SIsEmpty(&stack) && WhoPrecOp(SPeek(&stack), tok) >= 0)
            convExp[idx++] = SPop(&stack);
        SPush(&stack, tok);
        break;
}
```

함수 ConvToRPNExp의 일부인 switch문

tok에 저장된 연산자를 스택에 저장하기 위한 과정

중위 → 후위 : 프로그램의 실행

```
int main(void)
{
    char exp1[] = "1+2*3";
    char exp2[] = "(1+2)*3";
    char exp3[] = "((1-2)+3)*(5-2)";

    ConvToRPNExp(exp1);
    ConvToRPNExp(exp2);
    ConvToRPNExp(exp3);

    printf("%s \n", exp1);
    printf("%s \n", exp2);
    printf("%s \n", exp3);
    return 0;
}
```

InfixToPostfix.h InfixToPostfix.c ConvToRPNExp 함수의
ListBaseStack.h ListBaseStack.c 선언과 정의
 스택관련 함수의
 선언과 정의
InfixToPostfixMain.c main 함수의 정의

123*+
12+3*
12-3+52-*

실행결과

후기 표기법 수식의 계산

3 + 2 * 4

↓ 후위 표기법 수식으로

3 2 4 * +

↓

3 2 4 * +

↓ 2와 4의 곱 진행

3 8 +

피연산자 두 개가 연산자 앞에 항상 위치하는 구조

(1 * 2 + 3) / 4

↓ 후위 표기법 수식으로

1 2 * 3 + 4 /

↓ 1과 2의 곱 진행

2 3 + 4 /

↓ 2와 3의 합 진행

5 4 /



후기 표기법 수식 계산 프로그램의 구현

계산의 규칙

- 피연산자는 무조건 스택으로 옮긴다.
- 연산자를 만나면 스택에서 두 개의 피연산자를 꺼내서 계산을 한다.
- 계산결과는 다시 스택에 넣는다.



▶ [그림 06-20: 후위 표기법의 수식 계산 1]

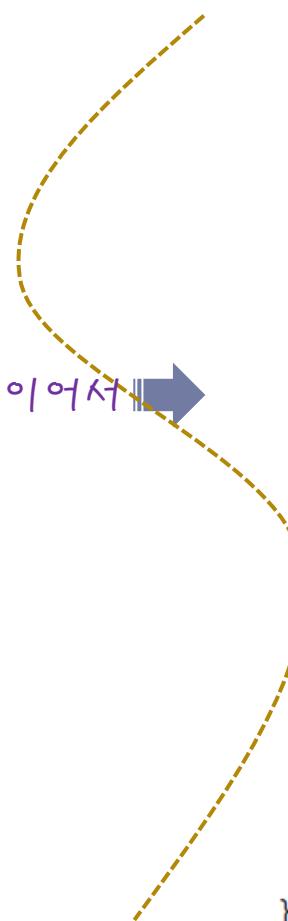


▶ [그림 06-21: 후위 표기법의 수식 계산 2]

후기 표기법 수식 계산 프로그램의 구현

```
int EvalRPNExp(char exp[])
{
    Stack stack;
    int expLen = strlen(exp);
    int i;
    char tok, op1, op2;

    StackInit(&stack);

    for(i=0; i<expLen; i++)
    {
        tok = exp[i];
        if(isdigit(tok))
        {
            SPush(&stack, tok - '0');
        }
        else
        {
            op2 = SPop(&stack);
            op1 = SPop(&stack);

이 어서 ➡
            switch(tok)
            {
                case '+':
                    SPush(&stack, op1+op2);
                    break;
                case '-':
                    SPush(&stack, op1-op2);
                    break;
                case '*':
                    SPush(&stack, op1*op2);
                    break;
                case '/':
                    SPush(&stack, op1/op2);
                    break;
            }
        }
    }
    return SPop(&stack);
}
```



후위 표기법 수식 계산 프로그램의 실행

PostCalculator.h EvalRPNEval 함수의
PostCalculator.c 선언과 정의

ListBaseStack.h 스택관련 함수의
ListBaseStack.c 선언과 정의

PostCalculatorMain.c main 함수의 정의

```
int main(void)
{
    char postExp1[] = "42*8+";
    char postExp2[] = "123+*4/";

    printf("%s = %d \n", postExp1, EvalRPNEval(postExp1));
    printf("%s = %d \n", postExp2, EvalRPNEval(postExp2));

    return 0;
}
```

42*8+ = 16

123+*4/ = 1

실행결과

계산기 프로그램의 완성1

계산의 과정

중위 표기법 수식 → ConvToRPNExp → EvalRPNExp → 연산결과

계산기 프로그램의 파일 구성

• 스택의 활용	ListBaseStack.h, ListBaseStack.c
• 후위 표기법의 수식으로 변환	InfixToPostfix.h, InfixToPostfix.c
• 후위 표기법의 수식을 계산	PostCalculator.h, PostCalculator.c
• 중위 표기법의 수식을 계산	InfixCalculator.h, InfixCalculator.c
• main 함수	InfixCalculatorMain.c

InfixCalculator.h

InfixCalculator.c



계산기 프로그램의 완성2

InfixCalculator.h

```
#ifndef __INFIX_CALCULATOR__
#define __INFIX_CALCULATOR__

int EvalInfixExp(char exp[]);

#endif

int EvalInfixExp(char exp[])
{
    int len = strlen(exp);
    int ret;
    char * expcpy = (char*)malloc(len+1); // 문자열 저장공간 마련
    strcpy(expcpy, exp); // exp를 expcpy에 복사

    ConvToRPNExp(expcpy); // 후위 표기법의 수식으로 변환
    ret = EvalRPNExp(expcpy); // 변환된 수식의 계산

    free(expcpy); // 문자열 저장공간 해제
    return ret; // 계산결과 반환
}
```

InfixCalculatorMain.c

```
int main(void)
{
    char exp1[] = "1+2*3";
    char exp2[] = "(1+2)*3";
    char exp3[] = "((1-2)+3)*(5-2)";

    printf("%s = %d \n", exp1, EvalInfixExp(exp1));
    printf("%s = %d \n", exp2, EvalInfixExp(exp2));
    printf("%s = %d \n", exp3, EvalInfixExp(exp3));
    return 0;
}
```

// 문자열 저장공간 마련
// exp를 expcpy에 복사

실행결과

```
1+2*3 = 7
(1+2)*3 = 9
((1-2)+3)*(5-2) = 6
```

수고하셨습니다~



Chapter 06에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 07. 큐(Queue)

Introduction To Data Structures Using C

Chapter 07. 큐(Queue)



Chapter 07-1:

큐의 이해와 ADT 정의



큐(Stack)의 이해와 ADT 정의



큐는 'LIFO(Last-in, First-out) 구조'의 자료구조이다. 때문에 먼저 들어간 것이 먼저 나오는, 일종의 **출서기**에 비유할 수 있는 자료구조이다.

- 큐에 데이터를 넣는 연산
- 큐에서 데이터를 꺼내는 연산

enqueue
dequeue

'] '큐'의 기본 연산

큐는 운영체제 관점에서 보면 프로세스나 쓰레드의 관리에 활용이 되는 자료구조이다. 이렇듯 운영체제의 구현에도 자료구조가 사용이 된다. 따라서 운영체제의 이해를 위해서는 자료구조에 대한 이해가 선행되어야 한다.



큐의 ADT 정의

- void QueueInit(Queue * pq);
 - 큐의 초기화를 진행한다.
 - 큐 생성 후 제일 먼저 호출되어야 하는 함수이다.
- int QIsEmpty(Queue * pq);
 - 큐가 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.
- void Enqueue(Queue * pq, Data data); *enqueue 연산*
 - 큐에 데이터를 저장한다. 매개변수 data로 전달된 값을 저장한다.
- Data Dequeue(Queue * pq); *dequeue 연산*
 - 저장순서가 가장 앞선 데이터를 삭제한다.
 - 삭제된 데이터는 반환된다.
 - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.
- Data QPeek(Queue * pq); *peek 연산*
 - 저장순서가 가장 앞선 데이터를 반환하되 삭제하지 않는다.
 - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.



Chapter 07. 큐(Queue)

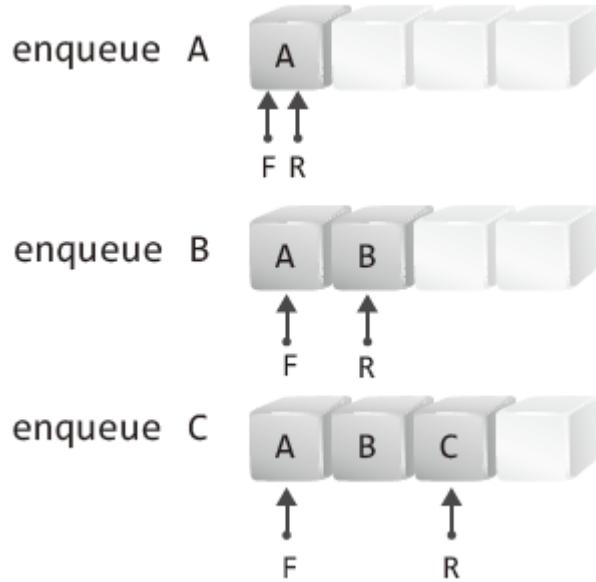


Chapter 07-2:

큐의 배열 기반 구현

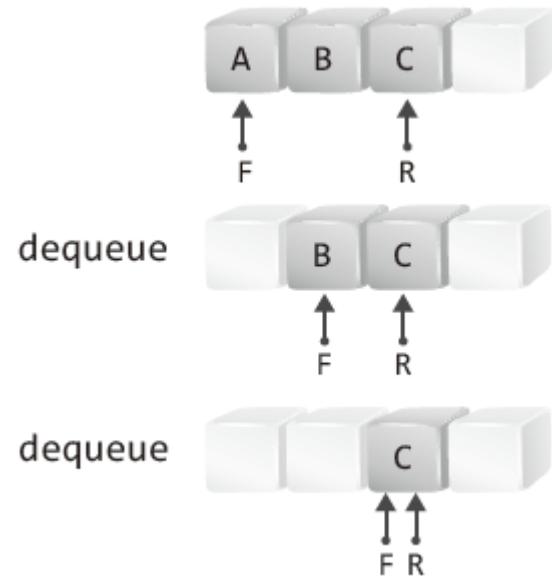


큐의 구현 논리



보편적이고도 올바른 *enqueue*
연산에 대한 방식

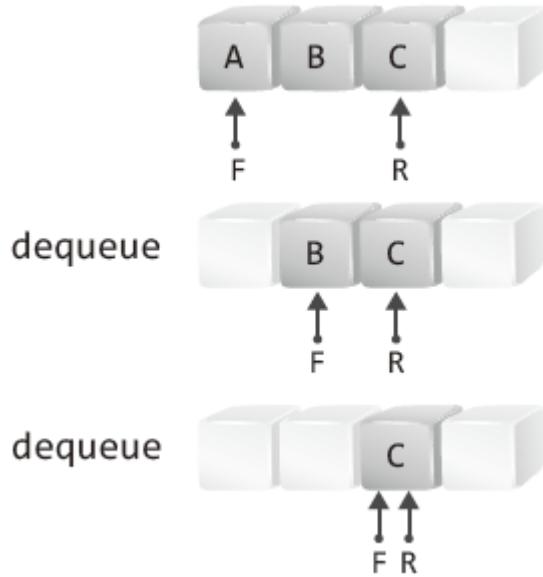
보편적이고도 올바른 *dequeue*
연산에 대한 방식



큐의 꼬리를 의미하는 R을 한칸 이동시키고
새 데이터를 저장한다.

큐의 머리를 의미하는 F가 가리키는 데이터를 반환하고 F를 한 칸 이동시킨다.

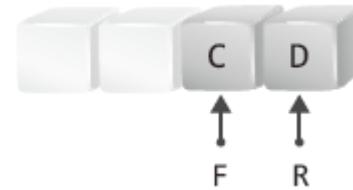
가장 기본적인 배열 기반 큐의 문제점



분명 배열은 비어있다. 하지만 R을 오른쪽으로 한칸 이동시킬 수 없어서 더 이상 데이터를 추가할 수 없다!



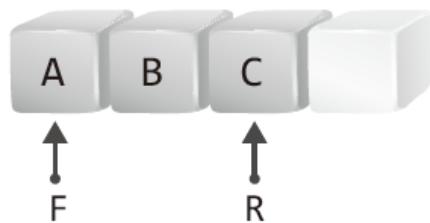
enqueue D



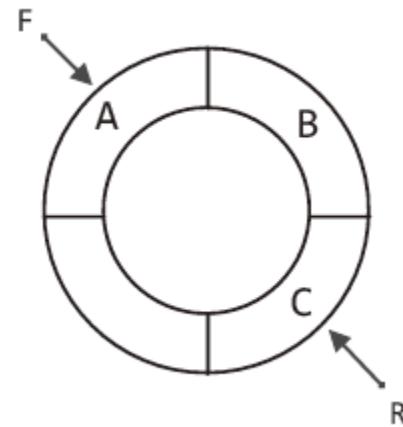
데이터를 더 추가하기 위해서는 R을 인덱스가 0인 위치로 이동시켜야 한다.
그리고 이러한 방식으로 문제를 해결한 것이 바로 '원형 큐'이다!



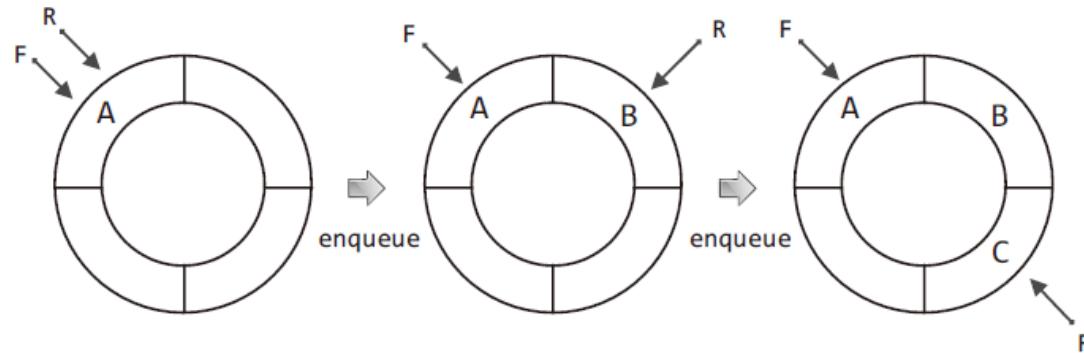
원형 큐의 소개



배열의 머리와 끝을 연결한 구조를
원의 형태로 바라본다.



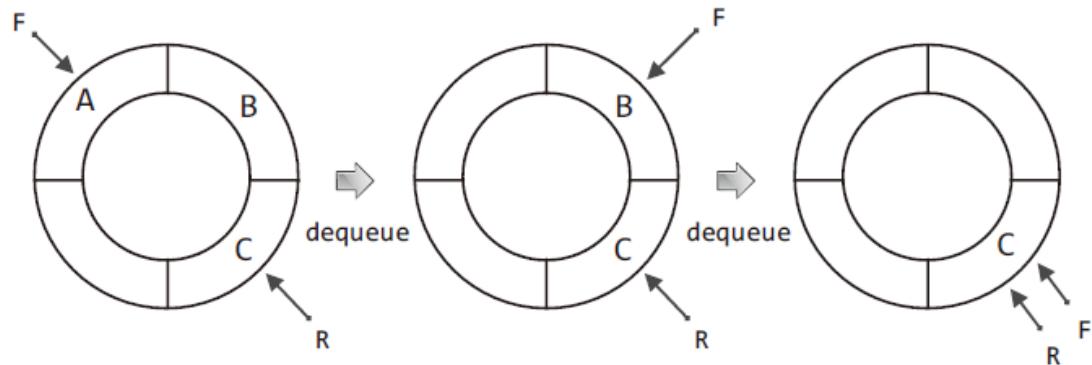
원형 큐의 단순한 연산



단순 배열 큐와 마찬가지로 R이
이동한 다음에 데이터 저장!

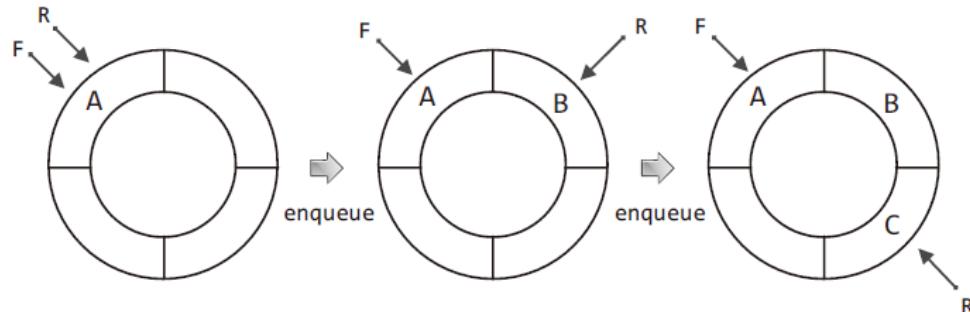
▶ [그림 07-6: 원형 큐의 enqueue 연산]

단순 배열 큐와 마찬가지로 F가
가리키는 데이터 반환 후 F 이동!



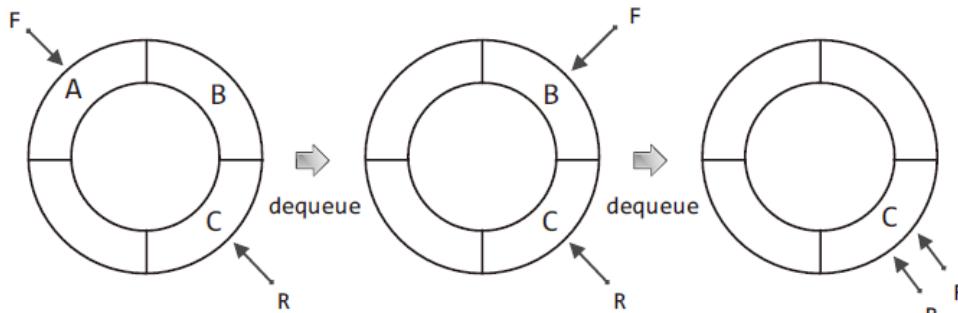
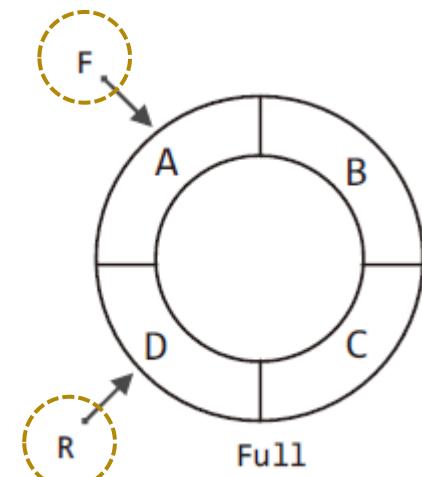
▶ [그림 07-7: 원형 큐의 dequeue 연산]

원형 큐의 단순한 연산의 문제점



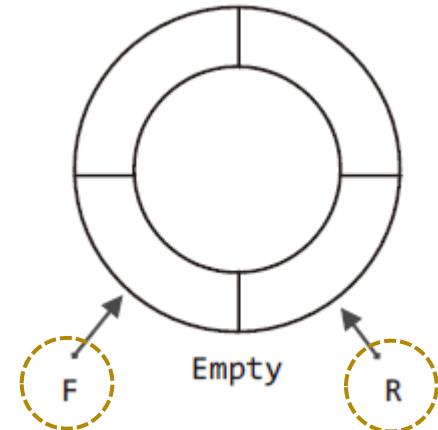
▶ [그림 07-6: 원형 큐의 enqueue 연산]

꽉 채운다!

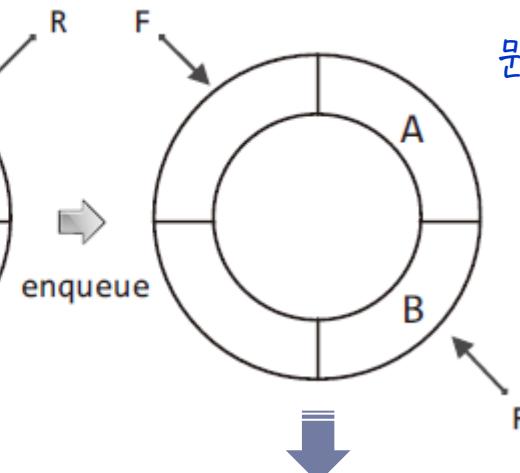
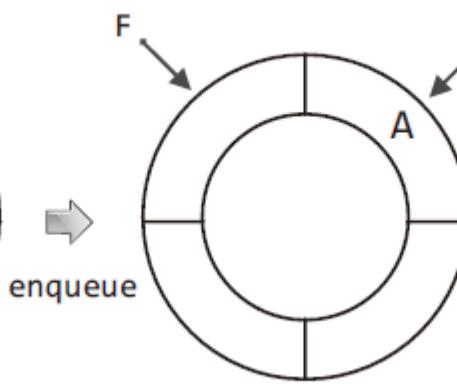
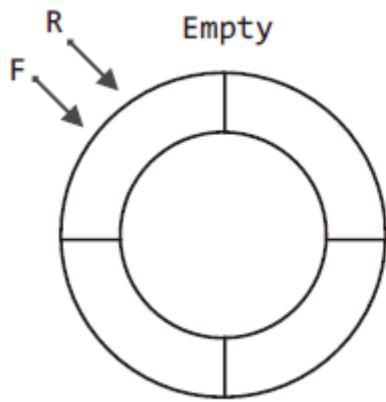


텅 비운다!

▶ [그림 07-7: 원형 큐의 dequeue 연산]



원형 큐의 문제점 해결

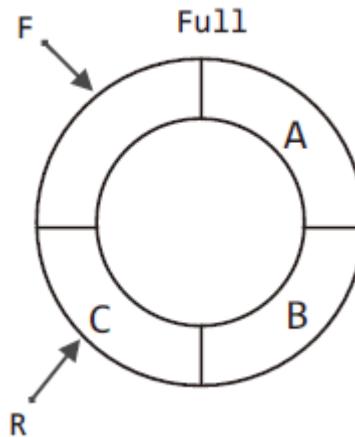


저장 공간 하나를
잃고
문제점을 해결한다!

데이터가 하나 존재하는 경우 F와 R이 같은 위치를
가리켰는데, 초기화 직후에 이 상태가 되게 한다.

그냥 하나 비운 상태에서 FULL로 인정한다!

그러면 Empty와 Full의 구분이 가능하다!



원형 큐의 구현: 헤더파일

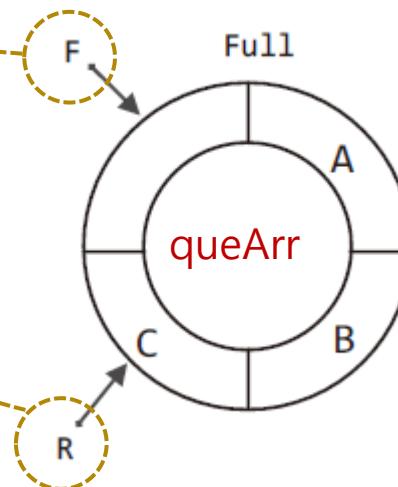
```
#define QUE_LEN 100  
typedef int Data;
```

```
typedef struct _cQueue  
{  
    int front;  
    int rear;  
    Data queArr[QUE_LEN];  
} CQueue;
```

```
typedef CQueue Queue;
```

```
void QueueInit(Queue * pq);  
int QIsEmpty(Queue * pq);
```

```
void Enqueue(Queue * pq, Data data);  
Data Dequeue(Queue * pq);  
Data QPeek(Queue * pq);
```



원형 큐의 구현: Helper Function

```
int NextPosIdx(int pos)
{
    if(pos == QUE_LEN-1)
        return 0;
    else
        return pos+1;
}
```

큐의 연산에 의해서 F(front)와 R(rear)이 이동할때 이동해야 할 위치
를 알려주는 함수! 원형 큐를 완성하게 하는 실질적인 함수이다!

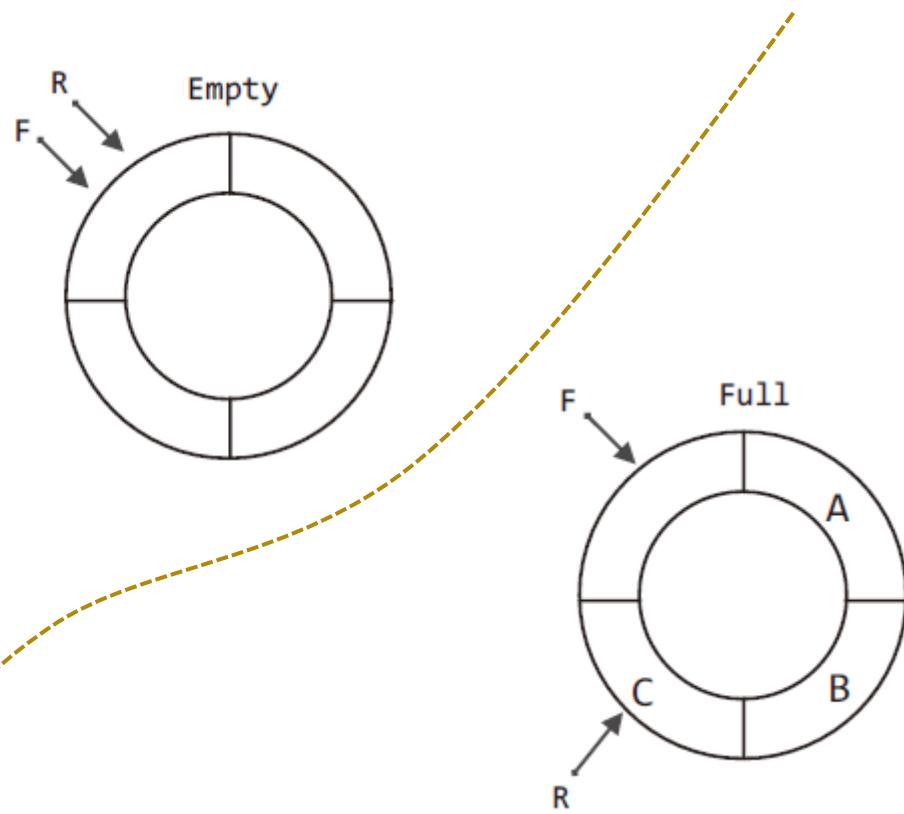
위 함수의 정의를 통해서 원형 큐의 나머지 구현은 매우 간단해진다.



원형 큐의 구현: 함수의 정의1

```
void QueueInit(Queue * pq)
{
    pq->front = 0;
    pq->rear = 0;
}
```

```
int QIsEmpty(Queue * pq)
{
    if(pq->front == pq->rear)
        return TRUE;
    else
        return FALSE;
}
```



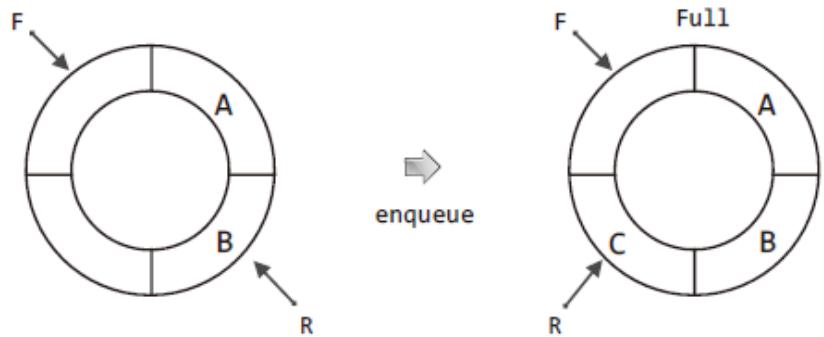
원형 큐의 구현: 함수의 정의2

```
void Enqueue(Queue * pq, Data data)
{
    if(NextPosIdx(pq->rear) == pq->front)
    {
        printf("Queue Memory Error!");
        exit(-1);
    }
    pq->rear = NextPosIdx(pq->rear);
    pq->queArr[pq->rear] = data;
}
```

rear을 이동시키고(NextPosIdx 함수의 호출을 통해),
그 위치에 데이터 저장

두 연산 모두 rear와 front를 우선 이동시키는 구조이다!

front를 이동시키고(NextPosIdx 함수의 호출을 통해),
그 위치의 데이터 반환



```
Data Dequeue(Queue * pq)
{
    if(QIsEmpty(pq))
    {
        printf("Queue Memory Error!");
        exit(-1);
    }
    pq->front = NextPosIdx(pq->front);
    return pq->queArr[pq->front];
}
```



원형 큐의 실행

```
int main(void)
{
    // Queue의 생성 및 초기화 ///////
    Queue q;
    QueueInit(&q);

    // 데이터 넣기 ///////
    Enqueue(&q, 1); Enqueue(&q, 2);
    Enqueue(&q, 3); Enqueue(&q, 4);
    Enqueue(&q, 5);

    // 데이터 꺼내기 ///////
    while(!QIsEmpty(&q))
        printf("%d ", Dequeue(&q));

    return 0;
}
```

CircularQueue.h } 원형 큐의 구현 결과
CircularQueue.c }
CircularQueueMain.c main 함수의 정의

1 2 3 4 5

실행결과

Chapter 07. 큐(Queue)



Chapter 07-3:

큐의 연결 리스트 기반 구현



연결 리스트 기반 큐의 헤더파일

```
typedef int Data;  
  
typedef struct _node  
{  
    Data data;  
    struct _node * next;  
} Node;  
  
typedef struct _lQueue  
{  
    Node * front;  
    Node * rear;  
} LQueue;
```

```
typedef LQueue Queue;  
  
void QueueInit(Queue * pq);  
int QIsEmpty(Queue * pq);  
  
void Enqueue(Queue * pq, Data data);  
Data Dequeue(Queue * pq);  
Data QPeek(Queue * pq);
```

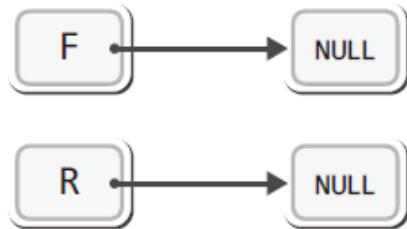
배열 기반의 구현보다 연결 리스트 기반의 구현에서
논의할 내용이 더 적다!

“스택과 큐의 유일한 차이점이 앞에서 꺼내느냐 뒤에서 꺼내느냐에 있
으니, 이전에 구현해 놓은 스택을 대상으로 꺼내는 방법만 조금 변경하
면 큐가 될 것 같다.”

연결 리스트 기반 큐의 경우!



연결 리스트 기반 큐의 구현: 초기화



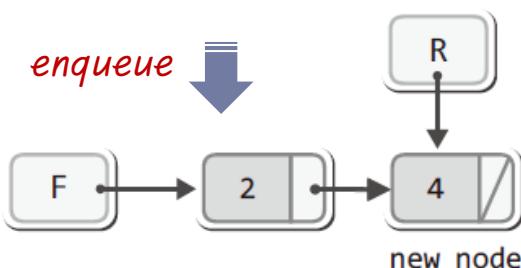
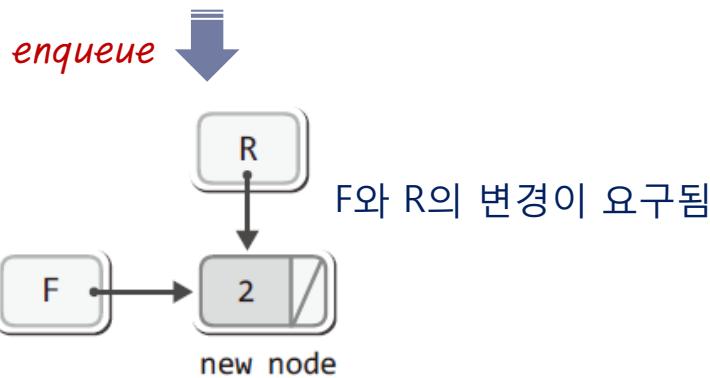
▶ [그림 07-12: 리스트 기반 큐의 초기상태]

```
void QueueInit(Queue * pq)
{
    pq->front = NULL;
    pq->rear = NULL;
}
```

```
int QIsEmpty(Queue * pq)
{
    if(pq->front == NULL)
        return TRUE;
    else
        return FALSE;
}
```



연결 리스트 기반 큐의 구현: enqueue



```
void Enqueue(Queue * pq, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->next = NULL;
    newNode->data = data;

    if(QIsEmpty(pq))
    {
        pq->front = newNode;
        pq->rear = newNode;
    }
    else
    {
        pq->rear->next = newNode;
        pq->rear = newNode;
    }
}
```

enqueue의 과정은 두 가지로 나뉜다!



연결 리스트 기반 큐의 구현: dequeue 논리

F가 다음 노드를 가리키게 하고, F가 이전에 가리키던 노드를 소멸시킨 결과!



마찬가지로! F가 다음 노드를 가리키게 하고, F가 이전에 가리키던 노드를 소멸시킨 결과!

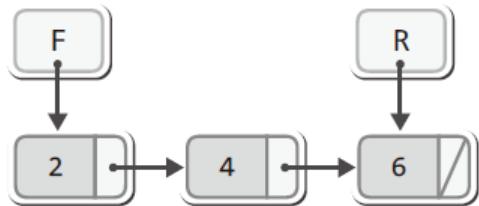


R은 그냥 내버려 둔다! 그래야 dequeue의 흐름을 하나로 유지할 수 있다!

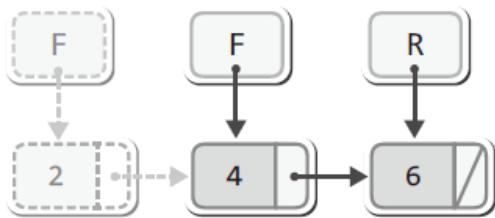
그리고 R은 그냥 내버려 두어도 된다!



연결 리스트 기반 큐의 구현: dequeue 정의



enqueue



```
Data Dequeue(Queue * pq)
{
    Node * delNode;
    Data retData;

    if(QIsEmpty(pq))
    {
        printf("Queue Memory Error!");
        exit(-1);
    }

    delNode = pq->front;
    retData = delNode->data;
    pq->front = pq->front->next;
    노드를 삭제한다. F가 다음 노드를 가리키게 하고!
    free(delNode);
    return retData;
}
```



연결 리스트 기반 큐의 실행

```
int main(void)
{
    // Queue의 생성 및 초기화 ///////
    Queue q;
    QueueInit(&q);

    // 데이터 넣기 ///////
    Enqueue(&q, 1); Enqueue(&q, 2);
    Enqueue(&q, 3); Enqueue(&q, 4);
    Enqueue(&q, 5);

    // 데이터 꺼내기 ///////
    while(!QIsEmpty(&q))
        printf("%d ", Dequeue(&q));

    return 0;
}
```

ListBaseQueue.h } 연결 리스트 기반 큐의 구현 결과
ListBaseQueue.c **ListBaseQueueMain.c** main 함수의 정의

1 2 3 4 5

실행결과



Chapter 07. 큐(Queue)



Chapter 07-4:

큐의 활용

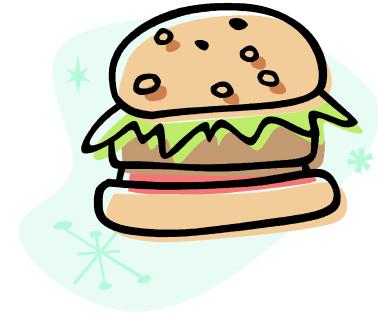


시뮬레이션의 주제

★ 점심시간 1시간 동안에는 고객이 15초당 1명씩 주문을 한다.

★ 종류별 햄버거를 만드는데 걸리는 시간은 다음과 같다.

- 치즈버거 12초
- 불고기버거 15초
- 더블버거 24초



시뮬레이션 할 상황!

이 상황에서 대기실의 크기를 결정하는데 필요한 정보를 추출하는 것이 목적!

- ✓ 수용인원이 30명인 공간 안정적으로 고객을 수용할 확률 50%
- ✓ 수용인원이 50명인 공간 안정적으로 고객을 수용할 확률 70%
- ✓ 수용인원이 100명인 공간 안정적으로 고객을 수용할 확률 90%
- ✓ 수용인원이 200명인 공간 안정적으로 고객을 수용할 확률 100%

시뮬레이션을 통해서 추출된 정보의 형태!



시뮬레이션 예제의 작성

- 점심시간은 1시간이고 그 동안 고객은 15초에 1명씩 주문을 하는 것으로 간주한다.
- 한 명의 고객은 하나의 버거 만을 주문한다고 가정한다.
- 주문하는 메뉴에는 가중치를 두지 않는다. 모든 고객은 무작위로 메뉴를 선택한다.
- 햄버거를 만드는 사람은 1명이다. 그리고 동시에 둘 이상의 버거가 만들어지지 않는다.
- 주문한 메뉴를 받을 다음 고객은 대기실에서 나와서 대기한다.

CircularQueue.h
CircularQueue.c
HambugerSim.c

실행결과1

```
Simulation Report!  
- Cheese burger: 80  
- Bulgogi burger: 72  
- Double burger: 88  
- Waiting room size: 100
```

실행결과2

```
Queue Memory Error!
```



Chapter 07. 큐(Queue)



Chapter 07-5:

덱(Deque)의 이해와 구현



덱의 이해

“덱은 앞으로도 뒤로도 넣을 수 있고, 앞으로도 뒤로도 뺄 수 있는 자료구조!”

- 앞으로 넣기
- 앞에서 빼기
- 뒤로 넣기
- 뒤에서 빼기

덱의 4가지 연산!

모든 연산은 Pair를 이루지 않는다!

개별적으로 연산 가능!

Deque는 double ended queue의 줄인 표현으로, 양쪽 방향으로 모두 입출력이 가능함을 의미한다. 그리고 스택과 큐의 특성을 모두 지니고 있다고도 말한다. 덱을 스택으로도 큐로도 활용할 수 있기 때문이다.



덱의 ADT

- void DequeInit(Deque * pdeq);

- 덱의 초기화를 진행한다.

- 덱 생성 후 제일 먼저 호출되어야 하는 함수이다.

- int DQIsEmpty(Deque * pdeq);

- 덱이 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.

앞으로 넣기

- void DQAddFirst(Deque * pdeq, Data data);

- 덱의 머리에 데이터를 저장한다. data로 전달된 값을 저장한다.

뒤로 넣기

- void DQAddLast(Deque * pdeq, Data data);

- 덱의 꼬리에 데이터를 저장한다. data로 전달된 값을 저장한다.

앞에서 빼기

- Data DQRemoveFirst(Deque * pdeq);

- 덱의 머리에 위치한 데이터를 반환 및 소멸한다.

뒤로 빼기

- Data DQRemoveLast(Deque * pdeq);

- 덱의 꼬리에 위치한 데이터를 반환 및 소멸한다.

앞에서 PEEK

- Data DQGetFirst(Deque * pdeq);

- 덱의 머리에 위치한 데이터를 소멸하지 않고 반환한다.

뒤에서 PEEK

- Data DQGetLast(Deque * pdeq);

- 덱의 꼬리에 위치한 데이터를 소멸하지 않고 반환한다.



덱의 구현: 헤더파일 정의

```
typedef int Data;  
  
typedef struct _node  
{  
    Data data;  
    struct _node * next;  
    struct _node * prev;  
} Node;  
  
typedef struct _dlDeque  
{  
    Node * head;  
    Node * tail;  
} DLDeque;
```

덱의 구현에 가장 어울리는 자료구조는 양방향 연결 리스트이다!

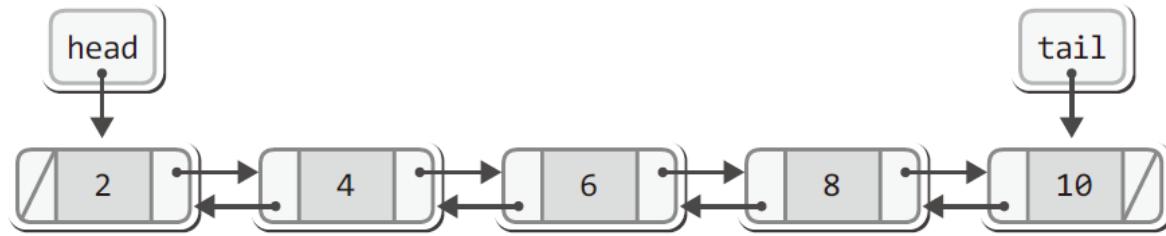
```
void DequeInit(D deque * pdeq);  
int DQIsEmpty(D deque * pdeq);  
void DQAddFirst(D deque * pdeq, Data data);  
void DQAddLast(D deque * pdeq, Data data);  
Data DQRemoveFirst(D deque * pdeq);  
Data DQRemoveLast(D deque * pdeq);  
Data DQGetFirst(D deque * pdeq);  
Data DQGetLast(D deque * pdeq);
```

앞과 뒤로 입출력 연산이 이뤄지기 때문에

head뿐만 아니라 tail도 정의되어야 한다.

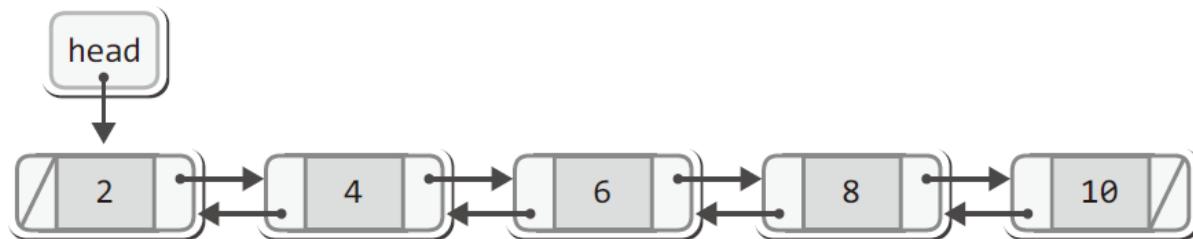


덱의 구현: 함수의 정의



덱의 구현을 위한 양방향 연결 리스트의 구조

이전에 구현한 양방향 연결 리스트와의 차이점은 tail의 존재 유무가 전부이니! 코드에 대한 해석은 여러분의 몫으로 남깁니다.



이전에 구현한 양방향 연결 리스트의 구조



수고하셨습니다~



Chapter 07에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 08. 트리(Tree)

Introduction To Data Structures Using C

Chapter 08. 트리(Tree)



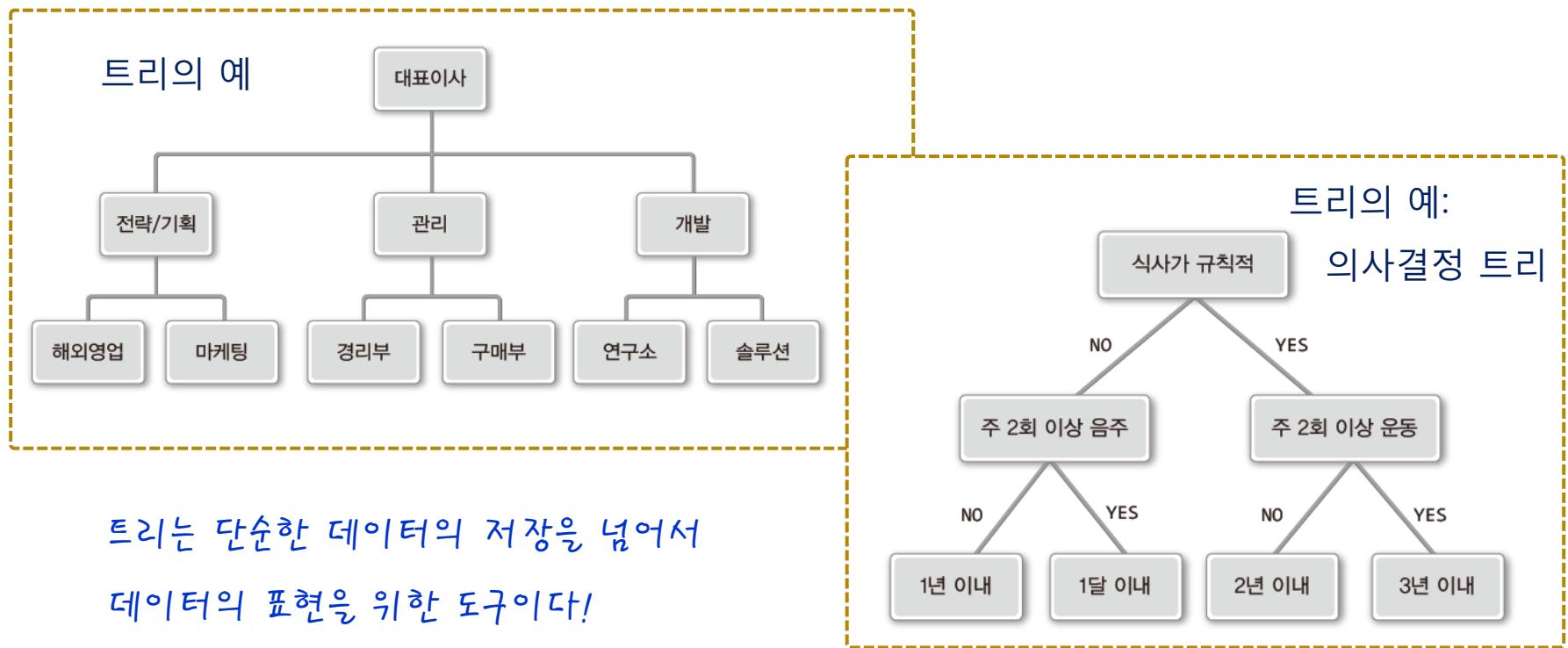
Chapter 08-1:

트리의 개요



트리의 접근과 이해

“트리는 계층적 관계(Hierarchical Relationship)를 표현하는 자료구조이다.”



트리 관련 용어의 소개

- 노드: node

트리의 구성요소에 해당하는 A, B, C, D, E, F와 같은 요소

- 간선: edge

노드와 노드를 연결하는 연결선

- 루트 노드: root node

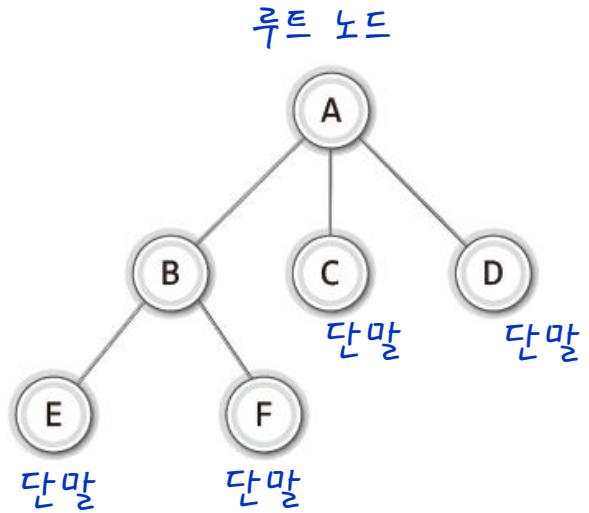
트리 구조에서 최상위에 존재하는 A와 같은 노드

- 단말 노드: terminal node

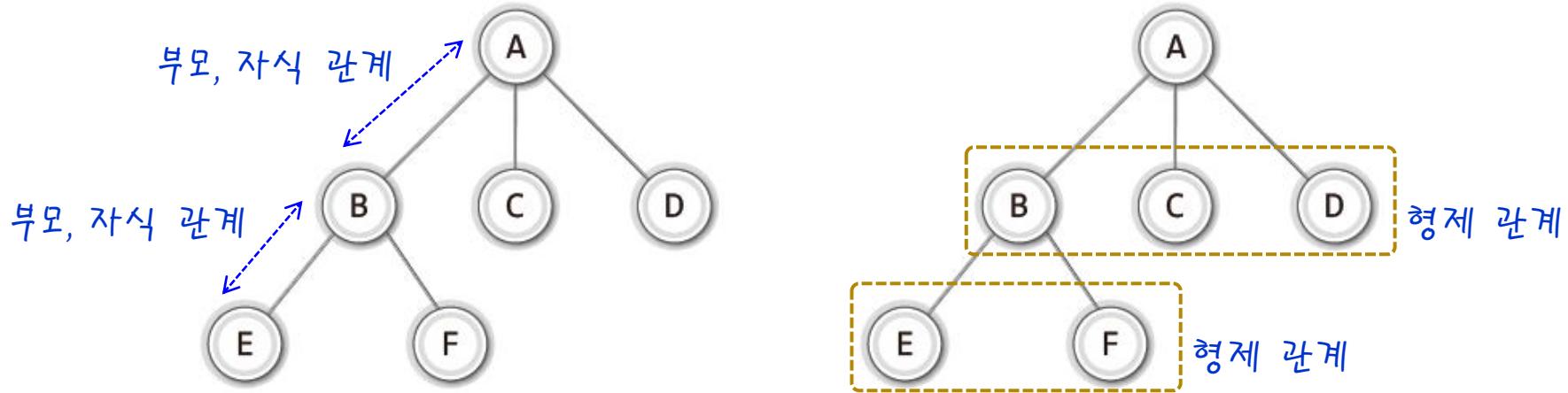
아래로 또 다른 노드가 연결되어 있지 않은 E, F, C, D와 같은 노드

- 내부 노드: internal node

단말 노드를 제외한 모든 노드로 A, B와 같은 노드



트리의 노드간 관계

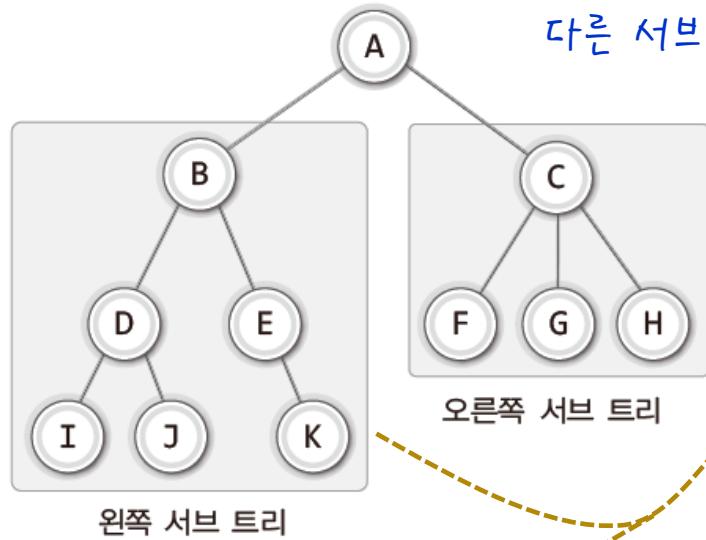


- 노드 A는 노드 B, C, D의 **부모 노드(parent node)**이다.
- 노드 B, C, D는 노드 A의 **자식 노드(child node)**이다.
- 노드 B, C, D는 부모 노드가 같으므로, 서로가 서로에게 **형제 노드(sibling node)**이다.

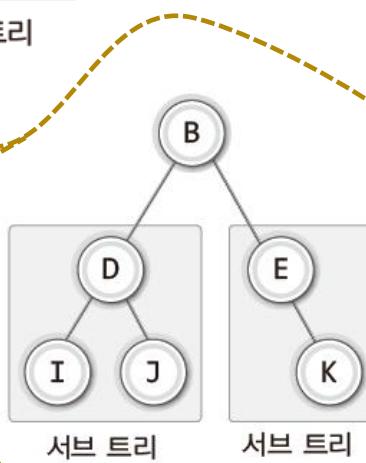


서브 트리의 이해

서브 트리 역시 서브 트리로 이뤄져 있으며, 그 서브 트리 역시 또 다른 서브 트리로 이뤄져 있다. 이렇듯 트리는 그 구조가 재귀적이다!



하나의 트리를 구성하는 왼쪽과 오른쪽의 작은 트리를 가리켜 서브 트리라 한다.



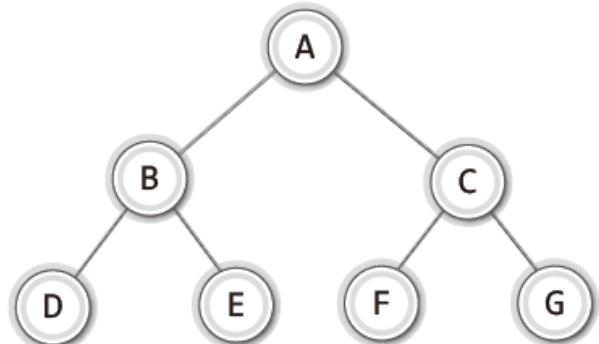
서브 트리 역시 또 다른 서브 트리를 갖는다!



이진 트리의 이해

이진 트리의 조건

- 루트 노드를 중심으로 두 개의 서브 트리로 나뉘어진다.
- 나뉘어진 두 서브 트리도 모두 이진 트리이어야 한다.



이진 트리의 예

“이진 트리가 되려면, 루트 노드를 중심으로 둘로 나뉘는 두 개의 서브 트리도 이진 트리이어야 하고, 그 서브 트리의 모든 서브 트리도 이진 트리이어야 한다.”

재귀적인 성향을 담아내지 못한 완전하지 않은 표현

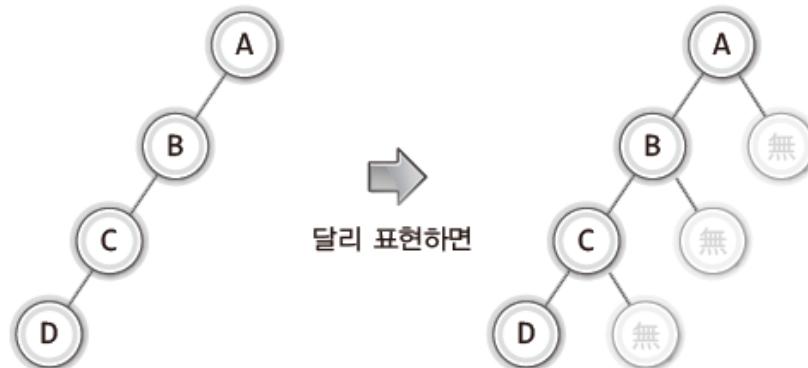
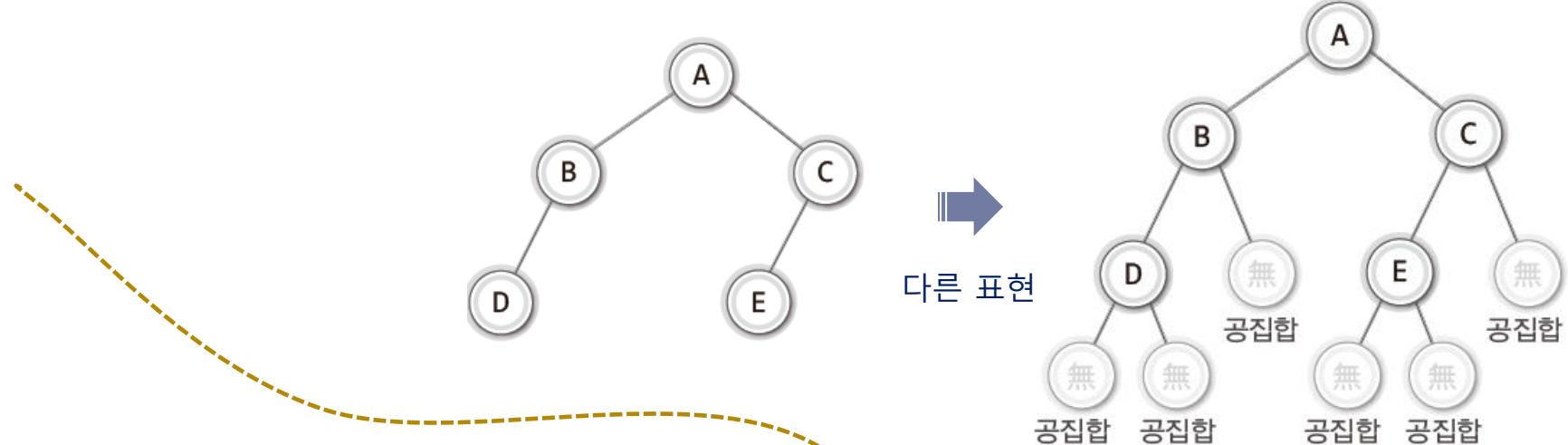
트리 그리고 이진 트리는 그 구조가 재귀적이다!

따라서 트리와 관련된 연산은 재귀적으로 사고하고 재귀적으로 구현할 필요가 있다!



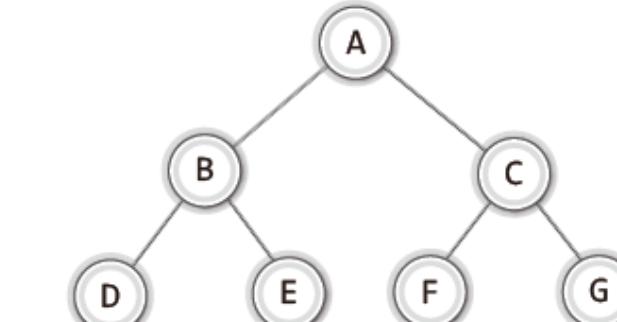
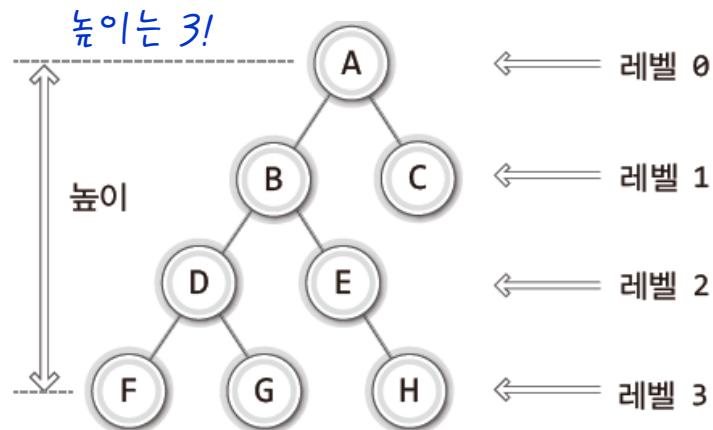
이진 트리와 공집합 노드

공집합(empty set)도 이진 트리에서는 노드로 간주한다!



하나의 노드에 두 개의 노드가 달리는
형태의 트리는 모두 이진 트리이다.

레벨과 높이, 그리고 포화, 완전 이진 트리

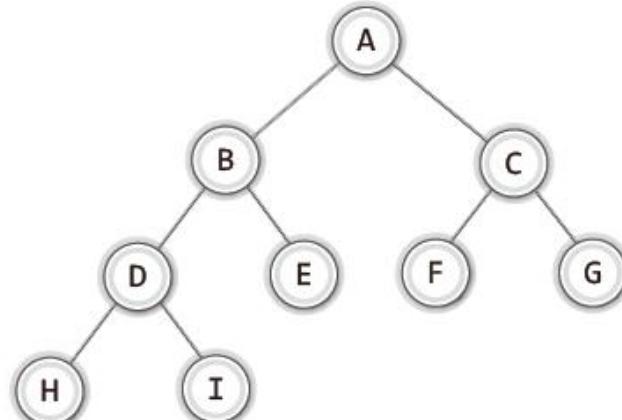


모든 레벨에 노드가 짝 찬! 포화 이진 트리

트리의 높이와 레벨의 최대 값은 같다!

완전 이진 트리는 위에서 아래로 왼쪽에서 오른쪽으로
채워진 트리를 의미한다.

따라서 포화 이진 트리는 동시에 완전 이진 트리이지만
그 역은 성립하지 않는다.



빈 틈 없이 차곡차곡 채워진! 완전 이진 트리



Chapter 08. 트리(Tree)

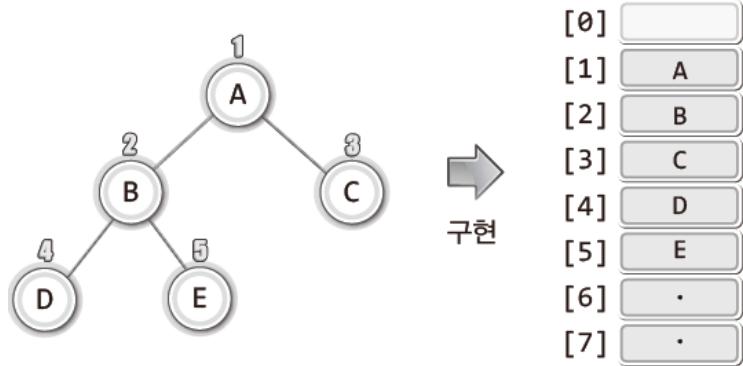


Chapter 08-2:

이진 트리의 구현

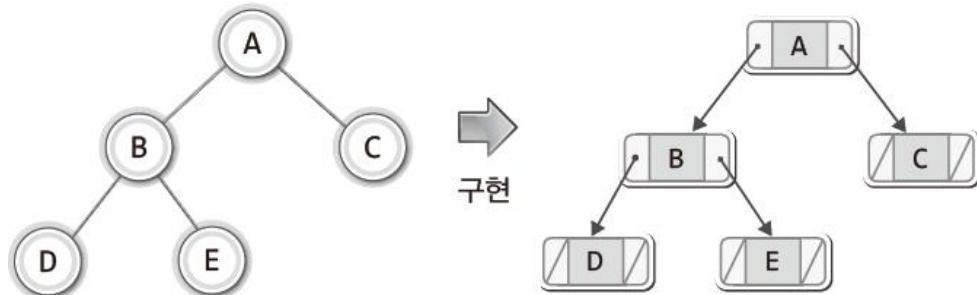


이진 트리 구현의 두 가지 방법



- 노드에 번호를 부여하고 그 번호에 해당하는 값을 배열의 인덱스 값으로 활용한다.
- 편의상 배열의 첫 번째 요소는 사용하지 않는다.

배열의 기본적인 장점! 접근이 용이하다라는 특성이 트리에서도 그대로 반영이 된다! 뿐만 아니라 배열을 기반으로 했을 때 완성하기 용이한 트리 관련 연산도 존재한다.



연결 리스트 기반에서는 트리의 구조와 리스트의 연결 구조가 일치한다.
따라서 구현과 관련된 직관적인 이해가 더 좋은 편이다.

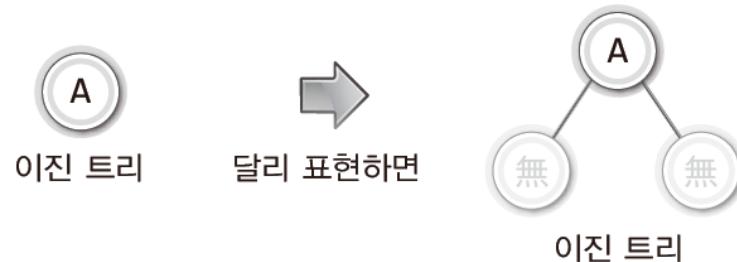


헤더파일에 정의된 구조체의 이해

```
// 이진 트리의 노드를 표현한 구조체
typedef struct _bTreeNode
{
    BTData data;
    struct _bTreeNode * left;
    struct _bTreeNode * right;
} BTreeNode;
```

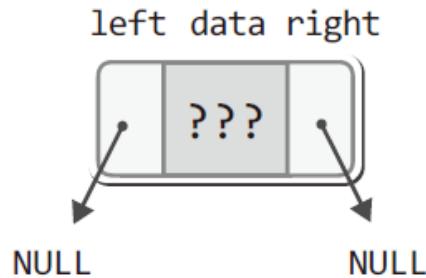
논리적으로도 하나의 노드는 그 자체로 이진 트리이다. 따라서 노드를 표현한 구조체는 실제로 이진 트리를 표현한 구조체가 된다.

이것이 노드이자 이진 트리를 표현한 구조체의 정의이다!
이진 트리의 모든 노드는 직/간접적으로 연결되어 있다.
따라서 루트 노드의 주소 값만 기억하면, 이진 트리 전체를 가리키는 것과 다름이 없다.



헤더파일에 선언된 함수들1

- BTTreeNode * MakeBTTreeNode(void); // 노드의 생성



이러한 형태의 노드를 동적으로 할당하여 생성한다!
유효한 데이터는 *SetData* 함수를 통해서 채워지 포인터
변수 *left*와 *right*는 *NULL*로 자동 초기화 된다.

- BTData GetData(BTTreeNode * bt); // 노드에 저장된 데이터를 반환

- void SetData(BTTreeNode * bt, BTData data); // 노드에 데이터를 저장

노드에 직접 접근하는 것보다. 함수를 통한 접근이 보다 칭찬받을 수 있는 구조!



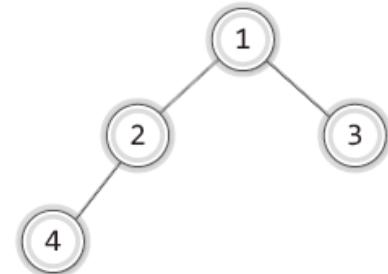
헤더파일에 선언된 함수들2

- BTreenode * GetLeftSubTree(BTreenode * bt);

왼쪽 서브 트리의 주소 값 반환!

- BTreenode * GetRightSubTree(BTreenode * bt);

오른쪽 서브 트리의 주소 값 반환!



- ✓ 루트 노드를 포함하여 어떠한 노드의 주소 값도 인자로 전달될 수 있다.
- ✓ 전달된 노드의 왼쪽, 오른쪽 '서브 트리의 루트 노드 주소 값' 또는 그냥 '노드의 주소 값'이 반환된다.

- void MakeLeftSubTree(BTreenode * main, BTreenode * sub);

main의 서브 왼쪽 서브 트리로 sub를 연결!

- void MakeRightSubTree(BTreenode * main, BTreenode * sub);

main의 오른쪽 서브 트리로 sub를 연결!

하나의 노드도 일종의 이진 트리이다! 따라서 위와 같이 함수를 이름을 짓는 것이 타당하다!

위의 함수들은 단순히 노드가 아니라 트리를 대상으로도 그 결과를 보인다는 사실을 기억하자!



정의된 함수들의 이해를 돋는 main 함수

```
int main(void)
{
    BTreenode * ndA = MakeBTreenode();      // 노드 A 생성
    BTreenode * ndB = MakeBTreenode();      // 노드 B 생성
    BTreenode * ndC = MakeBTreenode();      // 노드 C 생성

    ndB, ndB, ndC를 이용한 SetData 함수 호출을 통해 유용한 데이터 채운 후...

    // 노드 A의 왼쪽 자식 노드로 노드 B 연결
    MakeLeftSubTree(ndA, ndB);

    // 노드 A의 오른쪽 자식 노드로 노드 C 연결
    MakeRightSubTree(ndA, ndC);

    ...
}
```

앞서 정의한 이진 트리 관련 함수들은 이진 트리를 만드는 도구이다!



이진 트리의 구현

```
BTreeNode * MakeBTreeNode(void)
{
    BTreeNode * nd = (BTreeNode*)malloc(sizeof(BTreeNode));
    nd->left = NULL;
    nd->right = NULL;
    return nd;
}

BTData GetData(BTreeNode * bt)
{
    return bt->data;
}

void SetData(BTreeNode * bt, BTData data)
{
    bt->data = data;
}

BTreeNode * GetLeftSubTree(BTreeNode * bt)
{
    return bt->left;
}
```

구현 자체는 크게 어려움이 없다!

```
BTreeNode * GetRightSubTree(BTreeNode * bt)
{
    return bt->right;
}

void MakeLeftSubTree(BTreeNode * main, BTreeNode * sub)
{
    if(main->left != NULL)
        free(main->left);           기존에 연결된 노드는

    main->left = sub;            삭제되게 구현!

}

void MakeRightSubTree(BTreeNode * main, BTreeNode * sub)
{
    if(main->right != NULL)
        free(main->right);       기존에 연결된 노드는

    main->right = sub;          삭제되게 구현!
}
```

삭제되게 구현!



이진 트리 관련 main 함수

```
int main(void)
{
    BTTreeNode * bt1 = MakeBTTreeNode();          // 노드 bt1 생성
    BTTreeNode * bt2 = MakeBTTreeNode();          // 노드 bt2 생성
    BTTreeNode * bt3 = MakeBTTreeNode();          // 노드 bt3 생성
    BTTreeNode * bt4 = MakeBTTreeNode();          // 노드 bt4 생성

    SetData(bt1, 1);   // bt1에 1 저장
    SetData(bt2, 2);   // bt2에 2 저장
    SetData(bt3, 3);   // bt3에 3 저장
    SetData(bt4, 4);   // bt4에 4 저장

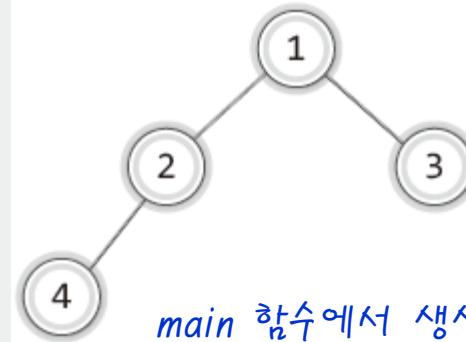
    MakeLeftSubTree(bt1, bt2);                  // bt2를 bt1의 왼쪽 자식 노드로
    MakeRightSubTree(bt1, bt3);                 // bt3를 bt1의 오른쪽 자식 노드로
    MakeLeftSubTree(bt2, bt4);                  // bt4를 bt2의 왼쪽 자식 노드로

    // bt1의 왼쪽 자식 노드의 데이터 출력
    printf("%d \n", GetData(GetLeftSubTree(bt1)));

    // bt1의 왼쪽 자식 노드의 왼쪽 자식 노드의 데이터 출력
    printf("%d \n", GetData(GetLeftSubTree(GetLeftSubTree(bt1))));

    return 0;
}
```

트리를 완전히 소멸시키는 방법은? ↪ 순회!



main 함수에서 생성하는 트리

BinaryTree.h
BinaryTree.c
BinaryTreeMain.c

실행결과

2
4

Chapter 08. 트리(Tree)



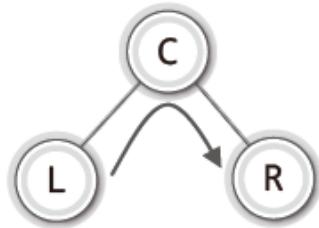
Chapter 08-3:

이진 트리의 순회(Traversal)



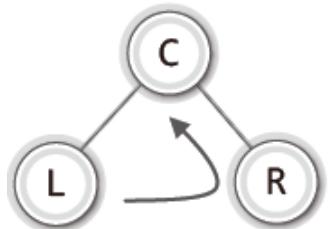
순회의 세 가지 방법

기준은 루트 노드를 언제 방문하느냐에 있다!



즉 루트 노드를 방문하는 시점에 따라서 중위, 후위, 전위 순회로 나뉘어 진다.

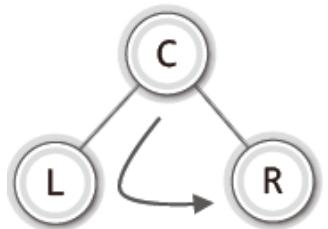
▶ [그림 08-21: 중위 순회]



▶ [그림 08-22: 후위 순회]

높이가 2 이상인 트리의 순회도 이와 다르지 않다.

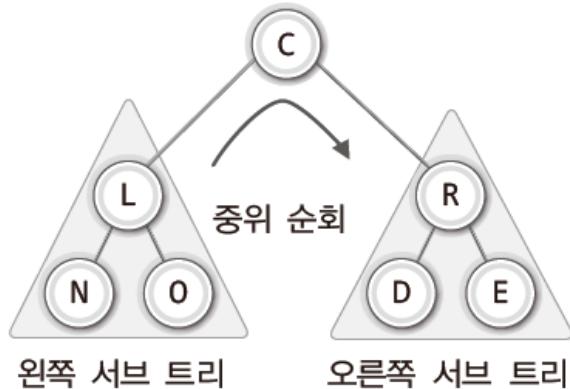
재귀적인 형태로 순회의 과정을 구성하면 높이에 상관없이 순회가 가능하다!



▶ [그림 08-23: 전위 순회]



순회의 재귀적 표현



▶ [그림 08-24: 이진 트리의 중위 순회]

- 1단계 왼쪽 서브 트리의 순회
- 2단계 루트 노드의 방문
- 3단계 오른쪽 서브 트리의 순회

재귀적 표현



```
void InorderTraverse(BTreeNode * bt)
{
    InorderTraverse(bt->left);
    printf("%d \n", bt->data);
    InorderTraverse(bt->right);
}
```

탈출 조건이 명시되지 않은 불완전한 구현



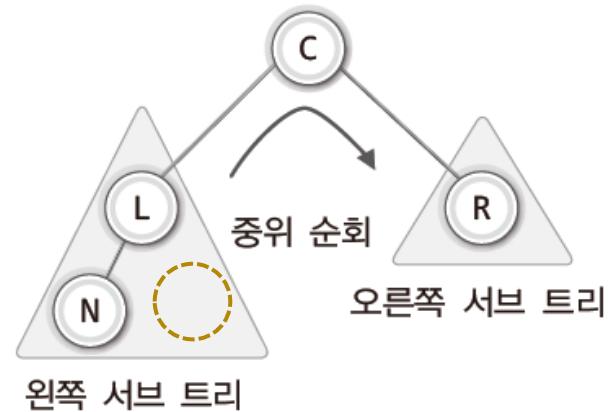
순회의 재귀적 표현 완성!

```
void InorderTraverse(BTreeNode * bt)
{
    if(bt == NULL)      // bt가 NULL이면 재귀 탈출!
        return;

    InorderTraverse(bt->left);
    printf("%d \n", bt->data);
    InorderTraverse(bt->right);
}
```

적용 가능

노드가 단말 노드인 경우,
단말 노드의 자식 노드는 *NULL*이다!



지금까지의 결과물 실행

```
int main(void)
{
    BTreeNode * bt1 = MakeBTreeNode();
    BTreeNode * bt2 = MakeBTreeNode();
    BTreeNode * bt3 = MakeBTreeNode();
    BTreeNode * bt4 = MakeBTreeNode();

    SetData(bt1, 1);
    SetData(bt2, 2);
    SetData(bt3, 3);
    SetData(bt4, 4);

    MakeLeftSubTree(bt1, bt2);
    MakeRightSubTree(bt1, bt3);
    MakeLeftSubTree(bt2, bt4);

    InorderTraverse(bt1);
    return 0;
}
```

BinaryTree.h

BinaryTree.c

BinaryTreeTraverseMain.c

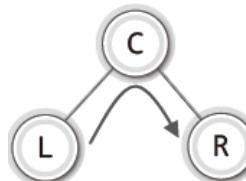
실행결과

4	
2	
1	
3	

전위 순회와 후위 순회

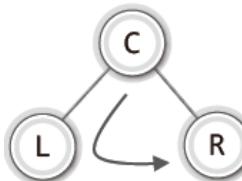
```
void InorderTraverse(BTreeNode * bt)
{
    if(bt == NULL) // bt가 NULL이면 재귀 탈출!
        return;

    InorderTraverse(bt->left);
    printf("%d \n", bt->data); 중위 순회
    InorderTraverse(bt->right);
}
```



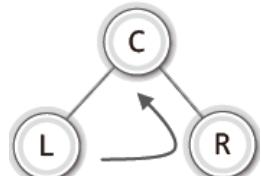
```
void PreorderTraverse(BTreeNode * bt)
{
    if(bt == NULL)
        return;

    printf("%d \n", bt->data); 전위 순회
    PreorderTraverse(bt->left);
    PreorderTraverse (bt->right);
}
```



```
void PostorderTraverse(BTreeNode * bt)
{
    if(bt == NULL)
        return;

    PostorderTraverse(bt->left);
    PostorderTraverse(bt->right);
    printf("%d \n", bt->data); 후위 순회
}
```



노드의 방문 이유! 자유롭게 구성하기!

(*VisitFuncPtr)로 대신해도 됩니다.

```
typedef void VisitFuncPtr(BTData data);
```

함수 포인터 형 VisitFuncPtr의 정의

```
void InorderTraverse(BTreeNode * bt, VisitFuncPtr action)
{
    if(bt == NULL)
        return;

    InorderTraverse(bt->left, action);
    action(bt->data);      // 노드의 방문
    InorderTraverse(bt->right, action);
}
```

action이 가리키는 함수를 통해서 방문을 진행~

```
void ShowIntData(int data)
{
    printf("%d ", data);
}
```

VisitFuncPtr형을 기준으로 정의된 함수



Chapter 08. 트리(Tree)



Chapter 08-4:

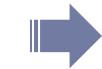
수식 트리(Expression Tree)의 구현



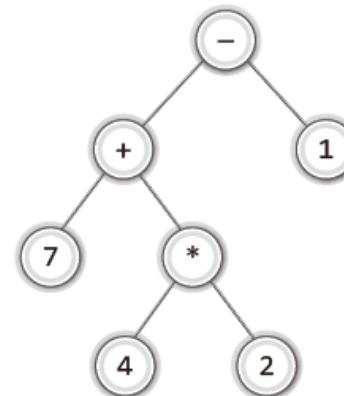
수식 트리의 이해

중위 표기법의 수식을 수식 트리로 변환하는 프로그램의 작성이 목적!

```
int main(void)
{
    int result = 0;
    result = 7 + 4 * 2 - 1;
    ...
}
```



수식 트리

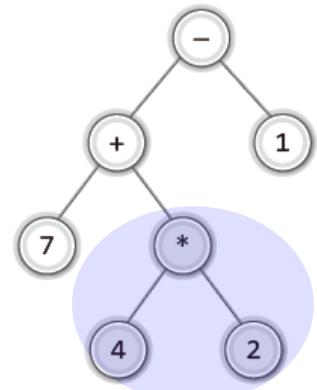


두 개의 자식 노드에는 피연산자를!

- 중위 표기법의 수식은 사람이 인식하기 좋은 수식이다. 컴퓨터의 인식에는 어려움이 있다.
- 그래서 컴파일러는 중위 표기법의 수식을 '수식 트리'로 재구성한다.
- 수식 트리는 해석이 쉽다. 연산의 과정에서 우선순위를 고려하지 않아도 된다!

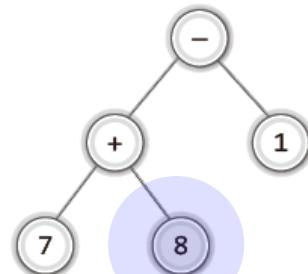


수식 트리의 계산과정

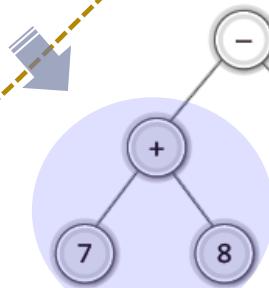


▶ [그림 08-27: 수식 트리의 연산과정 1/3]

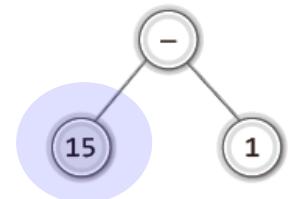
곱셈 연산 후



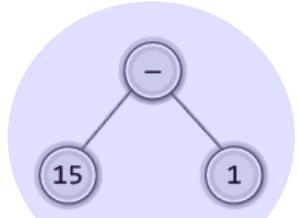
두 개의 자식 노드가 피연산자라는 단순하지만 전부인 하나의 특성을 근거로 연산이 매우 쉽게 진행된다!



덧셈 연산 후



▶ [그림 08-28: 수식 트리의 연산과정 2/3]



뺄셈 연산 후

▶ [그림 08-29: 수식 트리의 연산과정 3/3]



수식 트리를 만드는 절차!

Ch06의 ConvToRPNExp 함수에서 구현

중위 표기법의 수식



후위 표기법의 수식



수식 트리

그래서 후위 표기법의 수식을 수식 트리로 구성하는 방법만 고민!

중위 표기법의 수식을 바로 수식 트리로 표현하는 것은 쉽지 않다.

하지만 일단 후위 표기법의 수식으로 변경한 다음에 수식 트리로 표현하는 것은 어렵지 않다!

앞서 구현한 필요한 도구들!

· 수식 트리 구현에 필요한 이진 트리

BinaryTree2.h, BinaryTree2.c

· 수식 트리 구현에 필요한 스택

ListBaseStack.h, ListBaseStack.c



수식 트리의 구현과 관련된 헤더파일

트리 만드는 도구를 기반으로 함수를 정의한다!

```
#include "BinaryTree2.h"
```

```
BTreeNode * MakeExpTree(char exp[]); // 수식 트리 구성
```

후위 표기법의 수식을 인자로 받아서 수식 트리를 구성하고

루트 노드의 주소 값을 반환한다!

```
int EvaluateExpTree(BTreeNode * bt); // 수식 트리 계산
```

MakeExpTree가 구성한 수식 트리의 수식을 계산하여 그 결과를 반환한다!

```
void ShowPrefixTypeExp(BTreeNode * bt); // 전위 표기법 기반 출력
```

```
void ShowInfixTypeExp(BTreeNode * bt); // 중위 표기법 기반 출력
```

```
void ShowPostfixTypeExp(BTreeNode * bt); // 후위 표기법 기반 출력
```

전위, 중위, 후위 순회하여 출력 시

각각 전위, 중위, 후위 표기법의 수식이 출력된다.



수식 트리의 구성 방법: 그림상 이해하기



후위 표기법의 수식에서 먼저 등장하는 피연산자와 연산자를 이용해서 트리의 하단부
터 구성해 나가고 이어서 점진적으로 윗부분을 구성해 나간다.

이 사실을 이해하는 것과 이를 코드로 옮기는 것은 다소 다른 문제이다!



수식 트리의 구성 방법: 코드로 옮기기1



Me? 쟁반!

▶ [그림 08-31: 수식 트리의 구성 1/7]

피연산자는 무조건 스택으로!



2
1

Me? 쟁반!

▶ [그림 08-33: 수식 트리의 구성 3/7]

연산자 만나면 스택에서

피연산자 두 개 꺼내어 트리 구성!

2
1

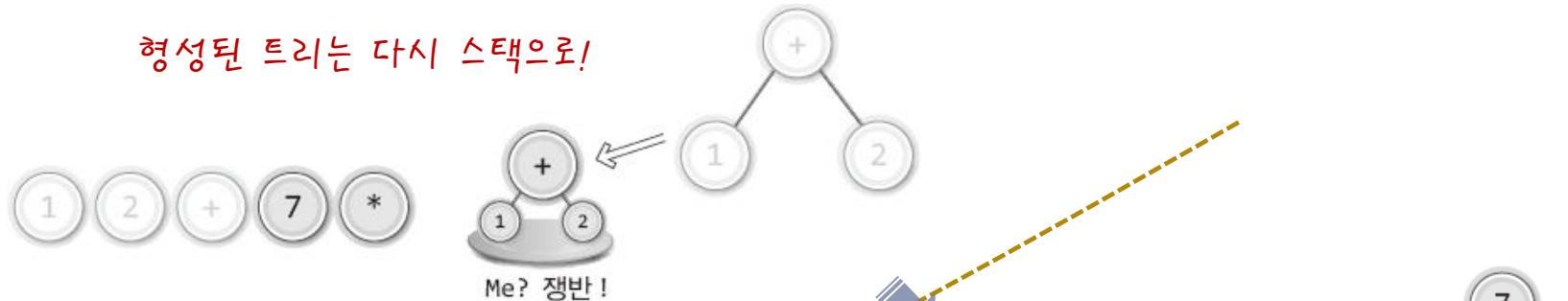
Me? 쟁반!

▶ [그림 08-34: 수식 트리의 구성 4/7]

1

Me? 쟁반!

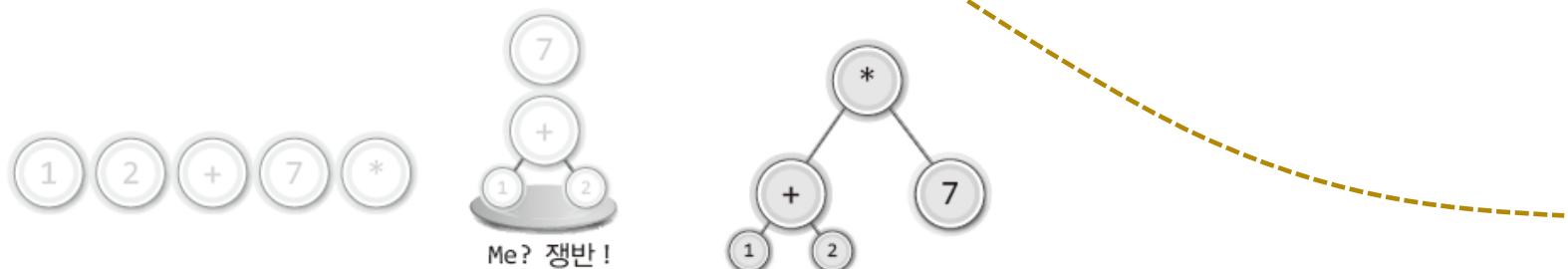
수식 트리의 구성 방법: 코드로 옮기기2



▶ [그림 08-35: 수식 트리의 구성 5/7]



▶ [그림 08-36: 수식 트리의 구성 6/7]



▶ [그림 08-37: 수식 트리의 구성 7/7]

수식 트리의 구성 방법: 코드로 옮기기3

```
BTreeNode * MakeExpTree(char exp[])
{
    Stack stack;
    BTreeNode * pnode;
    int expLen = strlen(exp);
    int i;
    for(i=0; i<expLen; i++)
    {
        pnode = MakeBTreeNode();
        if(isdigit(exp[i])) {           // 피연산자라면...
            SetData(pnode, exp[i]-'0'); // 문자를 정수로 바꿔서 저장
        }
        else {                         // 연산자라면...
            MakeRightSubTree(pnode, SPop(&stack));
            MakeLeftSubTree(pnode, SPop(&stack));
            SetData(pnode, exp[i]);
        }
        SPush(&stack, pnode);
    }
    return SPop(&stack);
}
```

- 피연산자는 스택으로 옮긴다.
- 연산자를 만나면 스택에서 두 개의 피연산자 꺼내어 자식 노드로 연결!
- 자식 노드를 연결해서 만들어진 트리는 다시 스택으로 옮긴다.



수식 트리의 순회: 그 결과

- | | |
|-----------------------|------------|
| · 전위 순회하여 데이터를 출력한 결과 | 전위 표기법의 수식 |
| · 중위 순회하여 데이터를 출력한 결과 | 중위 표기법의 수식 |
| · 후위 순회하여 데이터를 출력한 결과 | 후위 표기법의 수식 |

수식 트리를 구성하면, 전위, 중위, 후위 표기법으로의 수식 표현이 쉬워진다.

그리고 전회, 중위, 후위 순회하면서 출력되는 결과물을 통해서 MakeExpTree 함수를 검증할 수 있다.



수식 트리의 순회: 방법

```
void ShowPrefixTypeExp(BTreeNode * bt)
{
    PreorderTraverse(bt, ShowNodeData);
} 전위 표기법 수식 출력

void ShowInfixTypeExp(BTreeNode * bt)
{
    InorderTraverse(bt, ShowNodeData);
} 중위 표기법 수식 출력

void ShowPostfixTypeExp(BTreeNode * bt)
{
    PostorderTraverse(bt, ShowNodeData);
} 후위 표기법 수식 출력
```

VisitFuncPtr형 함수

```
void ShowNodeData(int data)
{
    if(0<=data && data<=9)
        printf("%d ", data); // 피연산자 출력
    else
        printf("%c ", data); // 연산자 출력
}
```



수식 트리 관련 예제의 실행

ListBaseStack.h의 type 선언 변경 필요하다!

```
typedef BTTreeNode * BTData;
```

```
int main(void)
{
    char exp[] = "12+7*";
    BTTreeNode * eTree = MakeExpTree(exp);

    printf("전위 표기법의 수식: ");
    ShowPrefixTypeExp(eTree); printf("\n");

    printf("중위 표기법의 수식: ");
    ShowInfixTypeExp(eTree); printf("\n");

    printf("후위 표기법의 수식: ");
    ShowPostfixTypeExp(eTree); printf("\n");

    printf("연산의 결과: %d \n", EvaluateExpTree(eTree));

    return 0;
}
```

- 이진 트리 관련
BinaryTree2.h, BinaryTree2.c
- 스택 관련
ListBaseStack.h, ListBaseStack.c
- 수식 트리 관련
ExpressionTree.h, ExpressionTree.c
- main 함수 관련
ExpressionMain.c

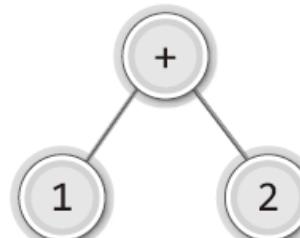
실행결과

```
전위 표기법의 수식: * + 1 2 7
중위 표기법의 수식: 1 + 2 * 7
후위 표기법의 수식: 1 2 + 7 *
연산의 결과: 21
```



수식 트리의 계산: 기본 구성

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;
    op1 = GetData(GetLeftSubTree(bt));           // 첫 번째 피연산자
    op2 = GetData(GetRightSubTree(bt));          // 두 번째 피연산자
    switch(GetData(bt))                         // 연산자를 확인하여 연산을 진행
    {
        case '+':
            return op1+op2;
        case '-':
            return op1-op2;
        case '*':
            return op1*op2;
        case '/':
            return op1/op2;
    }
    return 0;
}
```



이 모델을 대상으로 한 구현

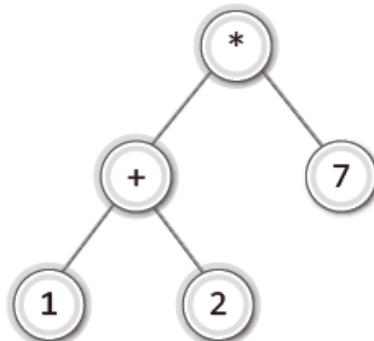


수식 트리의 계산: 재귀적 구성

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;
    op1 = GetData(GetLeftSubTree(bt));           // 첫 번째 피연산자
    op2 = GetData(GetRightSubTree(bt));          // 두 번째 피연산자
    switch(GetData(bt))                         // 연산자를 확인하여 연산을 진행
    {
        case '+':
            return op1+op2;
        case '-':
            return op1-op2;
        case '*':
            return op1*op2;
        case '/':
            return op1/op2;
    }
    return 0;
}
```

// 재귀적 구성

```
op1 = EvaluateExpTree(GetLeftSubTree(bt));
op2 = EvaluateExpTree(GetRightSubTree(bt));
```



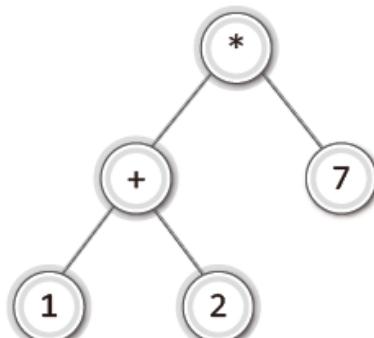
이 모델을 대상으로 한 구현
단! 단말노드에 대해서는 고려되지 않았다!



수식 트리의 계산:

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;
    if(GetLeftSubTree(bt)==NULL && GetRightSubTree(bt)==NULL) // 단말 노드라면
        return GetData(bt);

    op1 = EvaluateExpTree(GetLeftSubTree(bt));
    op2 = EvaluateExpTree(GetRightSubTree(bt));
    switch(GetData(bt))
    {
        .... 이전과 동일 ...
    }
    return 0;
}
```



탈출 조건!

이 모델을 대상으로 한 구현
단말노드는 탈출의 조건이다!



수고하셨습니다~



Chapter 08에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 09. 우선순위 큐와 힙

Introduction To Data Structures Using C

Chapter 09. 우선순위 큐와 힙



Chapter 09-1:

우선순위 큐의 이해



우선순위 큐와 우선순위의 이해

일반 큐의 두 가지 연산

- enqueue 큐에 데이터를 삽입하는 행위
 - dequeue 큐에서 데이터를 꺼내는 행위

들어간 순서를 근거로 dequeue 연산이 진행된다.

우선순위 큐의 두 가지 연산

- enqueue 우선순위 큐에 데이터를 삽입하는 행위
 - dequeue 우선순위 큐에서 데이터를 꺼내는 행위

들어간 순서에 상관 없이 우선순위를 근거로 dequeue 연산이 진행된다.

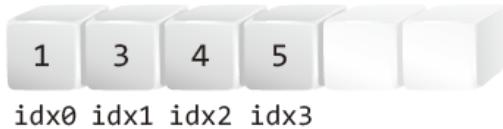
데이터 별 우선순위의 비교기준은 프로그래머가 결정할 뜻이다! 따라서 우선순위 큐 자료구조를 활용하는 프로그래머가 직접 우선순위 비교기준을 결정할 수 있도록 구현이 되어야 한다.



우선순위 큐의 구현 방법

우선순위 큐를 구현하는 세 가지 방법

- 배열을 기반으로 구현하는 방법
- 연결 리스트를 기반으로 구현하는 방법
- 힙(heap)을 이용하는 방법



두 가지 방법 모두 최악의 경우 새 데이터의 위치를 찾기 위해서 기존에 저장된 모든 데이터와 비교를 진행해야 한다.

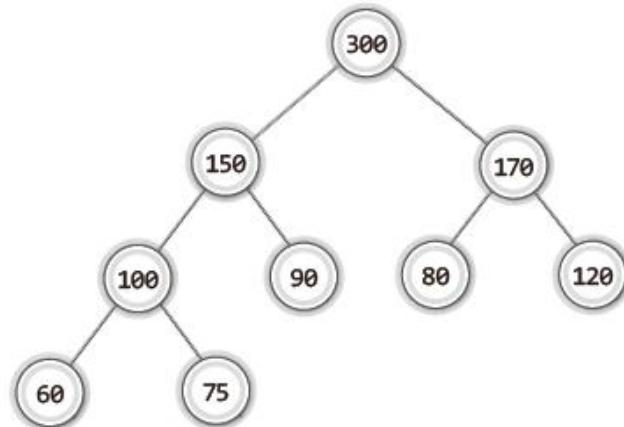


▶ [그림 09-1: 단순 배열과 연결 리스트 기반의 우선순위 큐 모델]

우선순위를 근거로 적정 위치를 찾아서 데이터를 저장하는 방식



힙(Heap)의 소개

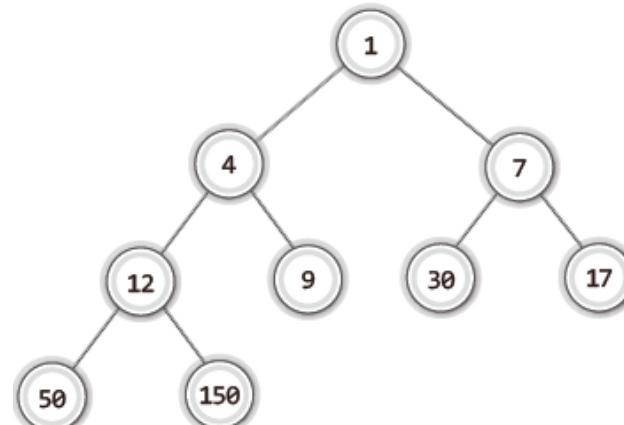


모든 노드에 저장된 값은 자식 노드에 저장된 값보다 크거나 같아야 한다.

즉 루트 노드에 저장된 값이 가장 커야 한다.

▶ [그림 09-2: 최대 힙(max heap)]

힙은 '완전 이진 트리'이다!



모든 노드에 저장된 값은 자식 노드에 저장된 값보다 크거나 같아야 한다.

즉 루트 노드에 저장된 값이 가장 커야 한다.

▶ [그림 09-3: 최소 힙(min heap)]



Chapter 09. 우선순위 큐와 힙

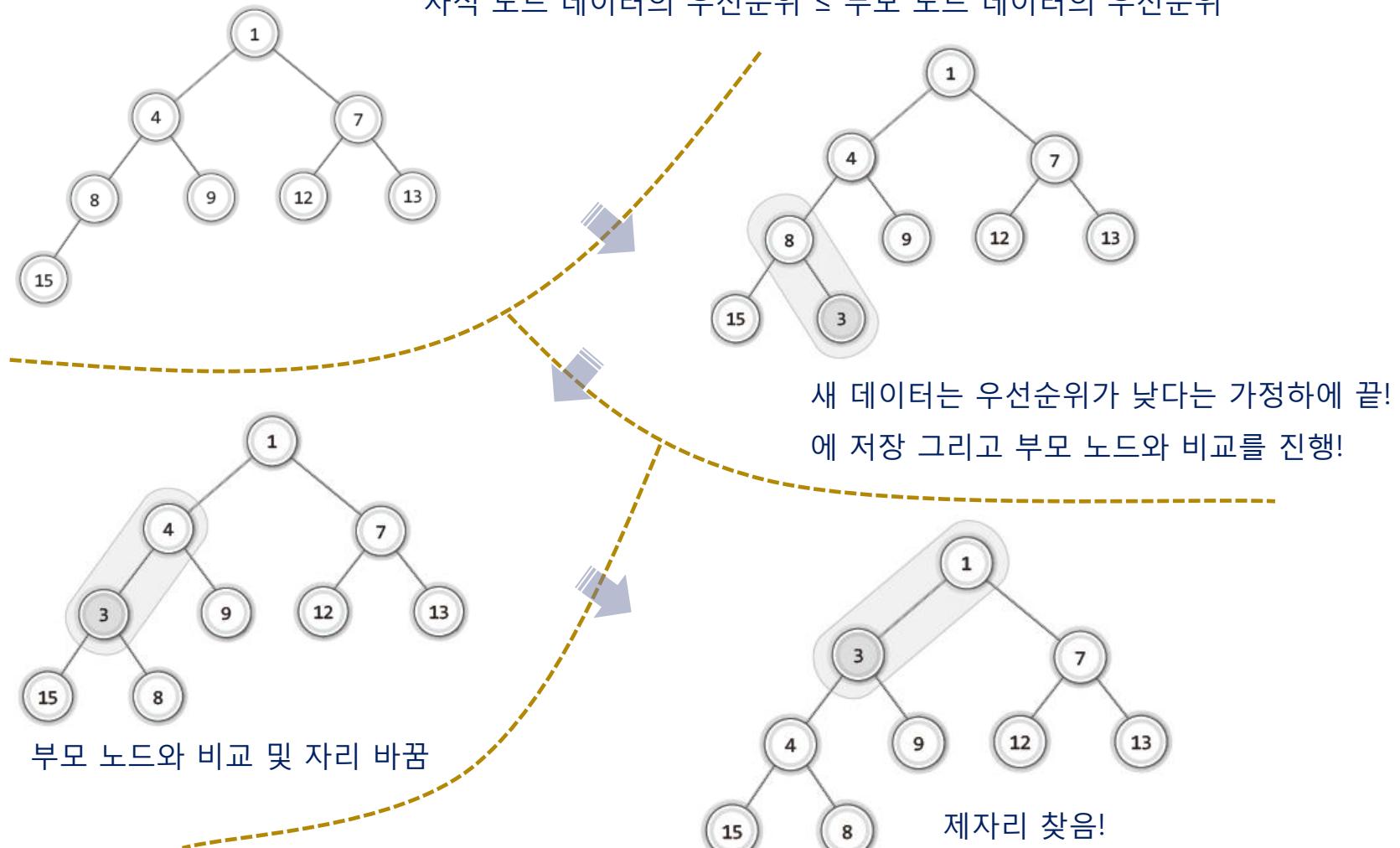


Chapter 09-2:

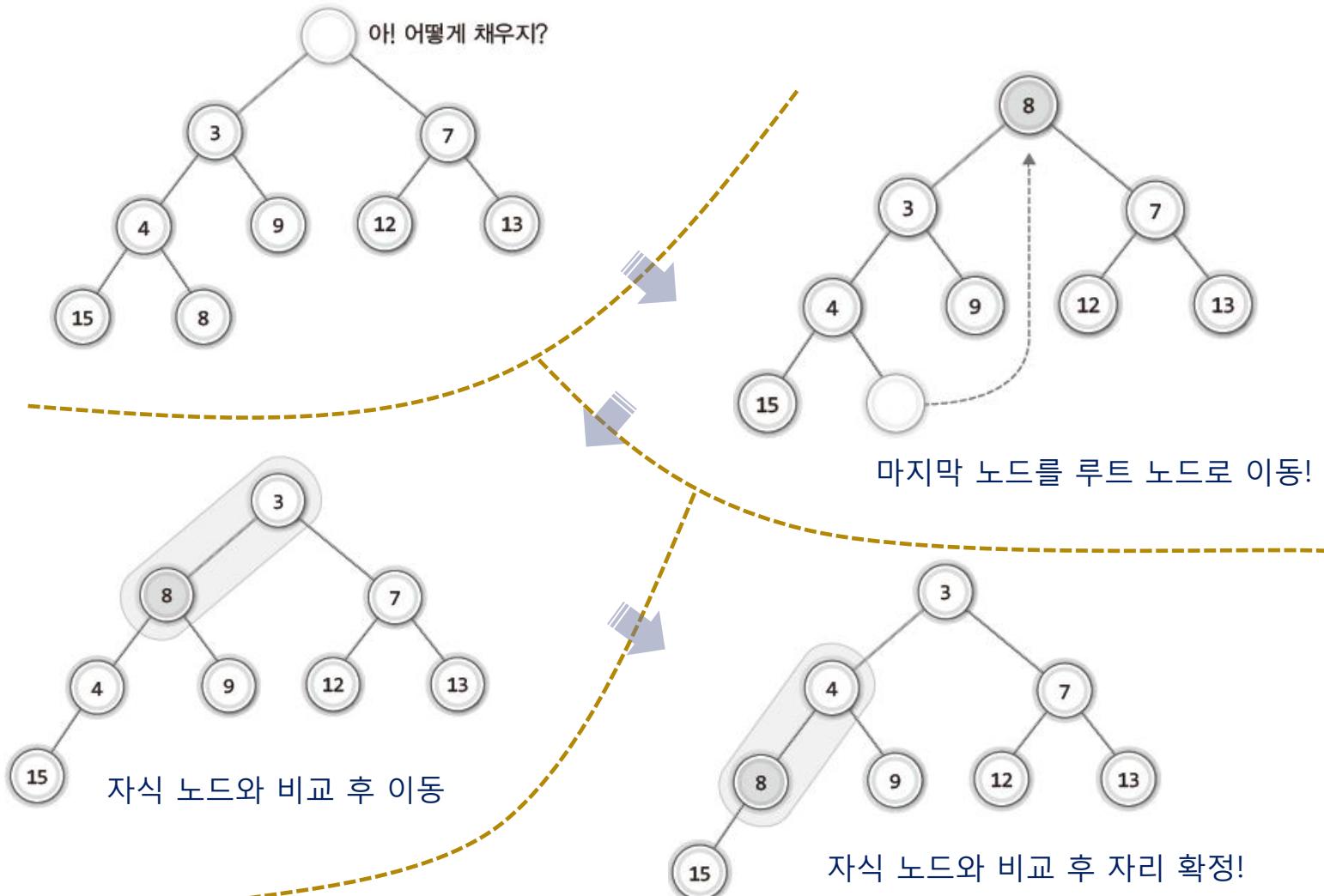
힙의 구현과 우선순위 큐의 완성



힙에서의 데이터 저장과정



힙에서의 데이터 삭제과정



삽입과 삭제의 과정에서 보인 성능의 평가

배열 기반 우선순위 큐의 시간 복잡도

- 배열 기반 데이터 삽입의 시간 복잡도 $O(n)$
- 배열 기반 데이터 삭제의 시간 복잡도 $O(1)$

연결 리스트 기반 우선순위 큐의 시간 복잡도

- 연결 리스트 기반 데이터 삽입의 시간 복잡도 $O(n)$
- 연결 리스트 기반 데이터 삭제의 시간 복잡도 $O(1)$

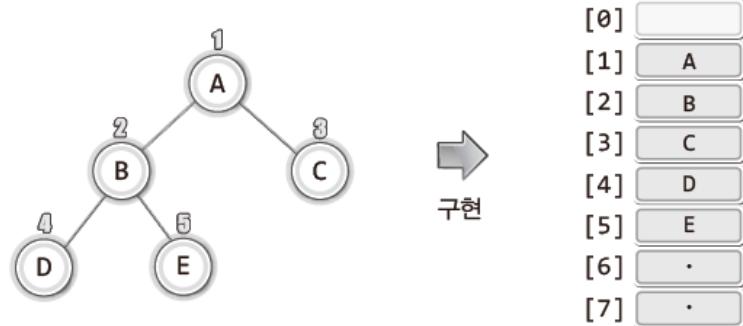
힙 기반 우선순위 큐의 시간 복잡도

- 힙 기반 데이터 삽입의 시간 복잡도 $O(\log_2 n)$
- 힙 기반 데이터 삭제의 시간 복잡도 $O(\log_2 n)$

단! 여기서 말하는 '완전 이진 트리'인 힙은 배열을 기반으로 구현한다!



배열을 기반으로 힙을 구현하는데 필요한 지식들



"연결 리스트를 기반으로 힙을 구현하면, 새로운 노드를 힙의 '마지막 위치'에 추가하는 것이 쉽지 않다."

배열 기반에서 인덱스 값 구하기!

- 왼쪽 자식 노드의 인덱스 값
- 오른쪽 자식 노드의 인덱스 값
- 부모 노드의 인덱스 값

부모 노드의 인덱스 값 $\times 2$

부모 노드의 인덱스 값 $\times 2 + 1$

자식 노드의 인덱스 값 $\div 2$

나눗셈은 정수형 나눗셈



원리 이해 중심의 힙 구현: 헤더파일의 소개

```
typedef char HData;  
typedef int Priority;
```

```
typedef struct _heapElem  
{  
    Priority pr; // 값이 작을수록 높은 우선순위  
    HData data;  
} HeapElem;
```

데이터와 우선순위 정보를 각각 구분하였음에 주목!
이것이 옳은 것만은 아니다!

```
typedef struct _heap  
{  
    int numOfData;  
    HeapElem heapArr[HEAP_LEN];  
} Heap;
```

힙을 구현하는 것으로 함수의 이름이 enqueue, dequeue가
아니다!

```
void HeapInit(Heap * ph);  
int HIsEmpty(Heap * ph);
```

우선순위 큐의 구현을 목적으로 하는 힙의 헤더파일 정의!

```
void HInsert(Heap * ph, HData data, Priority pr);
```

```
HData HDelete(Heap * ph);
```

우선순위가 가장 높은 데이터 삭제도록 정의!



원리 이해 중심의 힙 구현: 숙지할 내용

- 힙은 완전 이진 트리이다.
- 힙의 구현은 배열을 기반으로 하며 인덱스가 0인 요소는 비워둔다.
- 따라서 힙에 저장된 노드의 개수와 마지막 노드의 고유번호는 일치한다.
- 노드의 고유번호가 노드가 저장되는 배열의 인덱스 값이 된다.
- 우선순위를 나타내는 정수 값이 작을수록 높은 우선순위를 나타낸다고 가정한다

배열을 기반으로 하는 경우! 힙에 저장된 노드의 개수와 마지막 노드의 고유번호가 일치
하기 때문에 마지막 노드의 인덱스 값을 쉽게 알 수 있다! 이것은 중요한 특징이다!



원리 이해 중심의 힙 구현: 초기화와 Helper

```
void HeapInit(Heap * ph)
{
    ph->numOfData = 0;
}
```

초기화!

```
int GetParentIDX(int idx)
{
    return idx/2;
}
```

Helper!

부모 노드의 인덱스 값 반환

```
int HIsEmpty(Heap * ph)
{
    if(ph->numOfData == 0)
        return TRUE;
    else
        return FALSE;
}
```

비었는지 확인

```
int GetLChildIDX(int idx)
{
    return idx*2;
}
```

Helper!

왼쪽 자식 노드의 인덱스 값 반환

```
int GetRChildIDX(int idx)
{
    return GetLChildIDX(idx)+1;
}
```

Helper!

오른쪽 자식 노드의 인덱스 값 반환



원리 이해 중심의 힙 구현: Helper

```
int GetHiPriChildIDX(Heap * ph, int idx)
{
    // 자식 노드가 존재하지 않는다면,
    if(GetLChildIDX(idx) > ph->numOfData)
        return 0; 자식 노드 없으면 0 반환!

    // 자식 노드가 왼쪽 자식 노드 하나만 존재한다면,
    else if(GetLChildIDX(idx) == ph->numOfData)
        return GetLChildIDX(idx);

    // 자식 노드가 둘 다 존재한다면,
    else
    {
        // 오른쪽 자식 노드의 우선순위가 높다면,
        if(ph->heapArr[GetLChildIDX(idx)].pr > ph->heapArr[GetRChildIDX(idx)].pr)
            return GetRChildIDX(idx); // 오른쪽 자식 노드의 인덱스 값 반환

        // 왼쪽 자식 노드의 우선순위가 높다면,
        else
            return GetLChildIDX(idx); // 왼쪽 자식 노드의 인덱스 값 반환
    }
}
```

우선 순위가 높은 자식의 인덱스 값 반환!

*numOfData는 마지막 노드의 고유번호이니,
자식 노드의 값이 이보다 크면 존재하지 않는 자식 노드이다.*

*자식 노드가 하나 존재하면 이는 왼쪽 자식 노드이다.
완전 이진 트리 이므로!*



원리 이해 중심의 힙 구현: HDelete

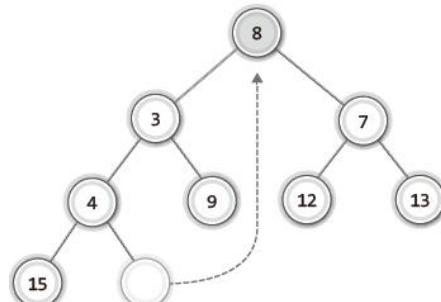
```
HData HDelete(Heap * ph)
{
    HData retData = (ph->heapArr[1]).data;           // 반환을 위해서 삭제할 데이터 저장
    HeapElem lastElem = ph->heapArr[ph->numOfData];   // 힙의 마지막 노드 저장
    마지막 노드를 임시 저장하여 그에 맞는 자리를 찾아나간다!
    // 아래의 변수 parentIdx에는 마지막 노드가 저장될 위치정보가 담긴다.
    int parentIdx = 1;      // 루트 노드가 위치해야 할 인덱스 값 저장
    int childIdx;

    // 루트 노드의 우선순위가 높은 자식 노드를 시작으로 반복문 시작
    while(childIdx = GetHiPriChildIDX(ph, parentIdx))
    {
        if(lastElem.pr <= ph->heapArr[childIdx].pr)    // 마지막 노드와 우선순위 비교
            break;          // 마지막 노드의 우선순위가 높으면 반복문 탈출!

        // 마지막 노드보다 우선순위 높으니, 비교대상 노드의 위치를 한 레벨 올림
        ph->heapArr[parentIdx] = ph->heapArr[childIdx];
        parentIdx = childIdx;    // 마지막 노드가 저장될 위치정보를 한 레벨 내림
    }      // 반복문 탈출하면 parentIdx에는 마지막 노드의 위치정보가 저장됨

    ph->heapArr[parentIdx] = lastElem;      // 마지막 노드 최종 저장
    ph->numOfData -= 1;
    return retData;
}
```

굳이 루트 노드의 자리로 옮기지 않아도 된다!



원리 이해 중심의 힙 구현: HInsert

```
void HInsert(Heap * ph, HData data, Priority pr)
{
    int idx = ph->numOfData+1;          // 새 노드가 저장될 인덱스 값을 idx에 저장
    HeapElem nelem = {pr, data};        // 새 노드의 생성 및 초기화

    // 새 노드가 저장될 위치가 루트 노드의 위치가 아니라면 while문 반복
    while(idx != 1)                    // 새 노드를 마지막 위치에 직접 저장하지 않아도 된다.
    {
        // 새 노드와 부모 노드의 우선순위 비교
        if(pr < (ph->heapArr[GetParentIDX(idx)].pr)) // 새 노드의 우선순위 높다면
        {
            // 부모 노드를 한 레벨 내림, 실제로 내림
            ph->heapArr[idx] = ph->heapArr[GetParentIDX(idx)];

            // 새 노드를 한 레벨 올림, 실제로 올리지는 않고 인덱스 값만 갱신
            idx = GetParentIDX(idx);
        }
        else // 새 노드의 우선순위가 높지 않다면
        {
            break;
        }
    }
    ph->heapArr[idx] = nelem; // 새 노드를 배열에 저장
    ph->numOfData += 1;
}
```

새 노드를 마지막 위치에 직접 저장하지 않아도 된다.
어차피 이동을 하니!

새 노드의 인덱스 정보를 갱신만 하자!
그림처럼 실제 저장까지 할 필요는 없다!
어차피 이동해야 하므로!

```
graph TD; 1((1)) --> 4((4)); 1 --> 7((7)); 4 --> 8((8)); 4 --> 9((9)); 7 --> 12((12)); 7 --> 13((13)); 8 --- 15((15)); 9 --- 3((3))
```



힙의 확인을 위한 main 함수! 그리고 반성!

```
int main(void)
{
    Heap heap;
    HeapInit(&heap);           // 힙의 초기화

    HInsert(&heap, 'A', 1);     // 문자 'A'를 최고의 우선순위로 저장
    HInsert(&heap, 'B', 2);     // 문자 'B'를 두 번째 우선순위로 저장
    HInsert(&heap, 'C', 3);     // 문자 'C'를 세 번째 우선순위로 저장
    printf("%c \n", HDelete(&heap));

    HInsert(&heap, 'A', 1);     // 문자 'A' 한 번 더 저장!
    HInsert(&heap, 'B', 2);     // 문자 'B' 한 번 더 저장!
    HInsert(&heap, 'C', 3);     // 문자 'C' 한 번 더 저장!
    printf("%c \n", HDelete(&heap));

    while(!HIsEmpty(&heap))
        printf("%c \n", HDelete(&heap));

    return 0;
}
```

SimpleHeap.h
SimpleHeap.c
SimpleHeapMain.c

실행결과

```
A
A
B
B
C
C
```

데이터를 저장할 때 우선순위 정보를 별도로 전달하는 것은 적합하지 않은 경우가 많다. 일반적으로 데이터의 우선순위는 데이터를 근거로 판단이 이뤄지기 때문이다.



제법 쓸만한 수준의 힙 구현: 구조체 변경

```
typedef struct _heapElem
{
    Priority pr;
    HData data;
} HeapElem;

typedef struct _heap
{
    int numOfData;
    HeapElem heapArr[HEAP_LEN];
} Heap;
```



구조체의 변경!

```
typedef struct _heap
{
    PriorityComp * comp;
    int numOfData;
    HData heapArr[HEAP_LEN];
} Heap;
```

typedef int PriorityComp(HData d1, HData d2);

```
void HeapInit(Heap * ph, PriorityComp pc)
{
    ph->numOfData = 0;
    ph->comp = pc;
}
```

구조체의 변경에 따른 초기화 함수의 변경!

프로그래머가 힙의 우선순위 판단 기준을 설정할 수 있어야 한다!



제법 쓸만한 수준의 힙 구현: PriorityComp

PriorityComp 형 함수의 정의 기준

- 첫 번째 인자의 우선순위가 높다면, 0보다 큰 값 반환!
- 두 번째 인자의 우선순위가 높다면, 0보다 작은 값 반환!
- 첫 번째, 두 번째 인자의 우선순위가 동일하다면, 0이 반환!

```
void HInsert(Heap * ph, HData data, Priority pr);
```



우선 순위 정보를 별도로 받지 않는다.

```
void HInsert(Heap * ph, HData data);
```

PriorityComp형 함수가 등록되면! HInsert 함수는 등록된 함수를 활용하여 우선순위를 비교 판단한다.



제법 쓸만한 수준의 힙 구현: Helper 함수의 변경

```
int GetHiPriChildIDX(Heap * ph, int idx)
{
    if(GetLChildIDX(idx) > ph->numOfData)
        return 0;

    else if(GetLChildIDX(idx) == ph->numOfData)
        return GetLChildIDX(idx);

    else
    {
        // if(ph->heapArr[GetLChildIDX(idx)].pr
        //     > ph->heapArr[GetRChildIDX(idx)].pr)
        if(ph->comp(ph->heapArr[GetLChildIDX(idx)],
                     ph->heapArr[GetRChildIDX(idx)]) < 0)
            return GetRChildIDX(idx);
        else
            return GetLChildIDX(idx);
    }
}
```

comp에 등록된 함수의 호출 결과를
통해서 우선순위를 판단한다.



제법 쓸만한 수준의 힙 구현: HInsert의 변경

```
void HInsert(Heap * ph, HData data)
{
    int idx = ph->numOfData+1;

    while(idx != 1)
    {
        // if(pr < (ph->heapArr[GetParentIDX(idx)].pr))
        if(ph->comp(data, ph->heapArr[GetParentIDX(idx)]) > 0)
        {
            ph->heapArr[idx] = ph->heapArr[GetParentIDX(idx)];
            idx = GetParentIDX(idx);
        }
        else
        {
            break;
        }
    }

    ph->heapArr[idx] = data;
    ph->numOfData += 1;
}
```

comp에 등록된 함수의 호출 결과를
통해서 우선순위를 판단!



제법 쓸만한 수준의 힙 구현: HDelete의 변경

```
HData HDelete(Heap * ph)
{
    HData retData = ph->heapArr[1];
    HData lastElem = ph->heapArr[ph->numOfData];

    int parentIdx = 1;
    int childIdx;

    while(childIdx = GetHiPriChildIDX(ph, parentIdx))
    {
        // if(lastElem.pr <= ph->heapArr[childIdx].pr)
        if(ph->comp(lastElem, ph->heapArr[childIdx]) >= 0)
            break;

        ph->heapArr[parentIdx] = ph->heapArr[childIdx];
        parentIdx = childIdx;
    }

    ph->heapArr[parentIdx] = lastElem;
    ph->numOfData -= 1;
    return retData;
}
```

comp에 등록된 함수의 호출 결과를
통해서 우선순위를 판단!



제법 쓸만한 수준의 힙 구현: main 함수

```
int main(void)
{
    Heap heap;
    HeapInit(&heap, DataPriorityComp);

    HInsert(&heap, 'A');
    HInsert(&heap, 'B');
    HInsert(&heap, 'C');
    printf("%c \n", HDelete(&heap));

    HInsert(&heap, 'A');
    HInsert(&heap, 'B');
    HInsert(&heap, 'C');
    printf("%c \n", HDelete(&heap));

    while(!HIsEmpty(&heap))
        printf("%c \n", HDelete(&heap));

    return 0;
}
```

```
int DataPriorityComp(char ch1, char ch2)
{
    return ch2-ch1;
//    return ch1-ch2;
}
```

아스키 코드 값이 작은 문자의 우선순위가 더 높다!

UsefulHeap.h

UsefulHeap.c

UsefulHeapMain.c

```
A  
A  
B  
B  
C  
C
```

실행결과



제법 쓸만한 수준의 힙을 이용한 우선순위 큐의 구현

```
#include "UsefulHeap.h"

typedef Heap PQueue;
typedef HData PQData;

void PQueueInit(PQueue * ppq, PriorityComp pc);
int PQIsEmpty(PQueue * ppq);

void PEnqueue(PQueue * ppq, PQData data);
PQData PDequeue(PQueue * ppq);
```

힙의 함수를 사실상 우선순위 큐의 내용으로 구현해 놓았기 때문에 실제 할 일은 별것 없다!

```
void PQueueInit(PQueue * ppq, PriorityComp pc)
{
    HeapInit(ppq, pc);
}

int PQIsEmpty(PQueue * ppq)
{
    return HIsEmpty(ppq);
}

void PEnqueue(PQueue * ppq, PQData data)
{
    HInsert(ppq, data);
}

PQData PDequeue(PQueue * ppq)
{
    return HDelete(ppq);
}
```



수고하셨습니다~



Chapter 09에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 10. 정렬

Introduction To Data Structures Using C

Chapter 10. 정렬

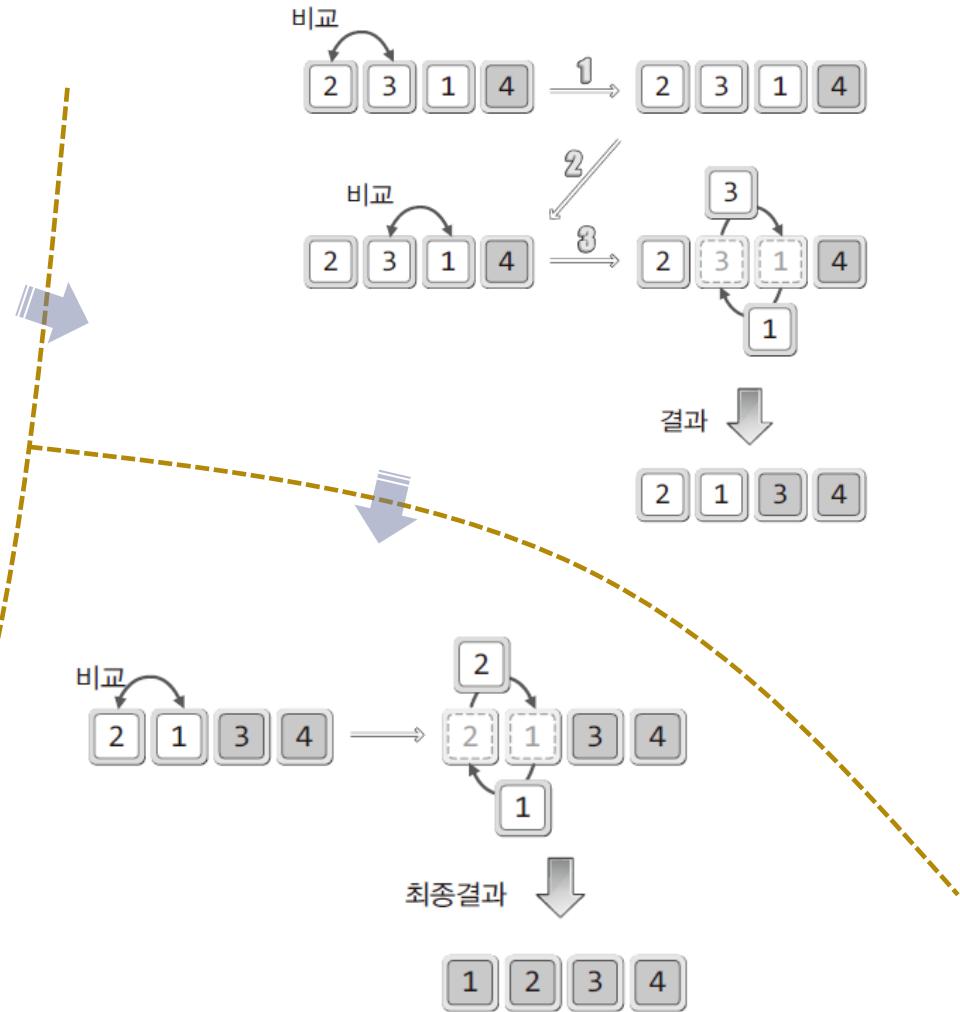
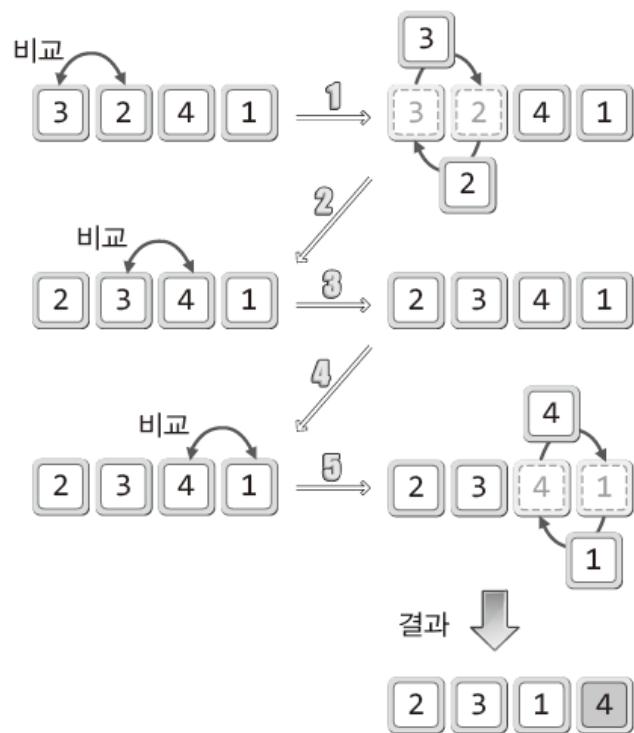


Chapter 10-1:

단순한 정렬 알고리즘



버블 정렬: 이해



버블 정렬: 구현

```
void BubbleSort(int arr[], int n)
{
    int i, j;
    int temp;

    for(i=0; i<n-1; i++)
    {
        for(j=0; j<(n-i)-1; j++)
        {
            if(arr[j] > arr[j+1])
            {
                // 데이터의 교환 ///////
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

버블 정렬의 실질적 구현 코드

```
int main(void)
{
    int arr[4] = {3, 2, 4, 1};
    int i;

    BubbleSort(arr, sizeof(arr)/sizeof(int));

    for(i=0; i<4; i++)
        printf("%d ", arr[i]);

    printf("\n");
    return 0;
}
```

실행결과

1 2 3 4



버블 정렬: 성능평가

성능 평가의 두 가지 기준


```
for(i=0; i<n-1; i++)
{
    for(j=0; j<(n-i)-1; j++)
    {
        if(arr[j] > arr[j+1]) { . . . }
    }           비교 연산
}
```

최악의 경우

비교의 횟수와 이동의 횟수는 일치한다.

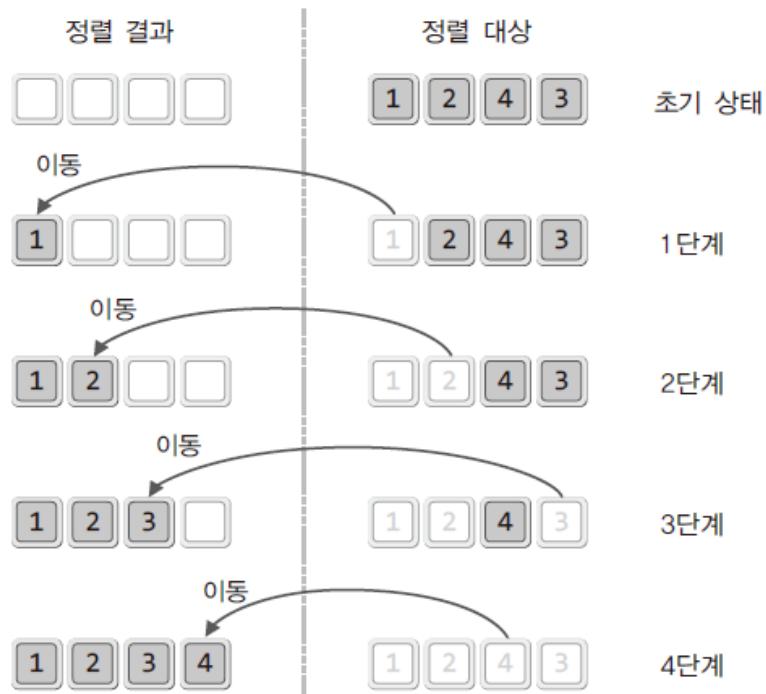
$$(n-1) + (n-2) + \dots + 2 + 1$$

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

$$O(n^2)$$



선택 정렬: 이해

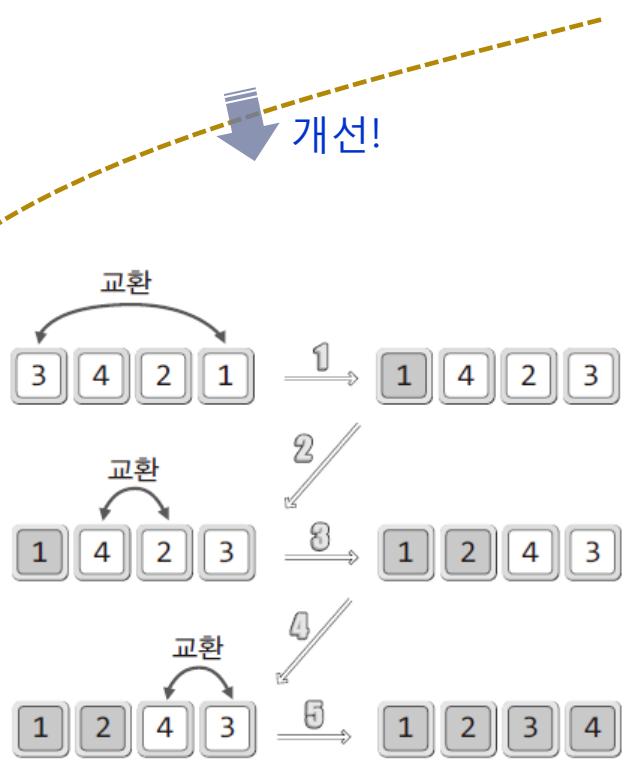


하나씩 비워 가면서 이동을 시킨다.

초기 상태

하나씩 선택해서 정렬 결과를 완성해 나간다!

별도의 메모리 공간이 요구된다는 단점이 있다!



선택 정렬: 구현

```
void SelSort(int arr[], int n)
{
    int i, j;
    int maxIdx;
    int temp;

    for(i=0; i<n-1; i++)
    {
        maxIdx = i;

        for(j=i+1; j<n; j++) // 최솟값 탐색
        {
            if(arr[j] < arr[maxIdx])
                maxIdx = j;
        }

        // 교환 ///////
        temp = arr[i];
        arr[i] = arr[maxIdx];
        arr[maxIdx] = temp;
    }
}
```

```
int main(void)
{
    int arr[4] = {3, 4, 2, 1};
    int i;

    SelSort(arr, sizeof(arr)/sizeof(int));

    for(i=0; i<4; i++)
        printf("%d ", arr[i]);

    printf("\n");
    return 0;
}
```

실행결과

1 2 3 4



선택 정렬: 성능평가

```
for(i=0; i<n-1; i++)  
{  
    maxIdx = i;  
  
    for(j=i+1; j<n; j++)  
    {  
        if(arr[j] < arr[maxIdx])  비교 연산  
            maxIdx = j;  
    }  
  
    // 교환 //////// 이동 연산  
    temp = arr[i];  
    arr[i] = arr[maxIdx];  
    arr[maxIdx] = temp;  
}
```

비교의 횟수

$$(n-1) + (n-2) + \dots + 2 + 1$$



$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

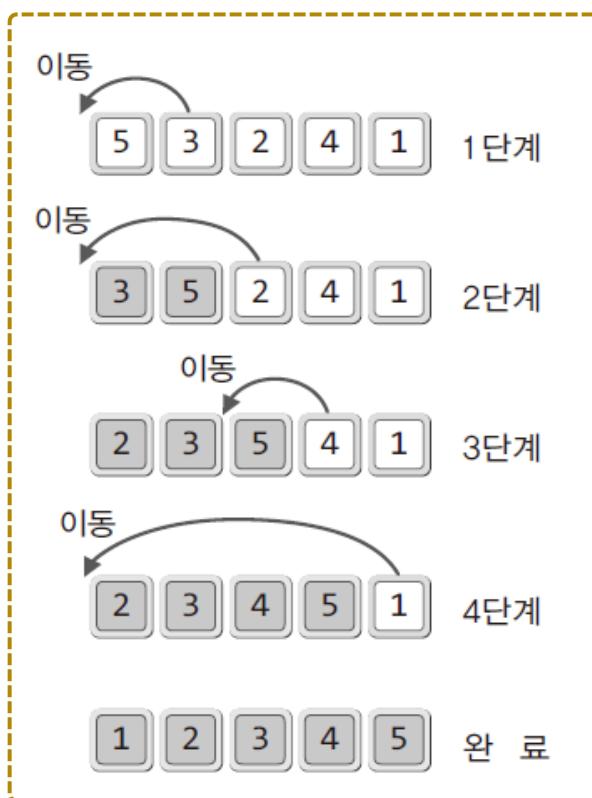
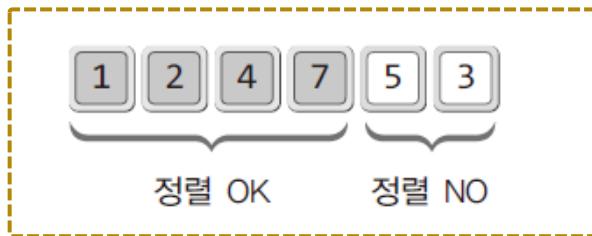


$$O(n^2)$$

최악의 경우와 최상의 경우 구분 없이 데이터 이동의 횟수는 동일하다!



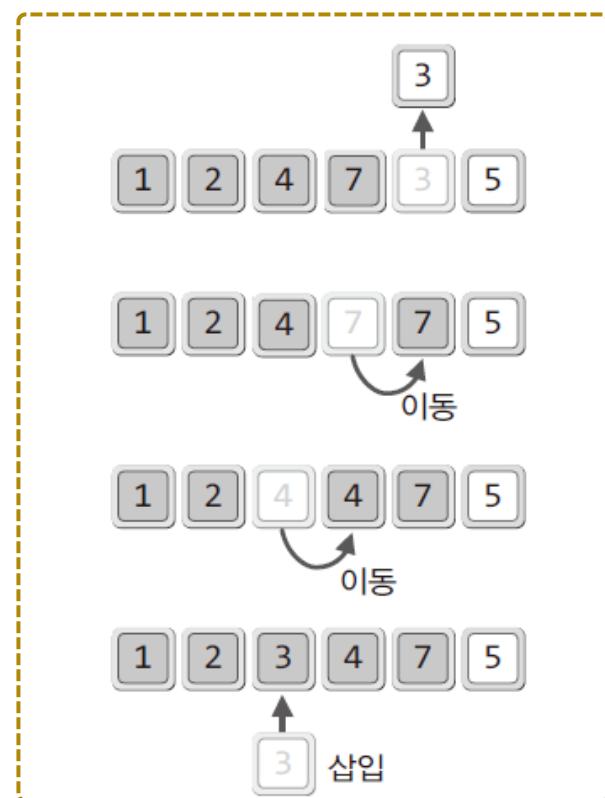
삽입 정렬: 이해



정렬이 완료된 영역과 그렇지 않은 영역을 구분하는 방법

뒤로 밀어 내는 방법을 포함한 그림

구현을 고려하면!



삽입 정렬: 구현

```
void InserSort(int arr[], int n)
{
    int i, j;
    int insData;

    for(i=1; i<n; i++)
    {
        insData = arr[i];           정렬 대상 저장

        for(j=i-1; j>=0 ; j--)
        {
            if(arr[j] > insData)
                arr[j+1] = arr[j];
            else      비교 대상 뒤로 한 칸 밀기
                break;
        }

        arr[j+1] = insData;         찾은 위치에 정렬대상 삽입
    }
}
```

```
int main(void)
{
    int arr[5] = {5, 3, 2, 4, 1};
    int i;

    InserSort(arr, sizeof(arr)/sizeof(int));

    for(i=0; i<5; i++)
        printf("%d ", arr[i]);

    printf("\n");
    return 0;
}
```

실행결과

1	2	3	4	5
---	---	---	---	---



삽입 정렬: 성능평가

```
for(i=1; i<n; i++)  
{  
    ....  
    for(j=i-1; j>=0 ; j--)  
    {  
        if(arr[j] > insData)  
            arr[j+1] = arr[j];  
        else  
            break;  
    }  
    ....  
}
```

데이터의 비교 및 이동 연산이 안쪽 for문 안에 존재한다.

따라서 최악의 경우 빅-오는 버블 및 삽입 정렬과 마찬가지로 $O(n^2)$

데이터간 비교연산

데이터간 이동연산

최악의 경우 이 break문은 한번도 실행이 되지 않는다.

Chapter 10. 정렬



Chapter 10-2:

복잡하지만 효율적인 정렬 알고리즘



힙 정렬: 이해와 구현

힙의 특성을 활용하여, 힙에 정렬할 대상을 모두 넣었다가 다시 꺼내어 정렬을 진행한다!

```
int PriComp(int n1, int n2)
{
    return n2-n1;          // 오름차순 정렬을 위한 문장
//    return n1-n2;
}

힙에서 요구하는 정렬 기준 함수와 동일한 기준을 적용한다!
void HeapSort(int arr[], int n, PriorityComp pc)
{
    Heap heap;
    int i;

    HeapInit(&heap, pc);

    // 정렬대상을 가지고 힙을 구성한다.
    for(i=0; i<n; i++)
        HInsert(&heap, arr[i]);

    // 순서대로 하나씩 꺼내서 정렬을 완성한다.
    for(i=0; i<n; i++)
        arr[i] = HDelete(&heap);
}
```

```
int main(void)
{
    int arr[4] = {3, 4, 2, 1};
    int i;

    HeapSort(arr, sizeof(arr)/sizeof(int), PriComp);

    for(i=0; i<4; i++)
        printf("%d ", arr[i]);

    printf("\n");
    return 0;
}
```

UsefulHeap.h

UsefulHeap.c

HeapSort.c

1 2 3 4

실행결과



힙 정렬: 성능평가

하나의 데이터를 힙에 넣고 빼는 경우에 대한 시간 복잡도

- 힙의 데이터 저장 시간 복잡도 $O(\log_2 n)$
- 힙의 데이터 삭제 시간 복잡도 $O(\log_2 n)$

↓ 하나로 묶으면

$O(2\log_2 n)$, 그런데

앞의 2는 빅-오에서 무시 할 수 있으므로! $O(\log_2 n)$

N개의 데이터



$O(n \log_2 n)$

두 빅-오의 비교를 위한 테이블

n	10	100	1,000	3,000	5,000
n^2	100	10,000	1,000,000	9,000,000	25,000,000
$n \log_2 n$	66	664	19,931	34,652	61,438



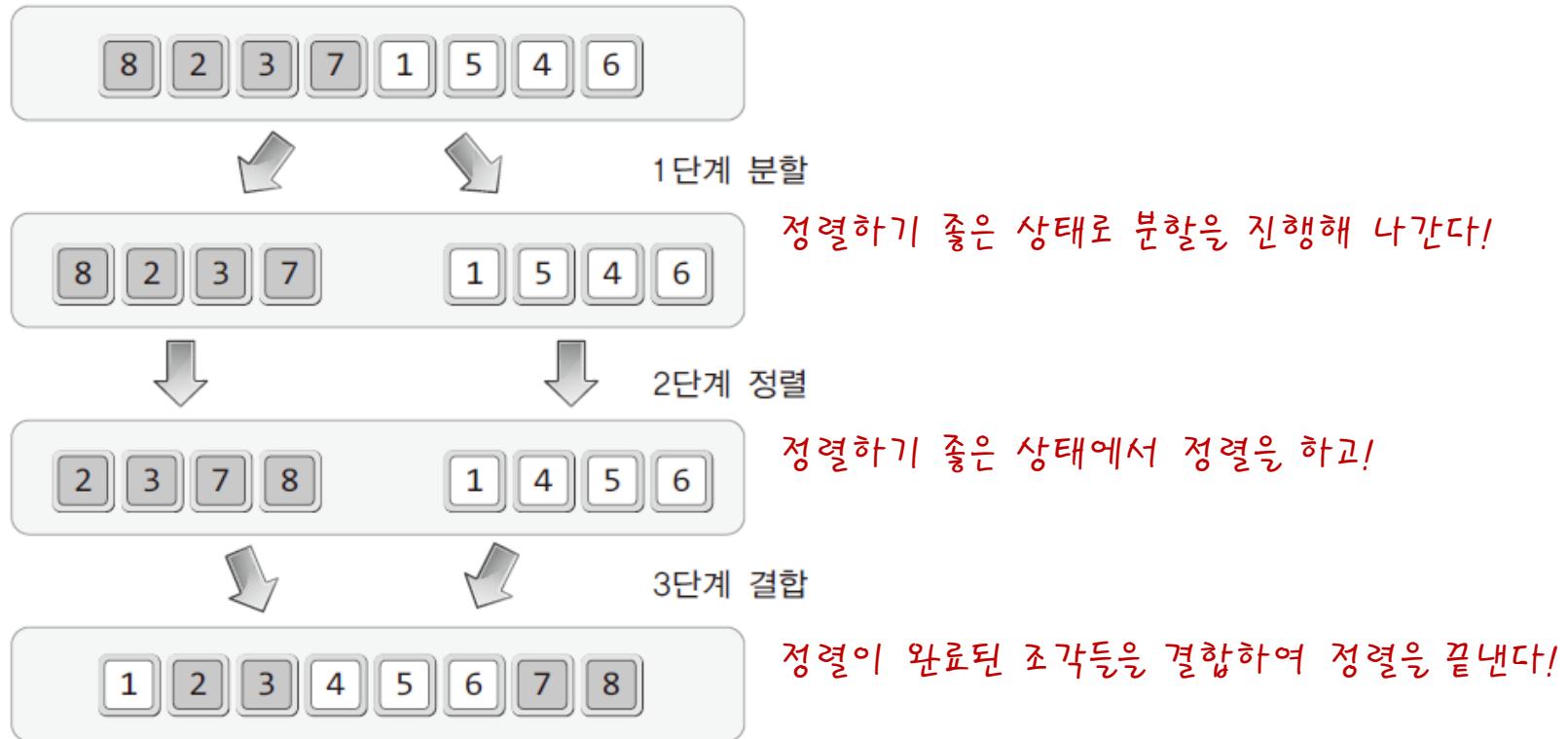
병합 정렬: Divide And Conquer

- 1단계 분할(Divide) 해결이 용이한 단계까지 문제를 분할해 나간다.
- 2단계 정복(Conquer) 해결이 용이한 수준까지 분할된 문제를 해결한다.
- 3단계 결합(Combine) 분할해서 해결한 결과를 결합하여 마무리한다.

병합 정렬 알고리즘 역시 *DAC*를 기반으로 설계된 알고리즘이다.



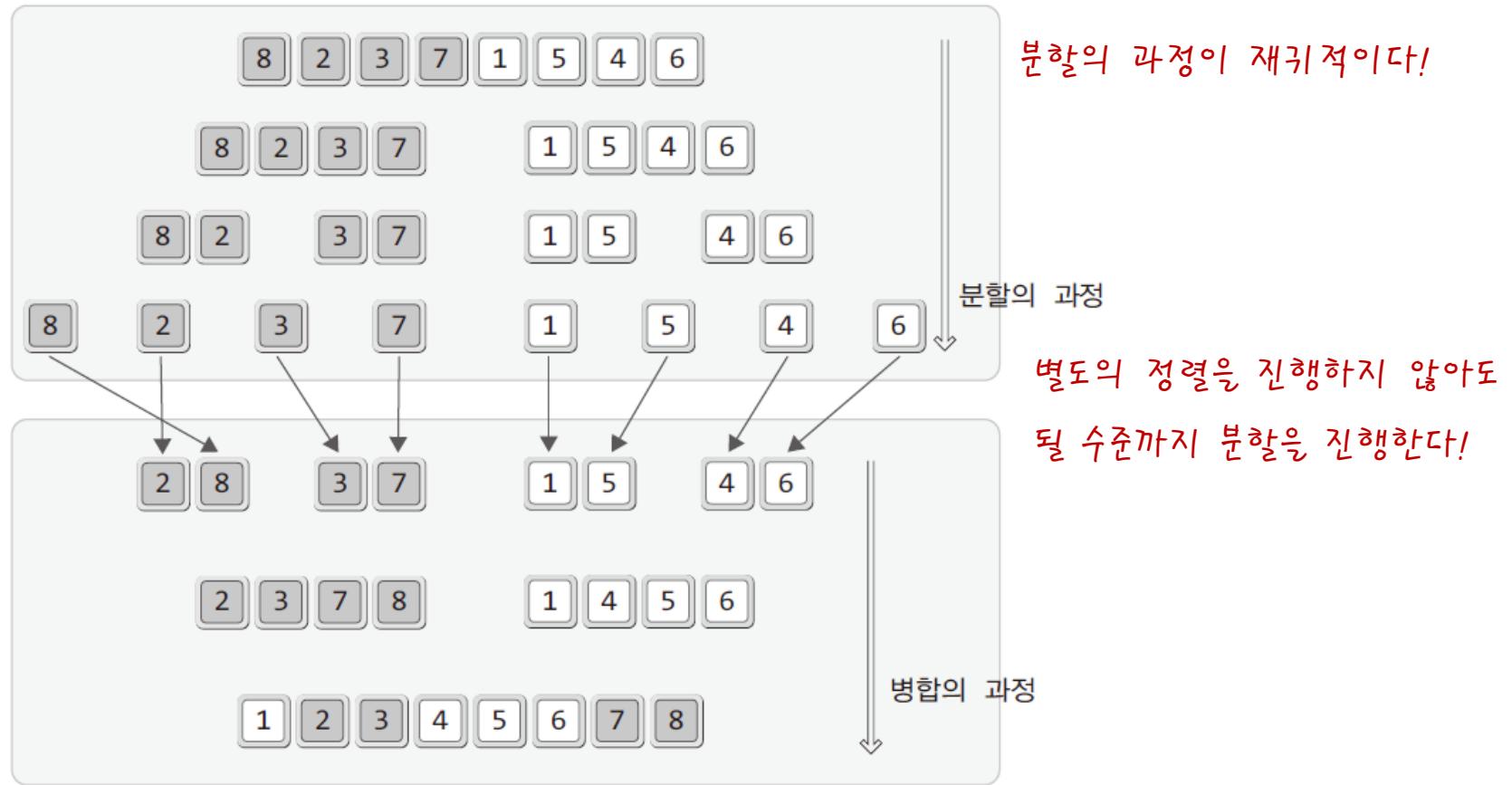
병합 정렬: DAC 관점에서의 이해



▶ [그림 10-9: 병합 정렬의 기본 원리]



병합 정렬: 분할의 방법은?



▶ [그림 10-10: 병합 정렬의 예]

분할보다 신경 써야 하는 것이 병합과정이다.

그래서 병합정렬이라 한다.



병합 정렬: 재귀적 구현

```
void MergeSort(int arr[], int left, int right)
{
    int mid;

    if(left < right)      // left가 작다는 것은 더 나눌 수 있다는 뜻!
    {
        // 중간지점을 계산한다.
        mid = (left+right) / 2;
        MergeSort 함수는 둘로 나눌 수 없을 때까지 재귀적으로 호출된다.

        // 둘로 나눠서 각각을 정렬한다.
        MergeSort(arr, left, mid);      // left~mid에 위치한 데이터 정렬!
        MergeSort(arr, mid+1, right);   // mid+1~right에 위치한 데이터 정렬!

        // 정렬된 두 배열을 병합한다.
        MergeTwoArea(arr, left, mid, right);
    }
}
```

병합 정렬: 병합을 위한 함수의 정의

```
void MergeTwoArea(int arr[], int left, int mid, int right)
{
    int fIdx = left;    int rIdx = mid+1;    int i;
    int * sortArr = (int*)malloc(sizeof(int)*(right+1)); 병합 결과를 담을 메모리 공간 할당
    int sIdx = left;

    while(fIdx <= mid && rIdx <= right) {
        if(arr[fIdx] <= arr[rIdx])
            sortArr[sIdx] = arr[fIdx++];
        else
            sortArr[sIdx] = arr[rIdx++];
        sIdx++;
    }

    if(fIdx > mid) {
        for(i=rIdx; i <= right; i++)
            sortArr[sIdx] = arr[i];
    }
    else {
        for(i=fIdx; i <= mid; i++)
            sortArr[sIdx] = arr[i];
    }

    for(i=left; i <= right; i++) arr[i] = sortArr[i]; 병합 결과를 옮겨 담는다!
    free(sortArr);
}
```

병합 할 두 영역의 데이터를 비교하여 배열 sortArr에 담는다!

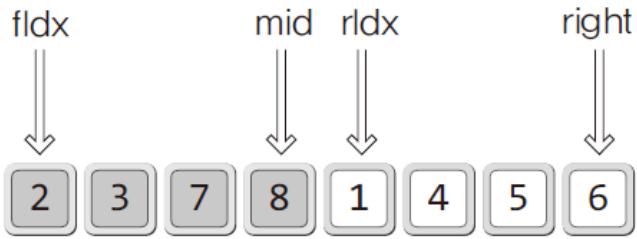
배열의 앞 부분이 sortArr로 모두 이동되어서 배열 뒷부분에 남은 데이터를 모두 sortArr로 이동!

배열의 뒷 부분이 sortArr로 모두 이동되어서 배열 앞부분에 남은 데이터를 모두 sortArr로 이동!

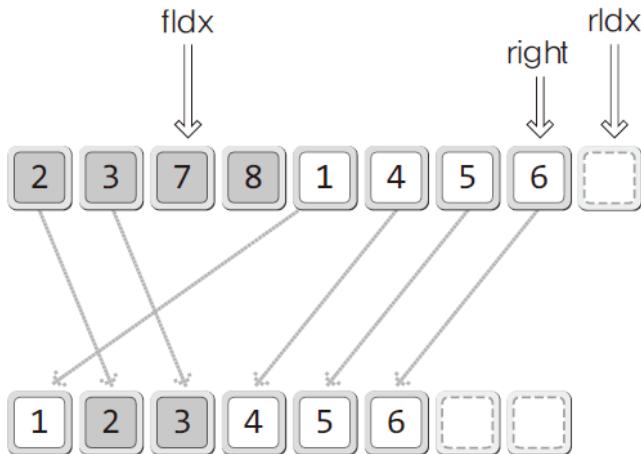
병합 결과를 옮겨 담는다!



병합 정렬: 병합 함수의 코드 설명



↓ 1차 병합의 결과



```
void MergeTwoArea(int arr[], int left, int mid, int right)
{
```

```
    int fIdx = left;    int rIdx = mid+1;    int i;
    int * sortArr = (int*)malloc(sizeof(int)*(right-left+1));
    int sIdx = left;
```

```
    while(fIdx <= mid && rIdx <= right) {
        if(arr[fIdx] <= arr[rIdx])
            sortArr[sIdx] = arr[fIdx++];
        else
            sortArr[sIdx] = arr[rIdx++];
        sIdx++;
    }
```

..... 1차 병합을 진행하는 부분

병합 정렬: 성능평가(내용 비교)

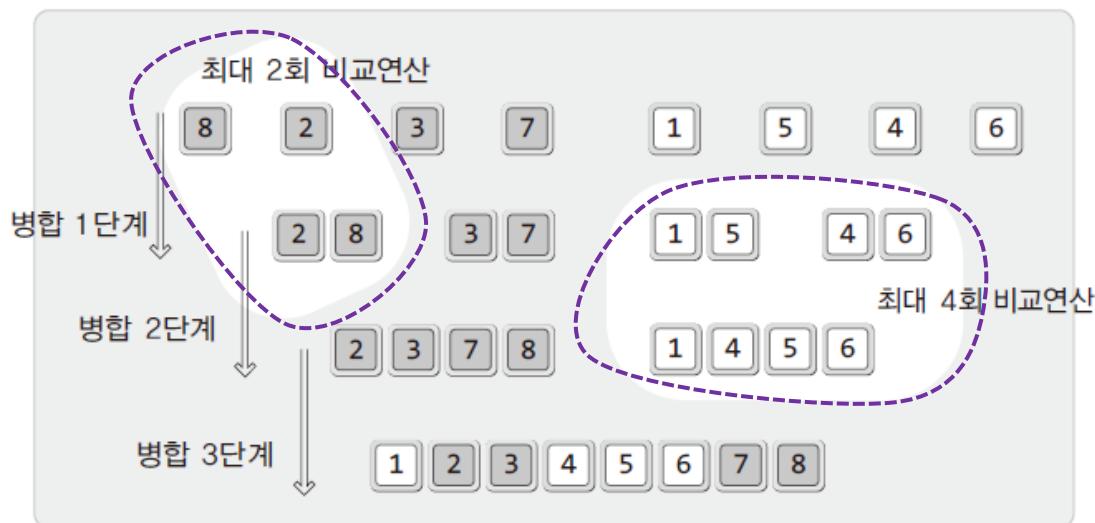
데이터의 비교 및 데이터의 이동은

MergeTwoArea 함수를 중심으로 진행!

따라서 병합 정렬의 성능은 MergeTwoArea
함수를 기준으로 계산!

MergeTwoArea의
핵심 루틴

```
while(fIdx<=mid && rIdx<=right)
{
    if(arr[fIdx] <= arr[rIdx])
        sortArr[sIdx] = arr[fIdx++];
    else
        sortArr[sIdx] = arr[rIdx++];
    sIdx++;
}
```



- 1과 4 비교 후 1 이동
- 5와 4 비교 후 4 이동
- 5와 6 비교 후 5 이동
- 6을 이동하기 위한 비교

병합 정렬: 성능평가(내용 비교)

“정렬의 대상인 데이터의 수가 n 개 일 때, 각 병합의 단계마다 최대 n 번의 비교연산이 진행된다.”



따라서 데이터 수에 대한 비교 연산의 횟수는

$$n \log_2 n$$



따라서 비교 연산에 대한 빅-오는

$$O(n \log_2 n)$$



병합 정렬: 성능평가(이동)

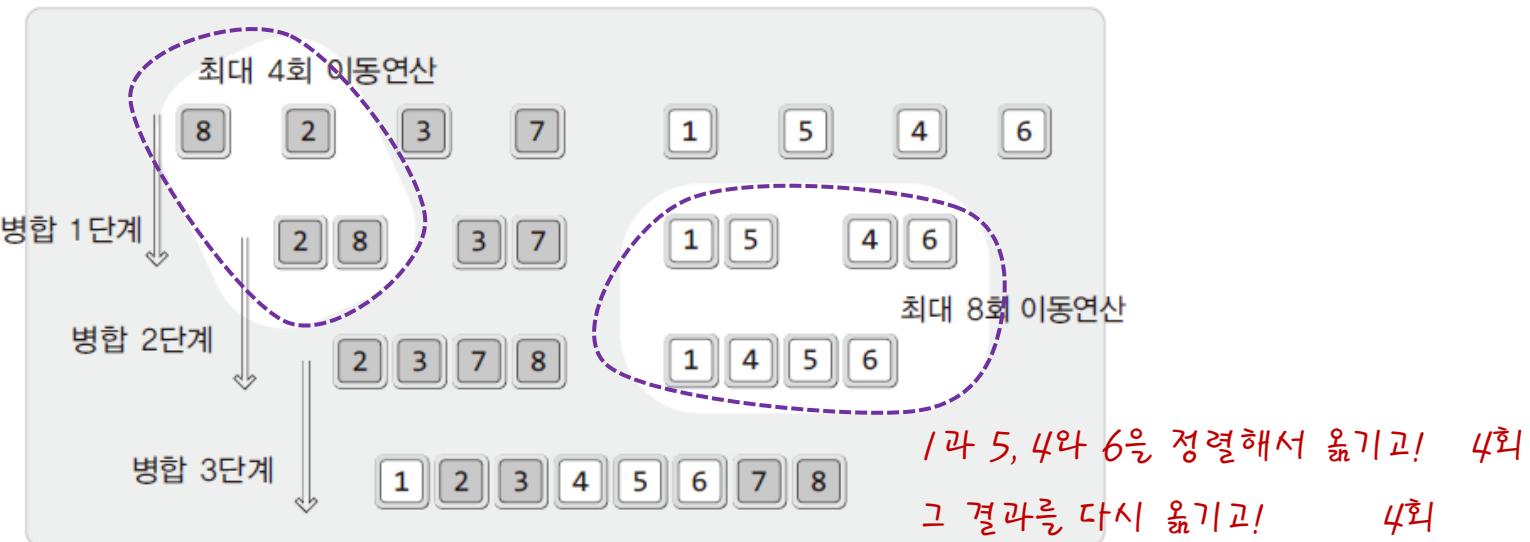
- 임시 배열에 데이터를 병합하는 과정에서 한 번!
- 임시 배열에 저장된 데이터 전부를 원위치로 옮기는 과정에서 한 번!

$$2n \log_2 n \Rightarrow O(n \log_2 n)$$

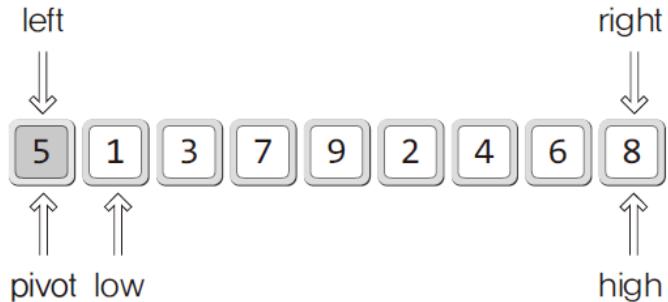
최악, 최선 상관 없이!

8과 2를 정렬해서 옮기고! 2회

그 결과를 다시 옮기고! 2회



퀵 정렬: 이해(1단계: 초기화)



퀵 정렬을 위해서는 총 5개의 변수

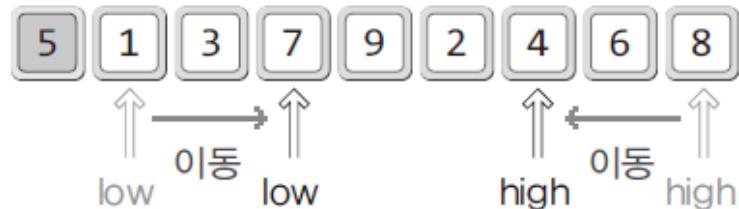
*left, right, pivot, low, high*를 선언해야 한다.

- **left** 정렬대상의 가장 왼쪽 지점을 가리키는 이름
- **right** 정렬대상의 가장 오른쪽 지점을 가리키는 이름
- **pivot** 피벗이라 발음하고 중심점, 중심축의 의미를 담고 있다.
- **low** 피벗을 제외한 가장 왼쪽에 위치한 지점을 가리키는 이름
- **high** 피벗을 제외한 가장 오른쪽에 위치한 지점을 가리키는 이름

가장 왼쪽에 위치한 데이터를 피벗으로 결정하기로 하자! 물론 피벗은 달리 결정할 수 있다!



퀵 정렬: 이해(2단계: low와 high의 이동)



Low와 high의 이동은 완전 별개이다!

low와 high의 첫번째 정거장

- low의 오른쪽 방향 이동 피벗보다 큰 값을 만날 때까지
- high의 왼쪽 방향 이동 피벗보다 작은 값을 만날 때까지

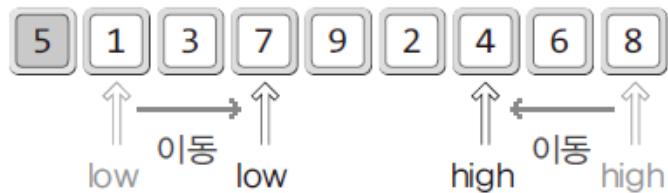


일반화 해서 표현하면

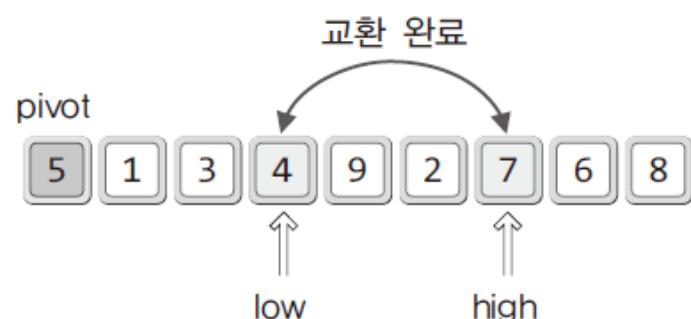
- low의 오른쪽 방향 이동 피벗보다 정렬의 우선순위가 낮은 데이터를 만날 때까지
- high의 왼쪽 방향 이동 피벗보다 정렬의 우선순위가 높은 데이터를 만날 때까지



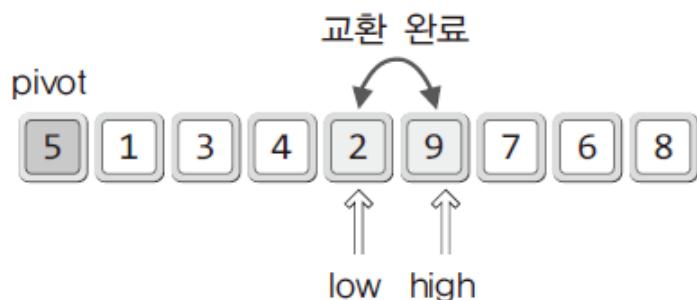
퀵 정렬: 이해(3단계: low와 high의 교환)



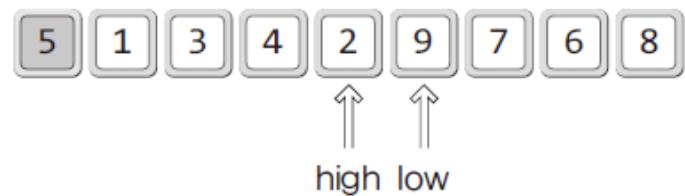
low와 high의 데이터 교환



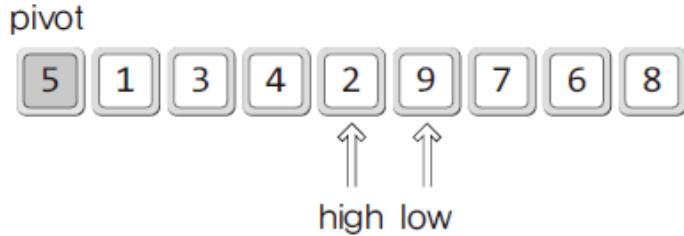
교환 후 이동을 계속,
그리고 또 교환!



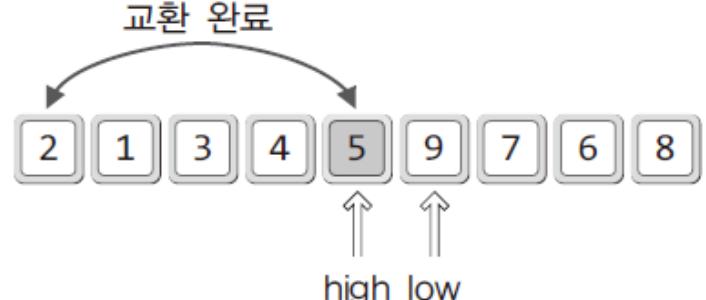
이동을 계속,
high와 low가 역전 될때까지
pivot



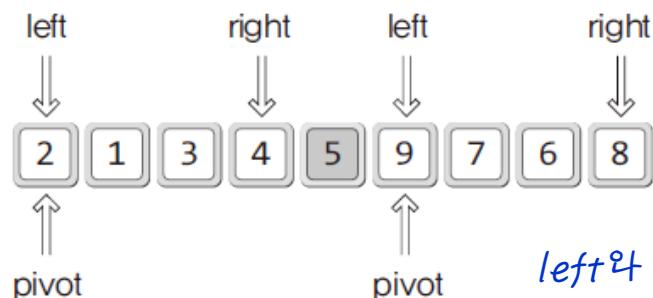
퀵 정렬: 이해(4단계: 피벗의 이동)



high와 low가 역전되면,
피벗과 high의 데이터 교환



두 개의 영역으로 나누어 반복 실행



여기까지가 회전 완료!

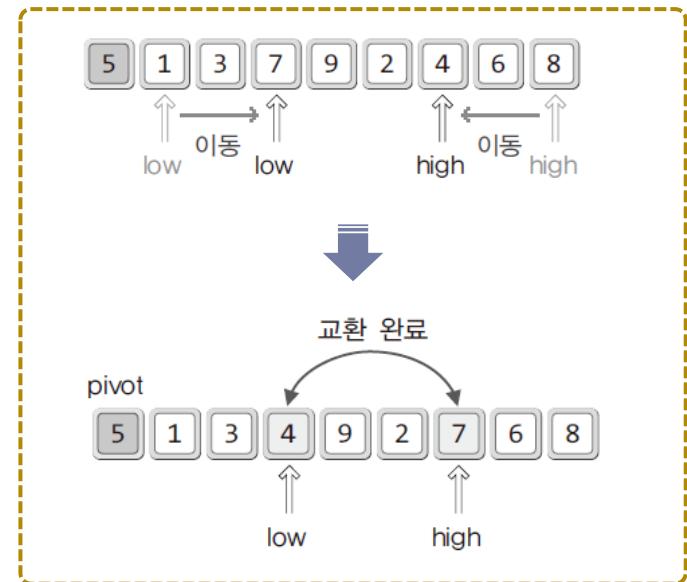
left와 right가 다음 관계에 놓일 때까지 반복!

$left > right$

퀵 정렬: 구현(핵심 알고리즘)

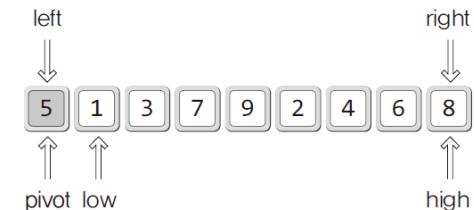
```
int Partition(int arr[], int left, int right)
{
    int pivot = arr[left];      // 피벗의 위치는 가장 왼쪽!
    int low = left+1;
    int high = right;
    while(low <= high)        // 교차되지 않을 때까지 반복
    {
        // 피벗보다 큰 값을 찾는 과정
        while(pivot > arr[low])
            low++;           // low를 오른쪽으로 이동
        // 피벗보다 작은 값을 찾는 과정
        while(pivot < arr[high])
            high--;          // high를 왼쪽으로 이동
        // 교차되지 않은 상태라면 Swap 실행
        if(low <= high)
            Swap(arr, low, high);
    }
    Swap(arr, left, high);    // 피벗과 high가 가리키는 대상 교환
    return high;              // 옮겨진 피벗의 위치정보 반환
}
```

while문 내에서 진행되는 일의 내용



퀵 정렬: 구현(재귀적 완성과 수정사항)

```
void QuickSort(int arr[], int left, int right)
{
    if(left <= right)
    {
        int pivot = Partition(arr, left, right);          // 둘로 나눠서
        QuickSort(arr, left, pivot-1);                   // 왼쪽 영역을 정렬
        QuickSort(arr, pivot+1, right);                  // 오른쪽 영역을 정렬
    }
}
```



```
int Partition(int arr[], int left, int right)
{
```

```
    ....  
    while(low <= high)
```

```
    {  
        while(pivot > arr[low])  
            low++;
    }
```

```
    while(pivot < arr[high])  
        high--;
    ....
```

```
    }  
    ....
```

피벗
↓
3, 3, 3
low
high

// 항상 '참'일 수밖에 없는 상황!

// 문제가 되는 지점!

// 문제가 되는 지점!

while(pivot >= arr[low] && low <= right)

정렬 범위 넘지 않기 위해!

while(pivot <= arr[high] && high >= (left+1))

정렬 범위 넘지 않기 위해!

퀵 정렬: 구현(피벗 선택에 대한 논의)

pivot

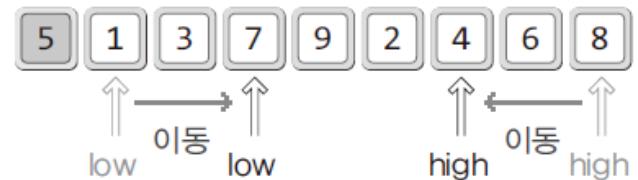


이렇듯 정렬이 되어 있고 피벗이 정렬 대상의 한 쪽 끝에 치우치는 경우 최악의 경우를 보인다.

pivot



이렇듯 데이터가 불규칙적으로 나열되어 있고 피벗이 중간에 해당하는 값에 가깝게 선택이 될 수록 최상의 경우를 보인다.

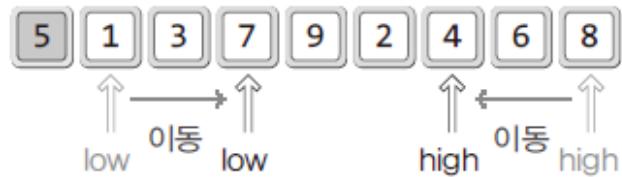


정렬과정에서 선택되는 피벗의 수가 적을 수록 최상의 경우에 해당이 된다!

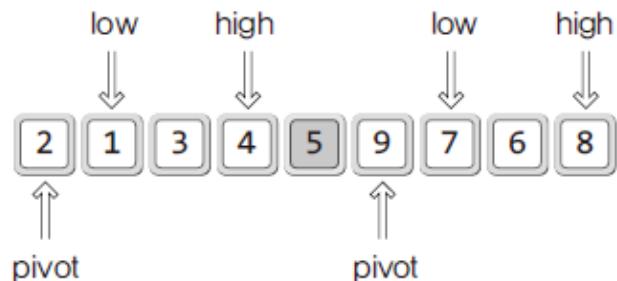
결론은 피벗이 가급적 중간에 해당하는 값이 선택되어야 좋은 성능을 보인다는 것!



퀵 정렬: 성능평가(최선의 경우)



▶ [그림 10-24: 비교연산 횟수의 힌트1]



▶ [그림 10-25: 비교연산 횟수의 힌트2]

- 31개의 데이터는 15개씩 둘로 나뉘어 총 2 조각이 된다.
- 이어서 각각 7개씩 둘로 나뉘어 총 4 조각이 된다.
- 이어서 각각 3개씩 둘로 나뉘어 총 8 조각이 된다.
- 이어서 각각 1개씩 둘로 나뉘어 총 16 조각이 된다.

$$k = \log_2 n$$

1차 나뉨
2차 나뉨
3차 나뉨
4차 나뉨

$O(n \log_2 n)$

최선의 경우 빅-오



퀵 정렬: 성능평가(최악의 경우)



중간에 가까운 값으로 빅-오를 선택하려는 노력을 조금만 하더라도 퀵 정렬은 최악의 경우를 만들지 않는다. 따라서 퀵 정렬의 성능은 최상의 경우를 근거로 이야기 한다.

퀵 정렬은 $O(n \log_2 n)$ 의 성능을 보이는 정렬 알고리즘 중에서 평균적으로 가장 좋은 성능을 보이는 알고리즘이다. 다른 알고리즘에 비해서 데이터의 이동이 적고 별도의 메모리 공간을 요구하지도 않는데 그 이유가 있다.



기수 정렬: 특징과 적용 범위

기수 정렬의 특징

- 정렬순서의 앞서고 뒤섬을 비교하지 않는다.
- 정렬 알고리즘의 한계로 알려진 $O(n \log_2 n)$ 을 뛰어 넘을 수 있다.
- 적용할 수 있는 대상이 매우 제한적이다. 길이가 동일한 데이터들의 정렬에 용이하다!

"배열에 저장된 1, 7, 9, 5, 2, 6을 오름차순으로 정렬하라!"

기수 정렬 OK!

"영단어 red, why, zoo, box를 사전편찬 순서대로 정렬하여라!"

기수 정렬 OK!

"배열에 저장된 21, -9, 125, 8, -136, 45를 오름차순으로 정렬하라!"

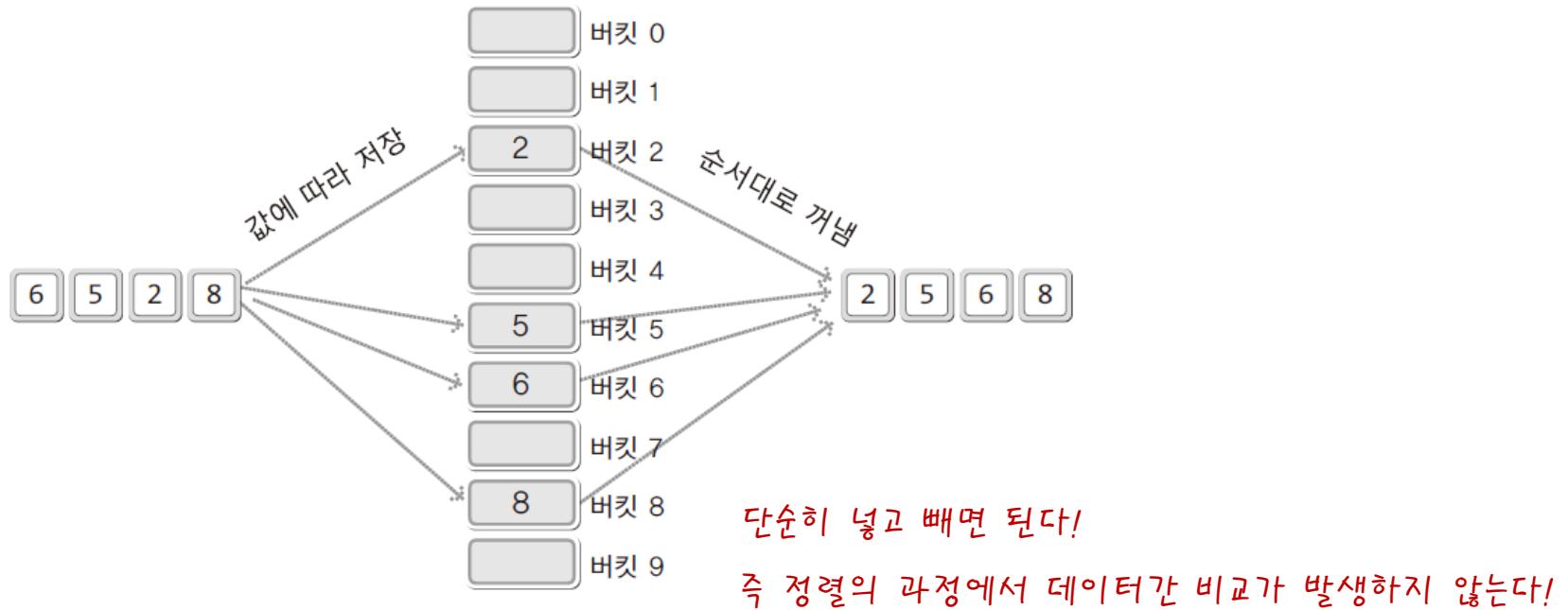
기수 정렬 NO!

"영단어 professionalism, few, hydroxyproline, simple을 사전편찬 순서대로 정렬하여라!"

기수 정렬 NO!



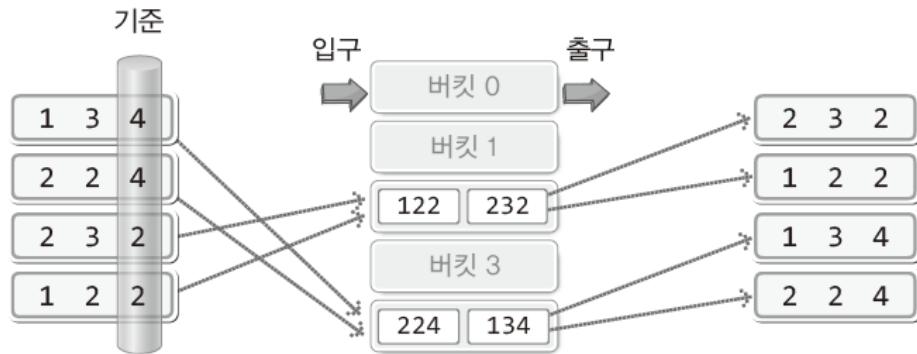
기수 정렬: 정렬의 원리



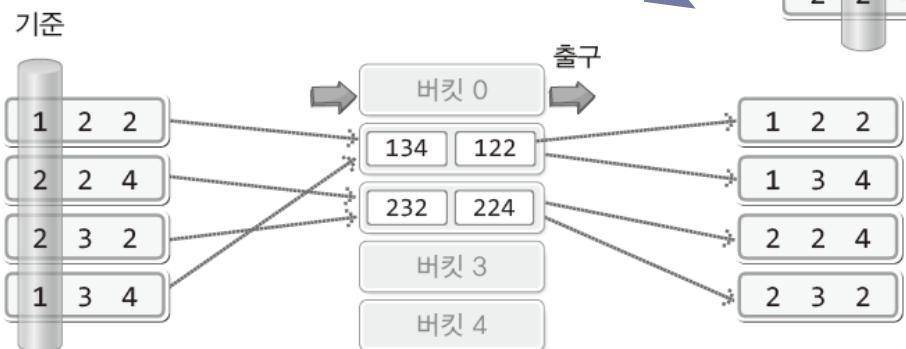
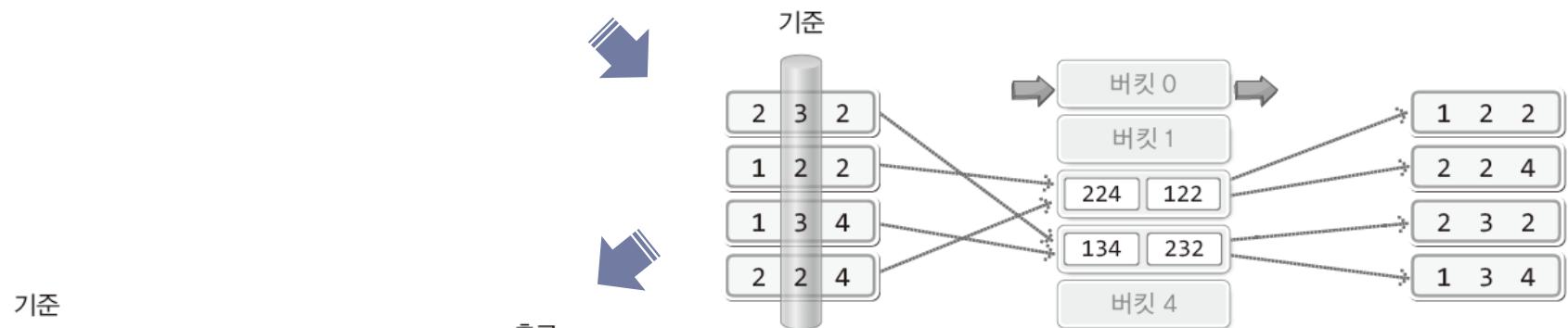
- 기수(radix): 주어진 데이터를 구성하는 기본 요소(기호)
- 버킷(bucket): 기수의 수에 해당하는 만큼의 버킷을 활용한다.



기수 정렬: LSD



List Significant Digit을 시작으로 정렬을
진행하는 방식!

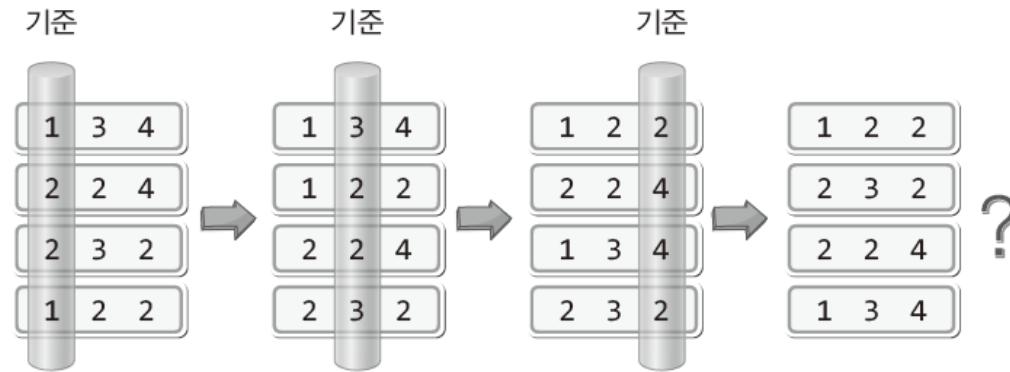


이렇듯 마지막까지 진행을 해야
값의 우선순위대로 정렬이 완성된다.



기수 정렬: MSD(Most Significant Digit)

MSD는 정렬의 기준 선정 방향이 LSD와 반대이다! 그렇다면 방향성에서만 차이를 보일까?



정렬의 기준 선정 방향만을 바꾸어서 기수 정렬을 진행한 결과!

MSD 방식은 점진적으로 정렬이 완성되어 가는 방식이다!

따라서 중간 중간에 정렬이 완료된 데이터는 더 이상의 정결과정을 진행하지 않아야 한다.



기수 정렬: LSD 기준 구현

MSD와 LSD의 빅-오는 같다. 하지만 LSD의 구현이 더 용이하다! 따라서 LSD를 기준으로 기수 정렬을 구현하는 것이 일반적이다.

- | | |
|-------------------------|----------------|
| · NUM으로부터 첫 번째 자리 숫자 추출 | NUM / 1 % 10 |
| · NUM으로부터 두 번째 자리 숫자 추출 | NUM / 10 % 10 |
| · NUM으로부터 세 번째 자리 숫자 추출 | NUM / 100 % 10 |

양의 정수라면 길이에 상관 없이 정렬 대상에 포함시키기 위한 간단한 알고리즘

LSD 방식에서는 모든 데이터가 정렬의 과정을 동일하게 거치도록 구현한다. 반면 MSD 방식에서는 선별해서 거치도록 구현해야 한다. 따라서 LSD 방식의 구현이 더 용이할 뿐만 아니라 생각과 달리 성능도 MSD에 비교해서 떨어지지 않는다.



기수 정렬: code 구현

```
#include "ListBaseQueue.h"

#define BUCKET_NUM      10

void RadixSort(int arr[], int num, int maxlen)
{
    Queue buckets[BUCKET_NUM];
    int bi;
    int pos;
    int di;
    int divfac = 1;
    int radix;

    // 총 10개의 버킷 초기화
    for(bi=0; bi<BUCKET_NUM; bi++)
        QueueInit(&buckets[bi]);

    // 가장 긴 데이터의 길이만큼 반복
    for(pos=0; pos<maxlen; pos++)
    {
        . . .
    }
}
```

```
// 정렬대상의 수만큼 반복
for(di=0; di<num; di++)
{
    // N번째 자리의 숫자 추출
    radix = (arr[di] / divfac) % 10;

    // 추출한 숫자를 근거로 버킷에 데이터 저장
    Enqueue(&buckets[radix], arr[di]);
}

// 버킷 수만큼 반복
for(bi=0, di=0; bi<BUCKET_NUM; bi++)
{
    // 버킷에 저장된 것 순서대로 다 꺼내서 다시 arr에 저장
    while(!QIsEmpty(&buckets[bi]))
        arr[di++] = Dequeue(&buckets[bi]);
}

// N번째 자리의 숫자 추출을 위한 피제수의 증가
divfac *= 10;
```

RadixSort 함수의 호출 예

```
int arr[7] = {13, 212, 14, 7141, 10987, 6, 15};
int len = sizeof(arr) / sizeof(int);
RadixSort(arr, len, 5);
```



기수 정렬: 성능평가

```
void RadixSort(int arr[], int num, int maxLen)
{
    . . .
    // 가장 긴 데이터의 길이만큼 반복
    for(pos=0; pos<maxLen; pos++)
    {
        // 정렬대상의 수만큼 버킷에 데이터 삽입
        for(di=0; di<num; di++)
        {
            // 버킷으로의 데이터 삽입 진행
        }

        // 정렬대상의 수만큼 버킷으로부터 데이터 추출
        for(bi=0, di=0; bi<BUCKET_NUM; bi++)
        {
            // 버킷으로부터의 데이터 추출 진행
        }
    }
}
```

버킷으로의 데이터 삽입과 추출을 근거로 빅-오를 결정!

삽입

삽입과 추출 연산을 한 쌍으로 묶으면,
이 한 쌍의 연산 수행 횟수는 다음과 같다.

$\text{maxLen} \times \text{num}$

추출

따라서 정렬대상의 수가 n 이고, 모든 정렬대상의 길이를 $/$ 이라 할 때 시
간 복잡도에 대한 기수 정렬의 빅-오는 다음과 같다.

$O(\ln)$



수고하셨습니다~



Chapter 10에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 11. 탐색1

Introduction To Data Structures Using C

Chapter 11. 탐색1



Chapter 11-1:

탐색의 이해와 보간 탐색



탐색의 이해

효율적인 탐색을 위해서는 '어떻게 찾을까'만을 고민해서는 안 된다.

그보다는 '효율적인 탐색을 위한 저장방법'을 우선 고민해야 한다.

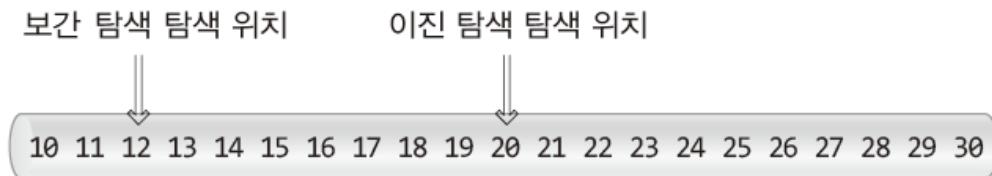
효율적인 탐색이 가능한 대표적인 자료구조는 트리이다.



보간 탐색

이진 탐색과 보간 탐색 모두 정렬이 완료된 데이터를 대상으로 탐색을 진행하는 알고리즘이다.

아래의 배열을 대상으로 정수 12를 찾는다고 가정하면?

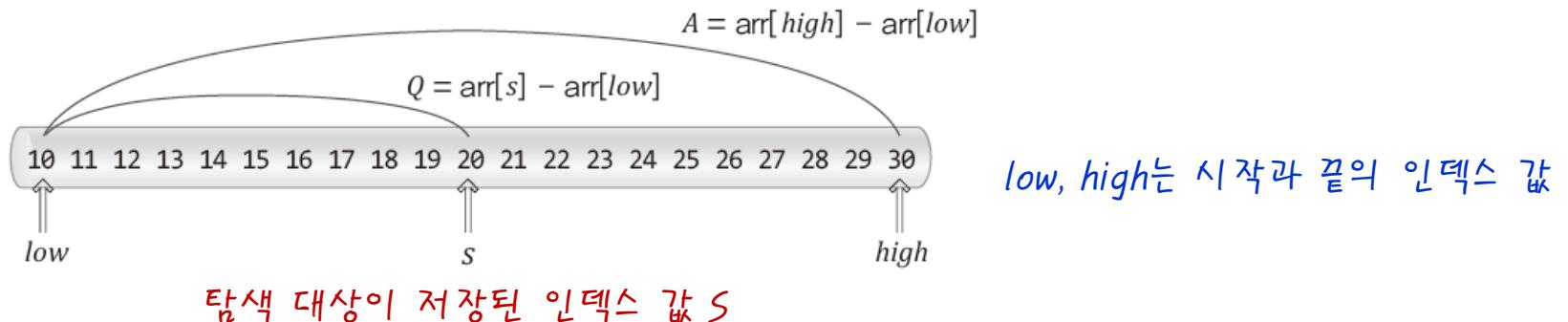


알고리즘 별 탐색 위치의 선택 방법

보간 탐색은 단번에 탐색 대상을 찾을 확률이 어느 정도 존재한다.



보간 탐색: 비례식 구성



$$A : Q = (high-low) : (s-low) \quad \Rightarrow \quad s = \frac{Q}{A} (high-low) + low$$

비례식 구성

$$s = \frac{x - \text{arr}[low]}{\text{arr}[high] - \text{arr}[low]} (high-low) + low$$

탐색 위치의 인덱스 값 계산식



탐색 키와 탐색 데이터

```
typedef Key int;           // 탐색 키에 대한 typedef 선언  
  
typedef Data double;        // 탐색 데이터에 대한 typedef 선언  
  
typedef struct item  
{  
    Key searchKey;          // 탐색 키(search key)  
    Data searchData;        // 탐색 데이터(search data)  
} Item;
```

프로그램 개발에 있어서 등장할 수 있는 유형의 구조체

실제 프로그램 개발에 있어서 탐색의 대상은 '데이터'가 아닌 '키(key)'이다. 다만 학습의 편의를 위해서, 우리는 데이터를 찾는 형태로 간단히 예제를 작성하고 있을뿐이다!



보간 탐색의 구현

이진 탐색

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;
    mid = (first+last) / 2;

    if(ar[mid] == target)
        return mid;
    else if(target < ar[mid])
        return BSearchRecur(ar, first, mid-1, target);
    else
        return BSearchRecur(ar, mid+1, last, target);
}
```

교체하면 보간 탐색이 된다!

mid = ((double)(target-ar[first]) / (ar[last]-ar[first]) *
 (last-first)) + first;

$$s = \frac{x - arr[low]}{arr[high] - arr[low]} (high - low) + low$$



보간 탐색의 구현: 오류의 수정

```
1       4       2
int ISearch(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1; // -1의 반환은 탐색의 실패를 의미
    mid = ((double)(target-ar[first])) / (ar[last]-ar[first]) *
          0      (last-first) + first;

    if(ar[mid] == target)
        return mid;           // 탐색된 타겟의 인덱스 값 반환
    else if(target < ar[mid])
        return ISearch(ar, first, mid-1, target);
    else
        return ISearch(ar, mid+1, last, target);
}
ISearch(arr, 1, 4, 2); 이전 호출과 동일한 인자 전달!
```

탐색 대상이 존재 않는 경우

```
int main(void)
{
    int arr[] = {1, 3, 5, 7, 9};
    . . .
    ISearch(arr, 1, 4, 2);
    . . .
}
```

위의 함수 호출로 *mid*는 0이 된다.

그래서 탈출 조건은 이진 탐색의 경우와 달리해야 한다. 보간 탐색의 탈출 조건은 다음의 특성을 기반으로 구성해야 한다.

“탐색대상이 존재하지 않는 경우, ISearch 함수가 재귀적으로 호출됨에 따라 target에 저장된 값은 first와 last가 가리키는 값의 범위를 넘어서게 된다.”



Chapter 11. 탐색1



Chapter 11-2:

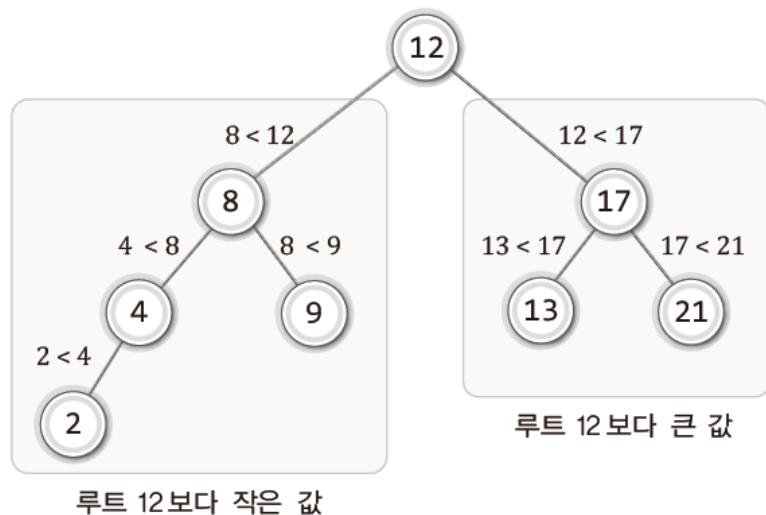
이진 탐색 트리



이진 탐색 트리의 이해

이진 탐색 트리 = 이진 트리 + 데이터의 저장 규칙

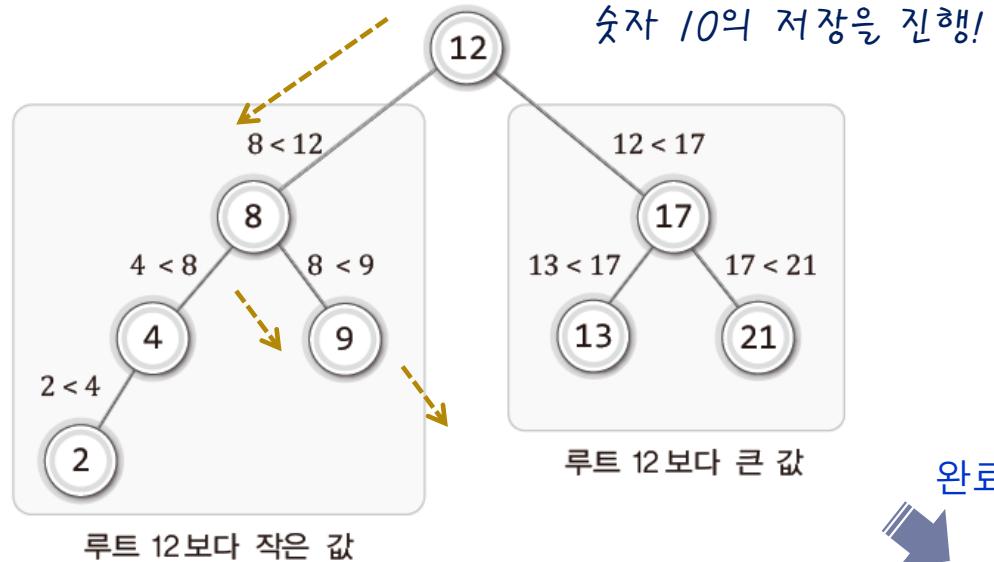
자료구조	위치 정보	데이터의 수	거치는 노드의 수
연결 리스트	있다!	10억 개	최악의 경우 10억 개
이진 탐색 트리	있다!	10억 개	평균 30개 미만



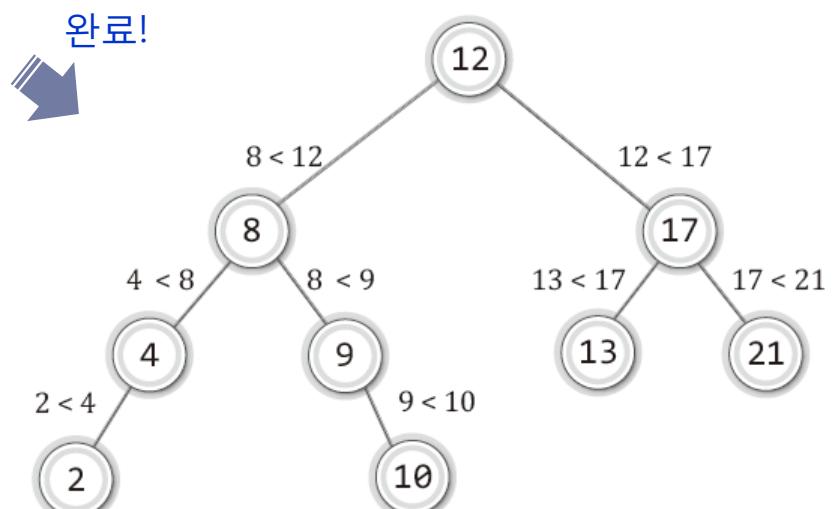
- 이진 탐색 트리의 노드에 저장된 키(key)는 유일!
- 루트 노드의 키 > 왼쪽 서브 트리를 구성하는 키
- 루트 노드의 키 < 오른쪽 서브 트리를 구성하는 키
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리!



이진 탐색 트리의 노드 추가 과정



새 노드의 추가 과정은, 반대로 탐색의 과정이 된다.



이진 탐색 트리의 구현 방안

- 구현 방법 1

이전에 구현한 이진 트리를 참조하여 처음부터 완전히 다시 구현을 한다.

- 구현 방법 2

이진 탐색 트리도 이진 트리이니, 이전에 구현한 이진 트리를 활용하여 구현한다.

BinaryTree2.h와 BinaryTree2.c를 활용하여 구현한다.

두 번째 구현방법의 적용이 가능하다면 당연히 이 방법을 선택해야 한다. 따라서 이 방법을 선택하여
이진 탐색 트리를 구현하기로 하겠다! 그리고 구현의 과정에서 앞서 구현한 '이진 트리를 만드는 도구
의 기능'이 부족할 수 있다. 이러한 경우 도구의 성능을 확장 및 개선 시키면 된다.



이진 탐색 트리의 헤더파일

BinaryTree2.h

- BTTreeNode * MakeBTTreeNode(void);

노드를 동적으로 할당해서 그 노드의 주소 값을 반환한다.

- BTData GetData(BTTreeNode * bt);

노드에 저장된 데이터를 반환한다.

- void SetData(BTTreeNode * bt, BTData data);

인자로 전달된 데이터를 노드에 저장한다.

- BTTreeNode * GetLeftSubTree(BTTreeNode * bt);

인자로 전달된 노드의 왼쪽 자식 노드의 주소 값을 반환한다.

- BTTreeNode * GetRightSubTree(BTTreeNode * bt);

인자로 전달된 노드의 오른쪽 자식 노드의 주소 값을 반환한다.

- void MakeLeftSubTree(BTTreeNode * main, BTTreeNode * sub); **기존 왼쪽 자식 노드 삭제!**

인자로 전달된 노드의 왼쪽 자식 노드를 교체한다.

- void MakeRightSubTree(BTTreeNode * main, BTTreeNode * sub); **기존 오른쪽 자식 노드 삭제!**

인자로 전달된 노드의 오른쪽 자식 노드를 교체한다.

BinarySearchTree.h

삭제 관련 함수는 후에 별도 추가!

// BST의 생성 및 초기화

void BSTMakeAndInit(BTTreeNode ** pRoot);

// 노드에 저장된 데이터 반환

BSTData BSTGetData(BTTreeNode * bst);

// BST를 대상으로 데이터 저장(노드의 생성과정 포함)

void BSTInsert(BTTreeNode ** pRoot, BSTData data);

// BST를 대상으로 데이터 탐색

BTTreeNode * BSTSearch(BTTreeNode * bst, BSTData target);

BinaryTree2.h에 선언된 함수들을 이용해서 위의 함수들을 정의한다!



이진 탐색 트리 기반 main 함수

```
int main(void)
{
    BTTreeNode * bstRoot;          // bstRoot는 BST의 루트 노드를 가리킨다.
    BTTreeNode * sNode;

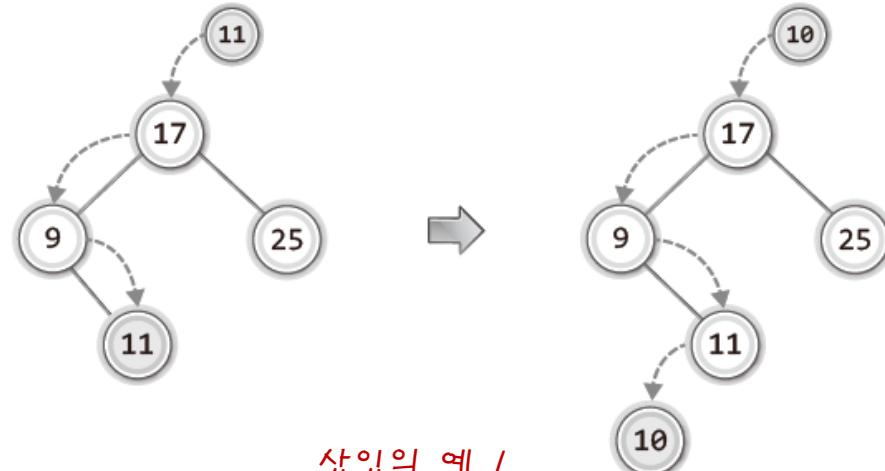
    BSTMakeAndInit(&bstRoot);     // Binary Search Tree의 생성 및 초기화

    BSTInsert(&bstRoot, 1);       // bstRoot에 1을 저장
    BSTInsert(&bstRoot, 2);       // bstRoot에 2를 저장
    BSTInsert(&bstRoot, 3);       // bstRoot에 3을 저장

    // 탐색! 1이 저장된 노드를 찾아서!
    sNode = BSTSearch(bstRoot, 1);      이진 탐색 트리의 사용자 입장에서는 단순히 데이터를
                                         저장하고 탐색하면 된다!
    if(sNode == NULL)
        printf("탐색 실패 \n");           어떠한 형태로 트리가 구성이 되는지 알 필요가 없다!
    else
        printf("탐색에 성공한 키의 값: %d \n", BSTGetData(sNode));
    return 0;
}
```

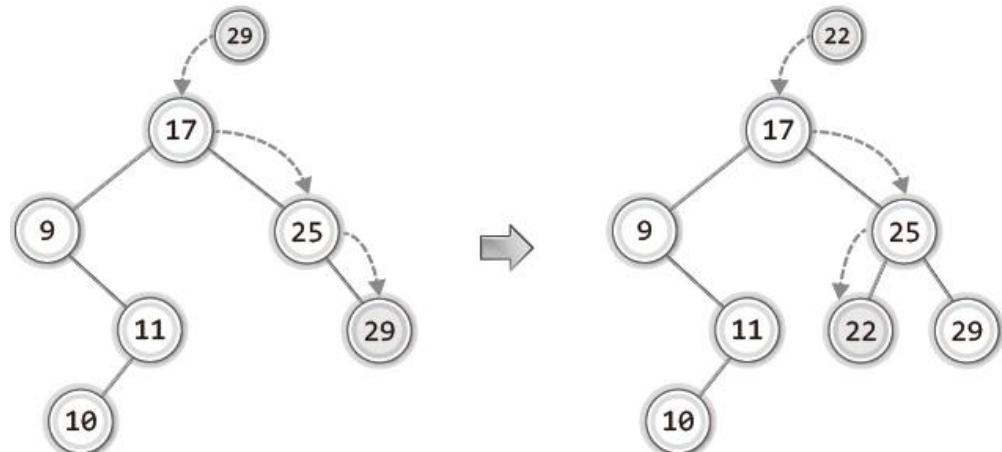


이진 탐색 트리의 구현: 삽입과 탐색



삽입의 예 1

"비교대상이 없을 때까지 내려간다. 그리고 비교대상이 없는 그 위치가 새 데이터가 저장될 위치이다."



삽입의 예 2



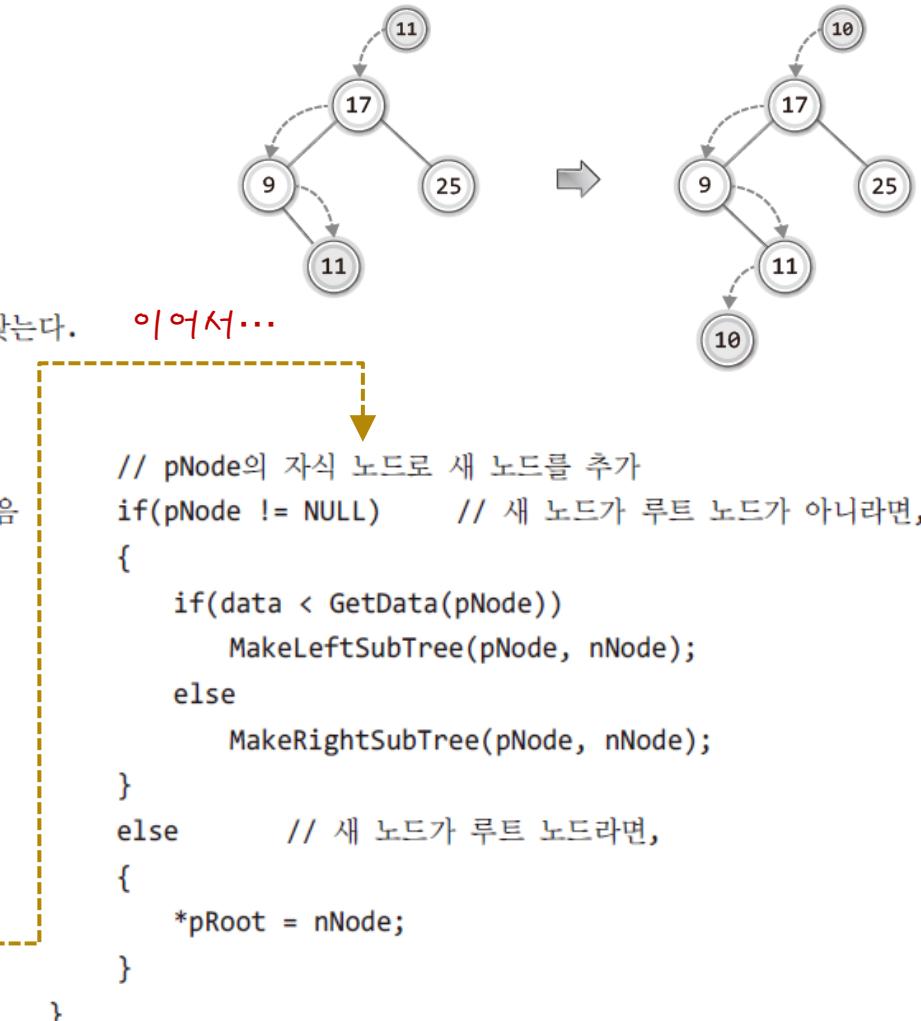
이진 탐색 트리의 구현: 삽입 함수의 구현

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    BTreeNode * pNode = NULL;          // parent node
    BTreeNode * cNode = *pRoot;         // current node
    BTreeNode * nNode = NULL;           // new node

    // 새로운 노드가(새 데이터가 담긴 노드가) 추가될 위치를 찾는다. 이어서...
    while(cNode != NULL)
    {
        if(data == GetData(cNode))
            return; // 데이터의(키의) 중복을 허용하지 않음

        pNode = cNode;

        if(GetData(cNode) > data)
            cNode = GetLeftSubTree(cNode);
        else
            cNode = GetRightSubTree(cNode);
    }
}
```



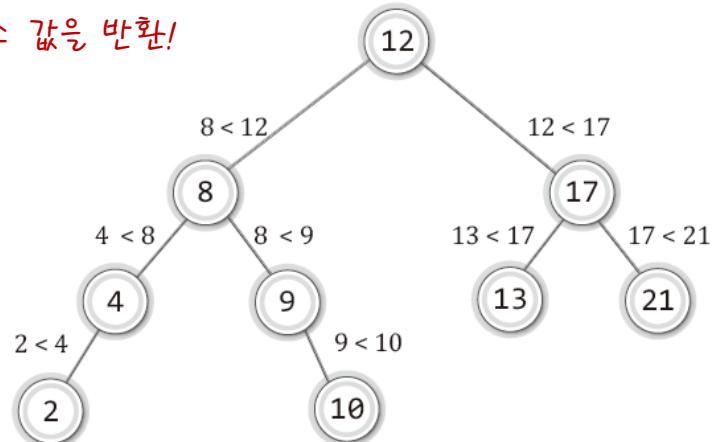
이진 탐색 트리의 구현: 탐색 함수의 구현

```
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target)
{
    BTreeNode * cNode = bst;          // current node
    BSTData cd;                      // current data

    while(cNode != NULL)             삽입의 과정을 근거로 탐색을 진행한다.
    {
        cd = GetData(cNode);         따라서 구현하기가 쉽다!

        if(target == cd)
            return cNode;           탐색에 성공하면 해당 노드의 주소 값을 반환!
        else if(target < cd)
            cNode = GetLeftSubTree(cNode);
        else
            cNode = GetRightSubTree(cNode);
    }

    return NULL;                    // 탐색대상이 저장되어 있지 않음.
}
```



이진 탐색 트리의 구현: 소스파일 & 실행

```
#include "BinaryTree2.h"
#include "BinarySearchTree.h"

void BSTMakeAndInit(BTreeNode ** pRoot)
{
    *pRoot = NULL;
}

BSTData BSTGetData(BTreeNode * bst)
{
    return GetData(bst);
}

void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    // 위에서 소개하였으니 생략합니다.
}

BTreeNode * BSTSearch(BTreeNode * bst, BSTData target)
{
    // 위에서 소개하였으니 생략합니다.
}
```

BinaryTree2.h

BinaryTree2.c

BinarySearchTree.h

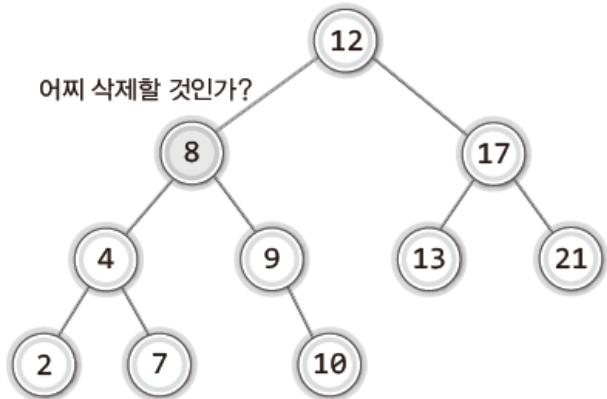
BinarySearchTree.c

BinarySearchTreeMain.c

실행을 위한 파일의 구성!



이진 탐색 트리 삭제 구현: 상황 별 삭제



삭제로 인한 빈 자리를 어떻게 채워야 할까?

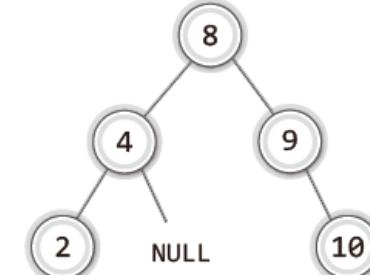
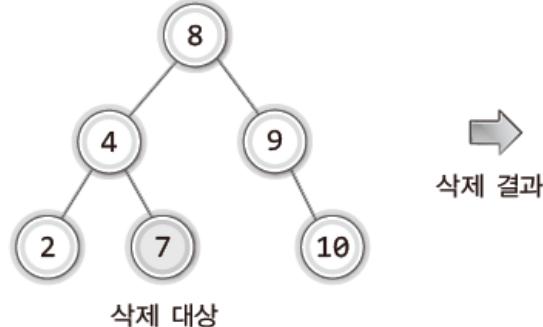
삭제와 관련해서 고려해야 할 세 가지 상황

- 상황 1 삭제할 노드가 단말 노드인 경우
- 상황 2 삭제할 노드가 하나의 자식 노드를(하나의 서브 트리를) 갖는 경우
- 상황 3 삭제할 노드가 두 개의 자식 노드를(두 개의 서브 트리를) 갖는 경우

각 상황 별로 추가로 삭제할 노드가 루트 노드인 경우를 구분해야 하지만 이를 피해가는 형태로 코드를 구성하기로 한다!



이진 탐색 트리 삭제 구현: 상황 1



삭제할 노드가 단말 노드인 경우!
가장 쉽게 삭제가 가능한 상황이다!

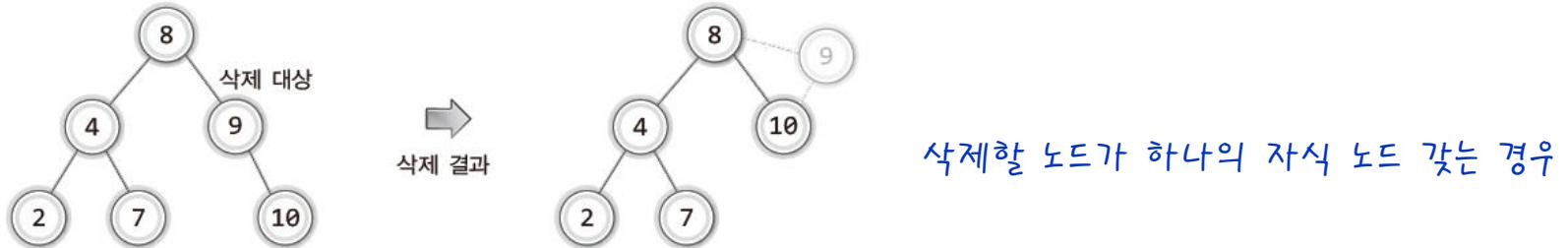
코드레벨 구현

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수  
if(삭제할 노드가 단말 노드이다!)  
{  
    if(GetLeftSubTree(pNode) == dNode)          // 삭제할 노드가 왼쪽 자식 노드라면,  
        RemoveLeftSubTree(pNode);                // 왼쪽 자식 노드 트리에서 제거  
    else                                         // 삭제할 노드가 오른쪽 자식 노드라면,  
        RemoveRightSubTree(pNode);               // 오른쪽 자식 노드 트리에서 제거  
}
```

RemoveLeftSubTree, RemoveRightSubTree 함수는 BinaryTree2.h와 BinaryTree2.c에 추가 할 함수



이진 탐색 트리 삭제 구현: 상황 2

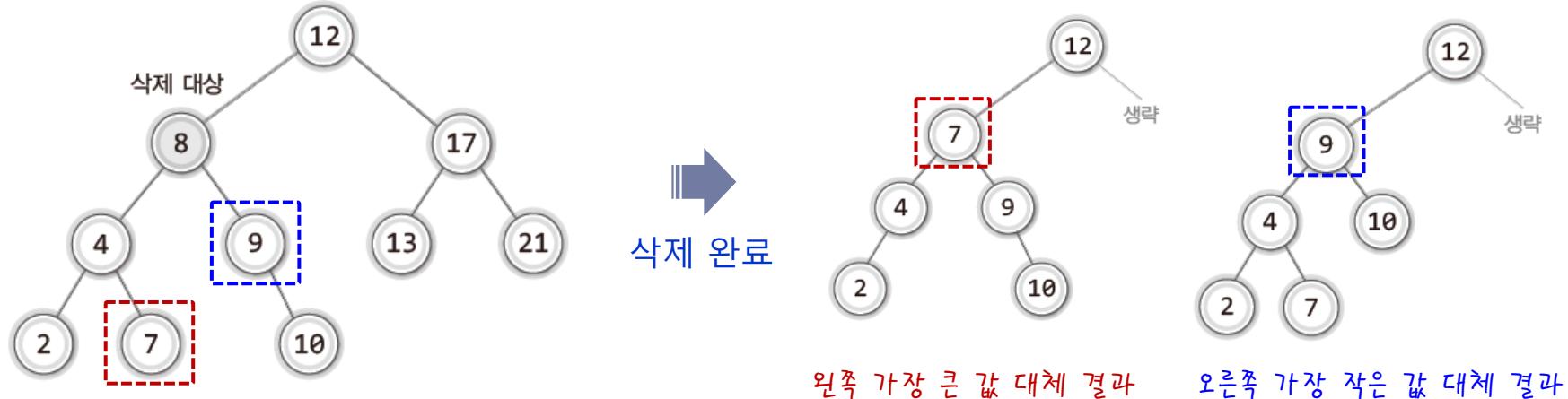


```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수  
if(삭제할 노드가 하나의 자식 노드를 지닌다!)  
{  
    BTreeNode * dcNode; // 삭제 대상의 자식 노드를 가리키는 포인터 변수  
  
    // 삭제 대상의 자식 노드를 찾는다.  
    if(GetLeftSubTree(dNode) != NULL) // 자식 노드가 왼쪽에 있다면,  
        dcNode = GetLeftSubTree(dNode);  
    else // 자식 노드가 오른쪽에 있다면,  
        dcNode = GetRightSubTree(dNode);  
  
    // 삭제 대상의 부모 노드와 자식 노드를 연결한다.  
    if(GetLeftSubTree(pNode) == dNode) // 삭제 대상이 왼쪽 자식 노드이면,  
        ChangeLeftSubTree(pNode, dcNode); // 왼쪽으로 연결  
    else // 삭제 대상이 오른쪽 자식 노드이면,  
        ChangeRightSubTree(pNode, dcNode); // 오른쪽으로 연결  
}
```

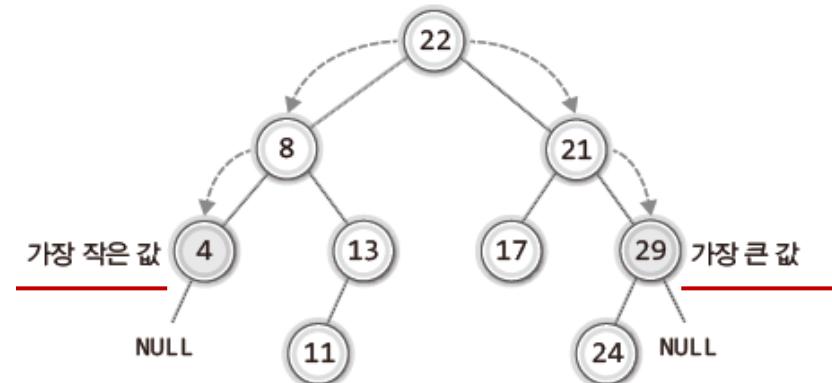
코드레벨 구현

ChangeLeftSubTree, ChangeRightSubTree 함수는
BinaryTree2.h와 BinaryTree2.c에 추가 할 함수

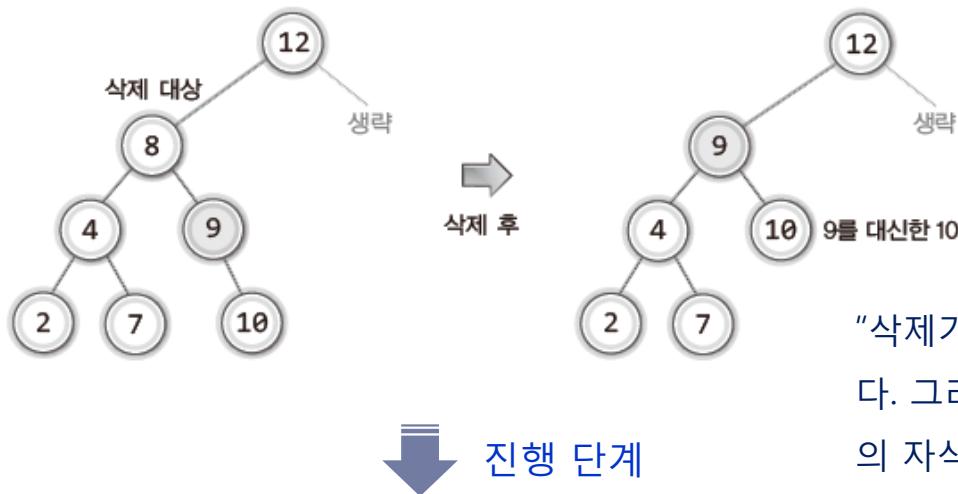
이진 탐색 트리 삭제 구현: 상황 3



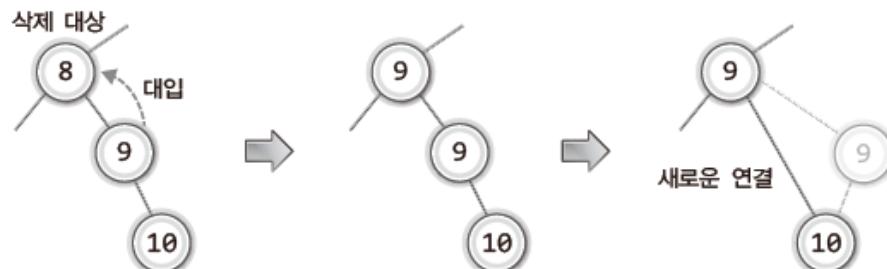
왼쪽 서브 트리에서 가장 큰 값,
또는 오른쪽 서브 트리에서 가장 작은 값으로 대체



이진 탐색 트리 삭제 구현: 적용 모델



“삭제가 되는 8이 저장된 노드를 9가 저장된 노드로 대체한다. 그리고 이로 인해서 생기는 빈 자리는 9가 저장된 노드의 자식 노드로 대체한다.”



- 단계 1 삭제할 노드를 대체할 노드를 찾는다.
- 단계 2 대체할 노드에 저장된 값을 삭제할 노드에 대입한다.
- 단계 3 대체할 노드의 부모 노드와 자식 노드를 연결한다.



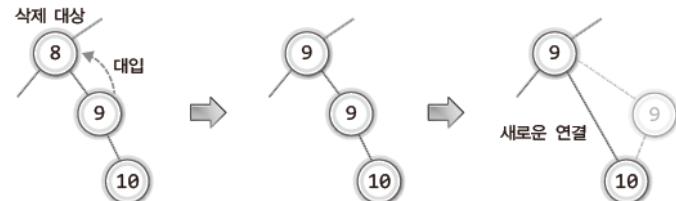
이진 탐색 트리 삭제 구현: 상황 3의 구현

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수  
if(삭제할 노드가 두 개의 자식 노드를 지닌다)  
{  
    BTreeNode * mNode = GetRightSubTree(dNode); // mNode는 대체 노드 가리킴  
    BTreeNode * mpNode = dNode; // mpNode는 대체 노드의 부모 노드 가리킴  
    . . .  
    // 단계 1. 삭제 대상의 대체 노드를 찾는다.  
    while(GetLeftSubTree(mNode) != NULL)  
    {  
        mpNode = mNode;  
        mNode = GetLeftSubTree(mNode);  
    }  
    // 단계 2. 대체할 노드에 저장된 값을 삭제할 노드에 대입한다.  
    SetData(dNode, GetData(mNode));  
    // 단계 3. 대체할 노드의 부모 노드와 자식 노드를 연결한다.  
    if(GetLeftSubTree(mpNode) == mNode) // 대체할 노드가 왼쪽 자식 노드라면  
    {  
        // 대체할 노드의 자식 노드를 부모 노드의 왼쪽에 연결  
        ChangeLeftSubTree(mpNode, GetRightSubTree(mNode));  
    }  
    else // 대체할 노드가 오른쪽 자식 노드라면  
    {  
        // 대체할 노드의 자식 노드를 부모 노드의 오른쪽에 연결  
        ChangeRightSubTree(mpNode, GetRightSubTree(mNode));  
    }  
    . . .  
}
```

자식 노드가 있다면 오른쪽 자식 노드이다!

이어서...

```
}  
else // 대체할 노드가 오른쪽 자식 노드라면  
{  
    // 대체할 노드의 자식 노드를 부모 노드의 오른쪽에 연결  
    ChangeRightSubTree(mpNode, GetRightSubTree(mNode));  
}  
. . .  
자식 노드가 있다면 오른쪽 자식 노드이다!
```



삭제 구현을 위한 이진 트리의 확장

BinaryTree2.h와 BinaryTree2.c에 추가되는 함수들!

// 왼쪽 자식 노드를 트리에서 제거, 제거된 노드의 주소 값이 반환된다.

- BTreenode * RemoveLeftSubTree(BTreenode * bt);

BinaryTree3.h의 일부

// 오른쪽 자식 노드를 트리에서 제거, 제거된 노드의 주소 값이 반환된다.

- BTreenode * RemoveRightSubTree(BTreenode * bt);

// 메모리 소멸을 수반하지 않고 main의 왼쪽 자식 노드를 변경한다.

- void ChangeLeftSubTree(BTreenode * main, BTreenode * sub);

// 메모리 소멸을 수반하지 않고 main의 오른쪽 자식 노드를 변경한다.

- void ChangeRightSubTree(BTreenode * main, BTreenode * sub);

필요에 의해서 앞서 만들어 놓은 도구의 기능을 확장 및 추가한 것으로 이는 타당한 접근 방법으로 보아야 옳다!



확장된 함수의 구현

```
// 메모리의 소멸을 수반하지 않고 main의 왼쪽 자식 노드를 변경한다.  
void ChangeLeftSubTree(BTreeNode * main, BTreeNode * sub)  
{  
    main->left = sub;  
}  
  
// 메모리의 소멸을 수반하지 않고 main의 오른쪽 자식 노드를 변경한다.  
void ChangeRightSubTree(BTreeNode * main, BTreeNode * sub)  
{  
    main->right = sub;  
}
```

BinaryTree3.c에 정의된 네개의 함수

```
// 왼쪽 자식 노드 제거, 제거된 노드의 주소 값이 반환된다.  
BTreeNode * RemoveLeftSubTree(BTreeNode * bt)  
{  
    BTreeNode * delNode;  
  
    if(bt != NULL) {  
        delNode = bt->left;  
        bt->left = NULL;  
    }  
    return delNode;  
}  
  
// 오른쪽 자식 노드 제거, 제거된 노드의 주소 값이 반환된다.  
BTreeNode * RemoveRightSubTree(BTreeNode * bt)  
{  
    BTreeNode * delNode;  
  
    if(bt != NULL) {  
        delNode = bt->right;  
        bt->right = NULL;  
    }  
    return delNode;  
}
```



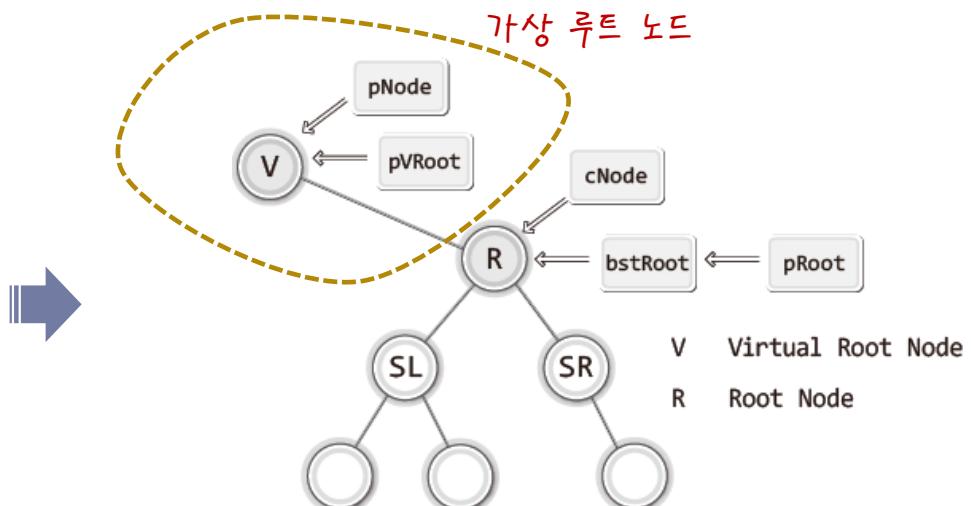
이진 탐색 트리 삭제의 완전한 구현: 초기화

```
BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    BTreeNode * pVRoot = MakeBTreeNode();
    BTreeNode * pNode = pVRoot;
    BTreeNode * cNode = *pRoot;
    BTreeNode * dNode;

    ChangeRightSubTree(pVRoot, *pRoot);
    . . .
}
```

// bstRoot에 루트 노드의 주소값 저장
BSTRemove(&bstRoot, 7);

가상 루트 노드를 형성한 이유는 삭제할 노드가 루트 노드인 경우의 예외적인 삭제흐름을 일반화하기 위함이다.



이진 탐색 트리 삭제의 완전한 구현: 삭제 대상 찾기

```
// 삭제 대상인 노드를 탐색
while(cNode != NULL && GetData(cNode) != target)
{
    pNode = cNode;

    if(target < GetData(cNode))
        cNode = GetLeftSubTree(cNode);
    else
        cNode = GetRightSubTree(cNode);
}

if(cNode == NULL)      // 삭제 대상이 존재하지 않는다면,
    return NULL;

dNode = cNode;          // 삭제 대상을 dNode가 가리키게 한다.
```

cNode의 부모 노드를 pNode가 가리키게 해야 하기 때문에 이 부분을 BSTSearch 함수의 호출로 대신할 수 없다!

여기까지가 실제 삭제의 상황 1 or 상황 2 or 상황 3 의 진행을 위한 준비과정이다!



이진 탐색 트리 삭제의 완전한 구현: 상황 1

앞서 작성한 코드

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수  
if(삭제할 노드가 단말 노드이다!)  
{  
    if(GetLeftSubTree(pNode) == dNode)      // 삭제할 노드가 왼쪽 자식 노드라면,  
        RemoveLeftSubTree(pNode);           // 왼쪽 자식 노드 트리에서 제거  
    else                                  // 삭제할 노드가 오른쪽 자식 노드라면,  
        RemoveRightSubTree(pNode);          // 오른쪽 자식 노드 트리에서 제거  
}
```

상황 1의 완성된 코드

```
// 첫 번째 경우: 삭제 대상이 단말 노드인 경우  
if(GetLeftSubTree(dNode) == NULL && GetRightSubTree(dNode) == NULL)  
{  
    if(GetLeftSubTree(pNode) == dNode)  
        RemoveLeftSubTree(pNode);  
    else  
        RemoveRightSubTree(pNode);  
}
```



이진 탐색 트리 삭제의 완전한 구현: 상황 2

앞서 작성한 코드

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수  
if(삭제할 노드가 하나의 자식 노드를 지닌다!)  
{  
    BTreeNode * dcNode; // 삭제 대상의 자식 노드를 가리키는 포인터 변수  
  
    // 삭제 대상의 자식 노드를 찾는다  
    if(GetLeftSubTree(dNode) != NULL)  
        dcNode = GetLeftSubTree(dNode);  
    else  
        dcNode = GetRightSubTree(dNode);  
  
    // 삭제 대상의 부모 노드와 자식 노드를 바꾼다  
    if(GetLeftSubTree(pNode) == dNode)  
        ChangeLeftSubTree(pNode, dcNode);  
    else  
        ChangeRightSubTree(pNode, dcNode);  
}
```

상황 2의 완성된 코드

```
// 두 번째 경우: 삭제 대상이 하나의 자식 노드를 갖는 경우  
else if(GetLeftSubTree(dNode) == NULL || GetRightSubTree(dNode) == NULL)  
{  
    BTreeNode * dcNode; // 삭제 대상의 자식 노드 가리킴  
  
    if(GetLeftSubTree(dNode) != NULL)  
        dcNode = GetLeftSubTree(dNode);  
    else  
        dcNode = GetRightSubTree(dNode);  
  
    if(GetLeftSubTree(pNode) == dNode)  
        ChangeLeftSubTree(pNode, dcNode);  
    else  
        ChangeRightSubTree(pNode, dcNode);  
}
```



이진 탐색 트리 삭제의 완전한 구현: 상황 3

```
// 세 번째 경우: 두 개의 자식 노드를 모두 갖는 경우
else
{
    BTreeNode * mNode = GetRightSubTree(dNode);      // 대체 노드 가리킴
    BTreeNode * mpNode = dNode;          // 대체 노드의 부모 노드 가리킴
    int delData;

    // 삭제 대상의 대체 노드를 찾는다.
    while(GetLeftSubTree(mNode) != NULL) {

        mpNode = mNode;
        mNode = GetLeftSubTree(mNode);
    }

    // 대체 노드에 저장된 값을 삭제할 노드에 대입한다.
    // 대체 노드의 부모 노드와 자식 노드를 연결한다.
    if(GetLeftSubTree(mpNode) == mNode)
        ChangeLeftSubTree(mpNode, GetRightSubTree(mNode));
    else
        ChangeRightSubTree(mpNode, GetRightSubTree(mNode));

    dNode = mNode;
    SetData(dNode, delData);      // 백업 데이터 복원
}
```

이어서...



이진 탐색 트리 삭제의 완전한 구현: 마무리

```
// 삭제된 노드가 루트 노드인 경우에 대한 추가적인 처리  
if(GetRightSubTree(pVRoot) != *pRoot)  
    *pRoot = GetRightSubTree(pVRoot); // 루트 노드의 변경을 반영  
  
free(pVRoot); // 가상 루트 노드의 소멸  
return dNode; // 삭제 대상의 반환  
}
```

[BinaryTree3.h](#)
[BinaryTree3.c](#)

[BinarySearchTree2.h](#)
[BinarySearchTree2.c](#)
[BinarySearchTreeMain.c](#)

실행을 위한 파일의 구성!



수고하셨습니다~



Chapter 11에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 12. 탐색2

Introduction To Data Structures Using C

Chapter 12. 탐색2



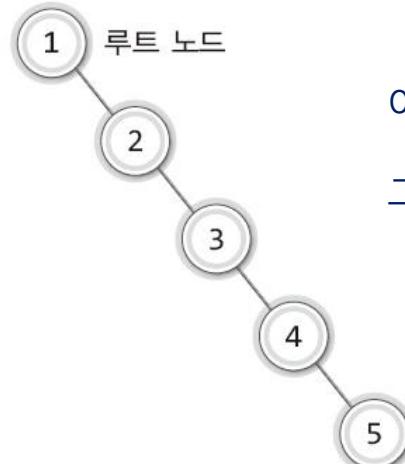
Chapter 12-1:

균형 잡힌 이진 탐색 트리: AVL 트리의 이해



이진 탐색 트리의 문제점과 AVL 트리

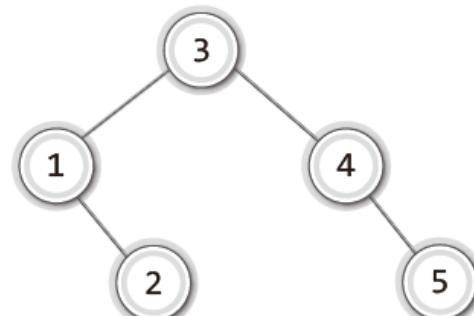
/부터 5까지 순서대로 저장이 이뤄진 경우!



이진 탐색 트리의 탐색 연산은 $O(\log_2 n)$ 의 시간 복잡도를 보인다.

그러나 균형이 맞지 않을수록 $O(n)$ 에 가까운 시간 복잡도를 보인다.

3이 제일 먼저 저장된 경우!



이진 탐색 트리의 균형 문제를 해결한 트리

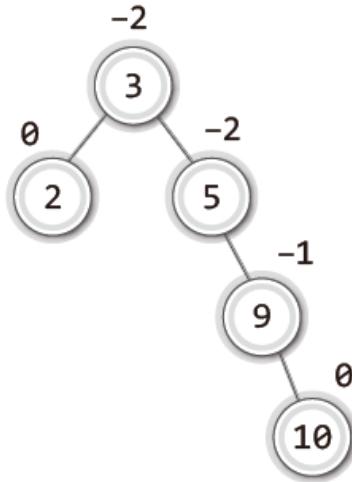
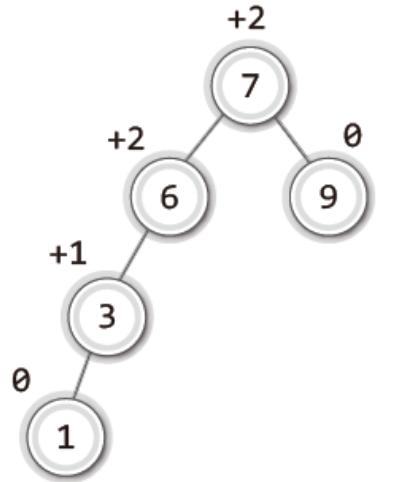
- AVL 트리
- 2-3-4 트리
- Red-Black 트리
- 등등...

약간의 순서 변화로 균형이 잡혔다!



자동으로 균형 잡는 AVL 트리와 균형 인수

균형 인수 = 왼쪽 서브 트리의 높이 - 오른쪽 서브 트리의 높이



AVL 트리는 균형 인수(Balance Factor)를 기준으로 트리의 균형을 잡기 위한 재조정(리밸런싱)의 진행 시기를 결정한다.



리밸런싱이 필요한 첫 번째 상태와 LL회전



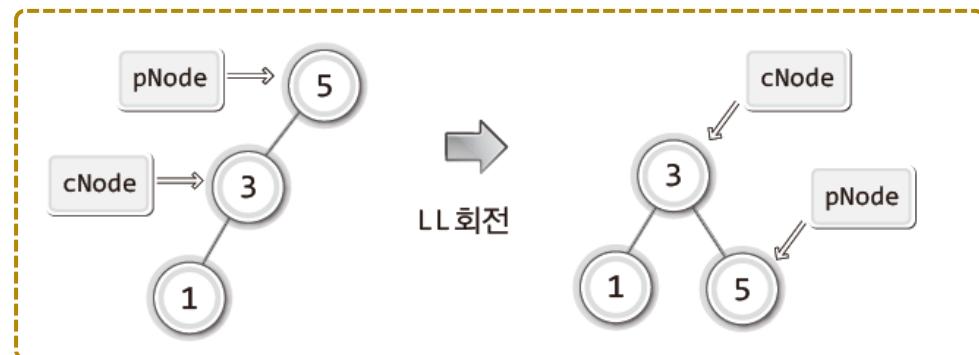
▶ [그림 12-4: LL회전의 방법과 그 결과]

“5가 저장된 노드의 **왼쪽(Left)**에 3이 저장된 자식 노드가 하나 존재하고, 그 자식 노드의 **왼쪽(Left)**에 1이 저장된 자식 노드가 또 하나 존재한다.”

이러한 상태를 균형 잡기 위해서 회전을 진행한다.

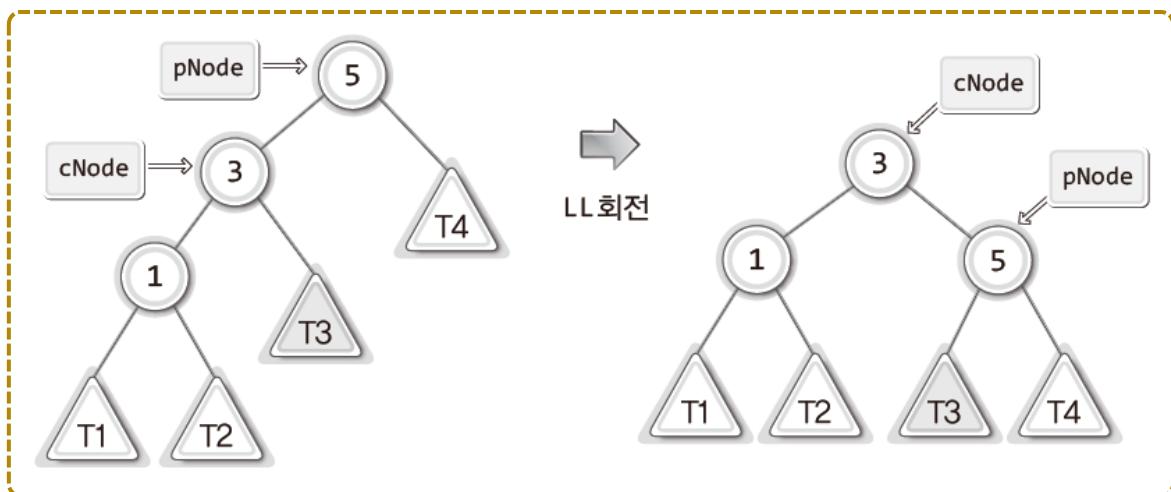
LL상태를 균형 잡기 위한 LL회전

단순한 예



`ChangeLeftSubTree(cNode, pNode);`

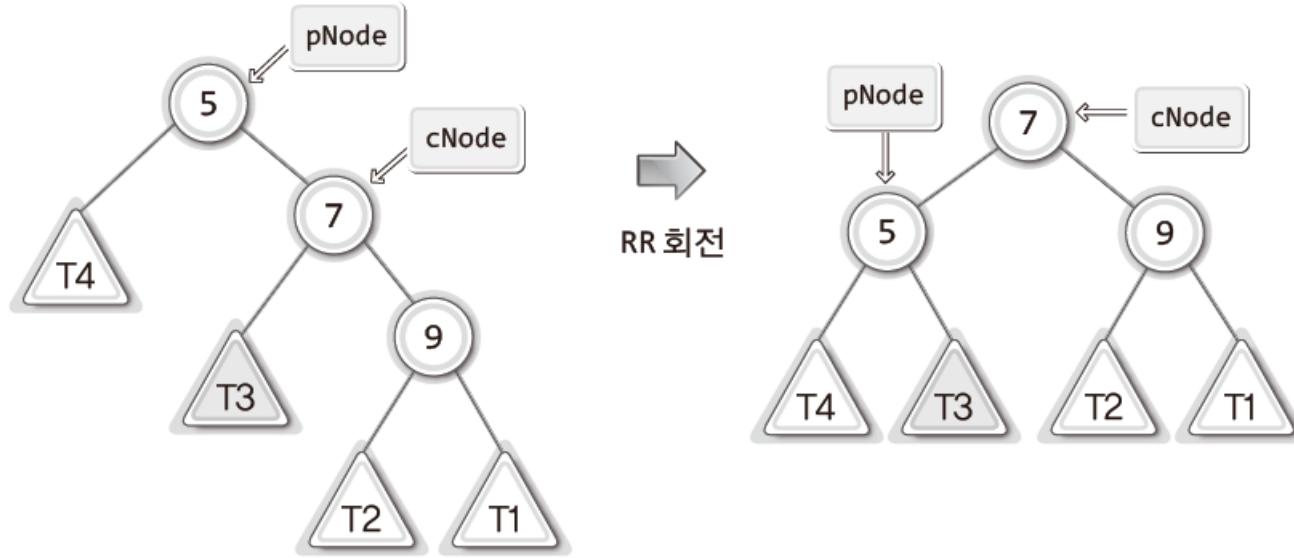
단순한 예의 일반화



`ChangeLeftSubTree(pNode, GetRightSubTree(cNode));`
`ChangeRightSubTree(cNode, pNode);`



RR상태와 RR회전

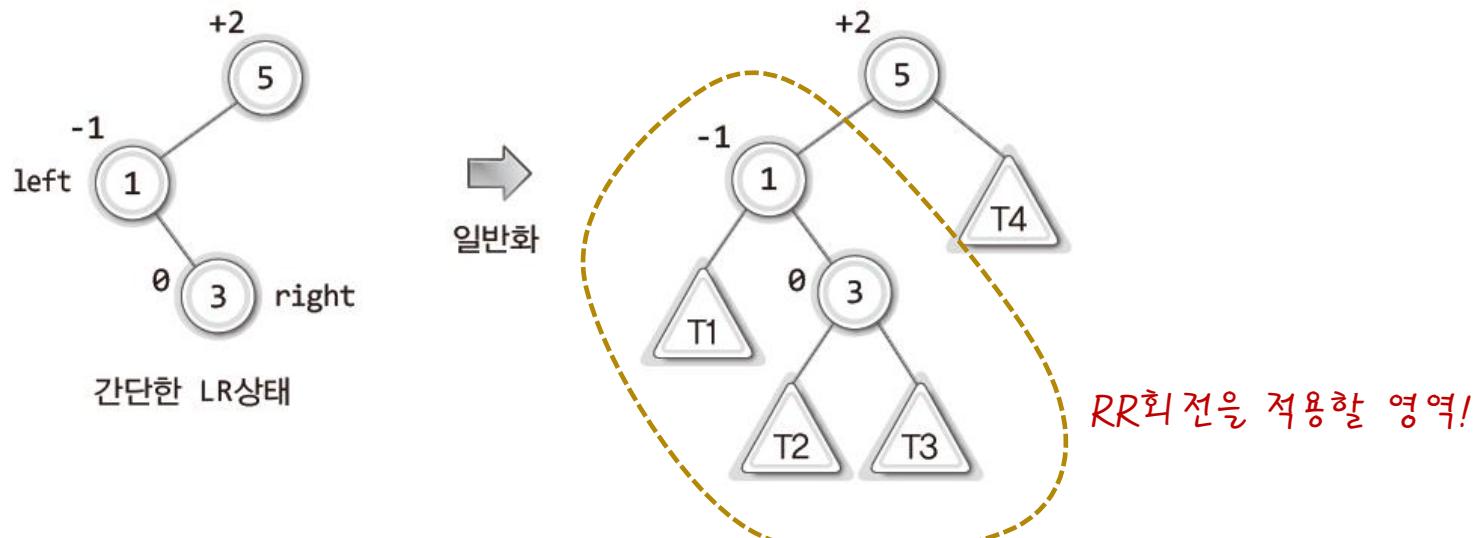


```
ChangeRightSubTree(pNode, GetLeftSubTree(cNode));  
ChangeLeftSubTree(cNode, pNode);
```



LR상태

LL상태 그리고 RR상태와 같이 한 번의 회전으로 균형을 잡을 수 없다. 따라서 LR 상태는 한 번의 회전으로 균형이 잡히는 LL상태 또는 RR상태가 되도록 하는 것이 우선이다!



LR상태는 RR회전을 통해서(RR회전의 부수적인 효과를 이용해서) LL상태가 되게 할 수 있다.



RR회전의 부수적인 효과



일반적인 RR회전



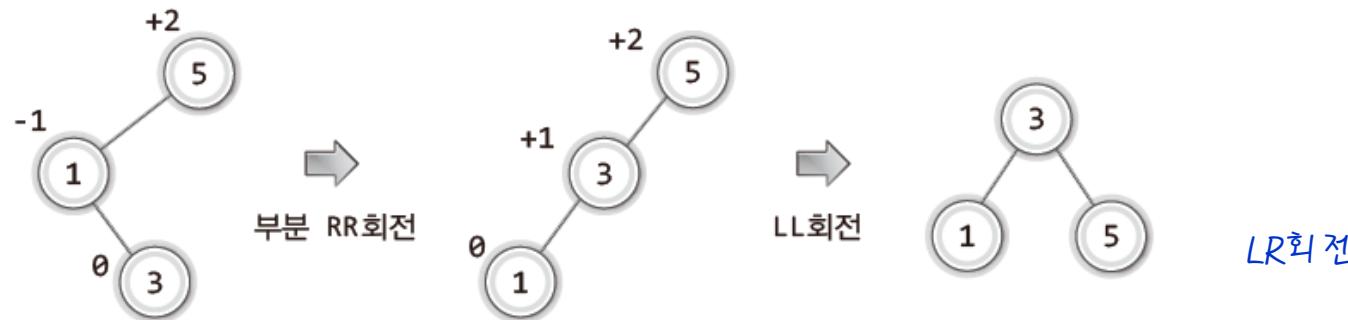
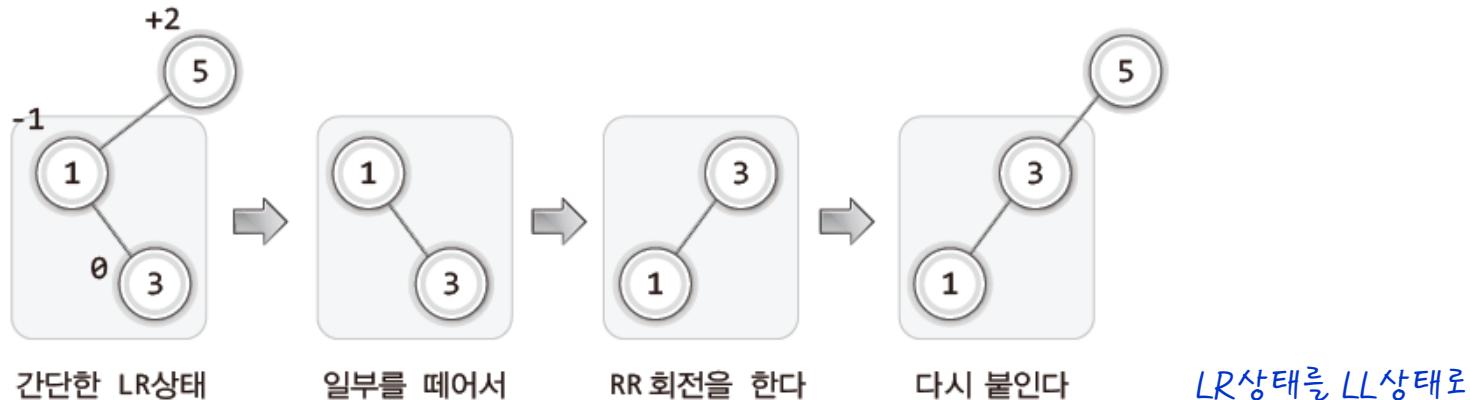
단말 노드가 NULL인 경우에도 RR 회전 가능하다!



부모 자식의 관계가 바뀌는 부수적인 효과,

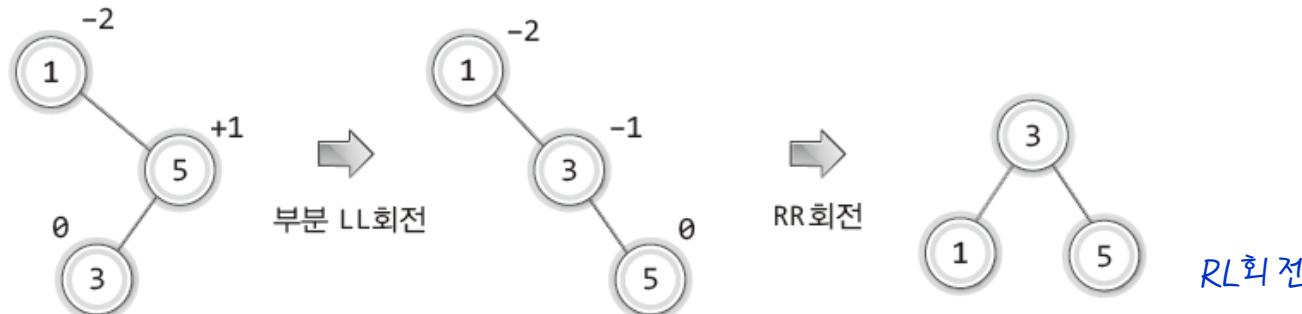
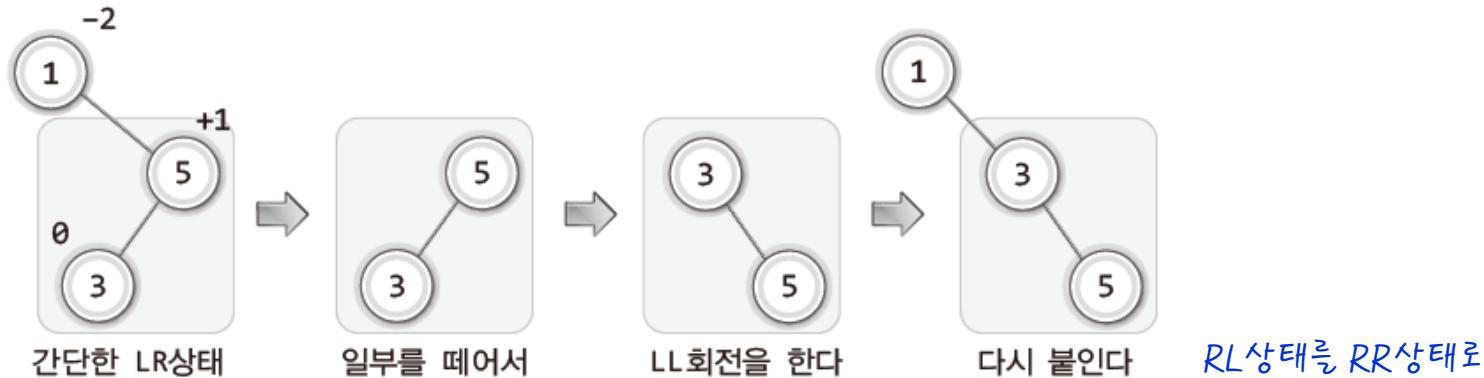


LR회전



RL상태와 RL회전

LR상태 LR회전, 그리고 RL상태 RL회전은 방향에서만 차이를 보인다.



Chapter 12. 트리2



Chapter 12-2:

AVL 트리의 구현



AVL 트리를 어떻게 구현할 것인가?

활용할 파일들

AVL 트리도 이진 탐색 트리의 일종이므로 앞서 구현한 이진 탐색 트리를 기반으로 구현한다.

BinarySearchTree2.c에 리밸런싱 기능을 추가하여, 파일의 이름을 **BinarySearchTree3.c**로 변경하자!

단 다음 두 파일을 추가하여 리밸런싱 도구를 정의하기로 하겠다.

새로 작성할 파일들

- AVLRebalance.h 리밸런싱 관련 함수들의 선언
 - AVLRebalance.c 리밸런싱 관련 함수들의 정의



AVL 트리 구현을 위한 확장 포인트

확장할 함수들

- BSTInsert 함수 트리에 노드를 추가
- BSTRemove 함수 트리에서 노드를 제거

균형은 노드의 삽입과 삭제의 순간에 깨지게 된다.

확장의 형태

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    . . .
    Rebalance(pRoot);      // 노드 추가 후 리밸런싱!
}

BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    . . .
    Rebalance(pRoot);      // 노드 제거 후 리밸런싱!
    return dNode;
}
```



리밸런싱 도구: 균형을 이루고 있는가?

```
// 두 서브 트리의 '높이의 차(균형 인수)'를 반환
int GetHeightDiff(BTreeNode * bst)
{
    int lsh;      // left sub tree height
    int rsh;      // right sub tree height

    if(bst == NULL)
        return 0;

    lsh = GetHeight(GetLeftSubTree(bst));
    rsh = GetHeight(GetRightSubTree(bst));
    return lsh - rsh;    // 균형 인수 계산결과 반환
}
```

모든 경로의 높이를 비교하기 위한 재귀적 구성

```
// 트리의 높이를 계산하여 반환
int GetHeight(BTreeNode * bst)
{
    int leftH;    // left height
    int rightH;   // right height

    if(bst == NULL)
        return 0;

    leftH = GetHeight(GetLeftSubTree(bst));
    rightH = GetHeight(GetRightSubTree(bst));

    // 큰 값의 높이를 반환한다.
    if(leftH > rightH)
        return leftH + 1;
    else
        return rightH + 1;
}
```



리밸런싱 도구: LL회전

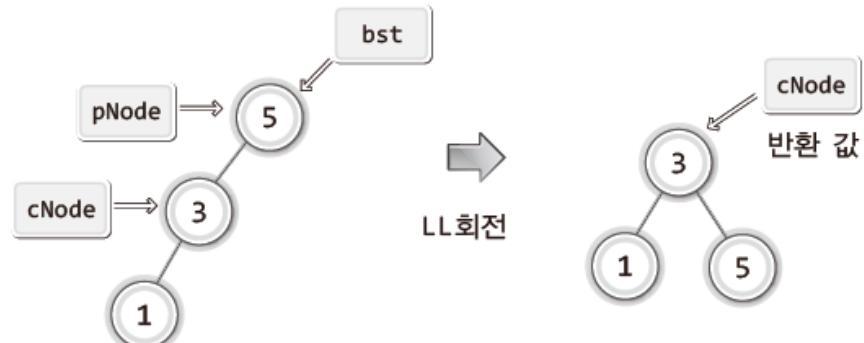
```
BTreeNode * RotateLL(BTreeNode * bst)      // LL회전을 담당하는 함수
{
    BTreeNode * pNode;          // parent node
    BTreeNode * cNode;          // child node

    // pNode와 cNode가 LL회전을 위해 적절한 위치를 가리키게 한다.
    pNode = bst;
    cNode = GetLeftSubTree(pNode);

    // 실제 LL회전을 담당하는 두 개의 문장
    ChangeLeftSubTree(pNode, GetRightSubTree(cNode));
    ChangeRightSubTree(cNode, pNode);

    // LL회전으로 인해서 변경된 루트 노드의 주소 값 반환
    return cNode;
}
```

회전 후 루트 노드가 변경되기 때문에
새로운 루트 노드의 주소 값을 반환해준다.



리밸런싱 도구: RR회전

```
BTreeNode * RotateRR(BTreeNode * bst)    // RR회전을 담당하는 함수
{
    BTreeNode * pNode;          // parent node
    BTreeNode * cNode;          // child node

    // pNode와 cNode가 RR회전을 위해 적절한 위치를 가리키게 한다.
    pNode = bst;
    cNode = GetRightSubTree(pNode);

    // 실제 RR회전을 담당하는 두 개의 문장
    ChangeRightSubTree(pNode, GetLeftSubTree(cNode));
    ChangeLeftSubTree(cNode, pNode);

    // RR회전으로 인해서 변경된 루트 노드의 주소 값 반환
    return cNode;
}
```

방향에 있어서만 차이를 보인다.

```
// 실제 LL회전을 담당하는 두 개의 문장
ChangeLeftSubTree(pNode, GetRightSubTree(cNode));
ChangeRightSubTree(cNode, pNode);
```



리밸런싱 도구: LR회전

// 부분적 RR회전에 이어서 LL회전을 진행

```
BTreeNode * RotateLR(BTreeNode * bst) // LR회전을 담당하는 함수
```

```
{
```

```
    BTreeNode * pNode; // parent node
```

```
    BTreeNode * cNode; // child node
```

// pNode와 cNode가 LR회전을 위해 적절한 위치를 가리키게 한다.

```
pNode = bst;
```

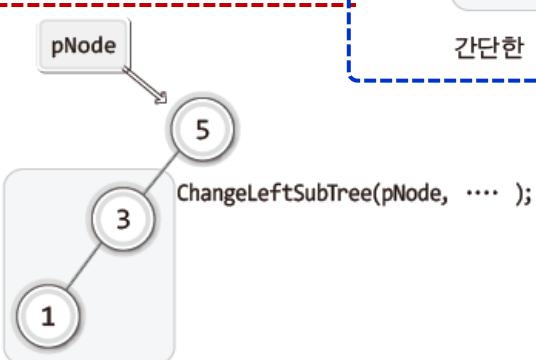
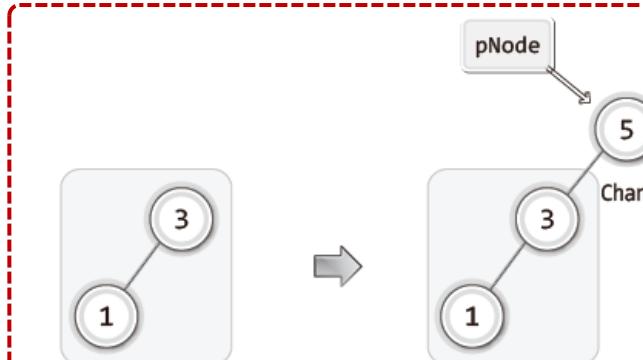
```
cNode = GetLeftSubTree(pNode);
```

// 실제 LR회전을 담당하는 두 개의 문장

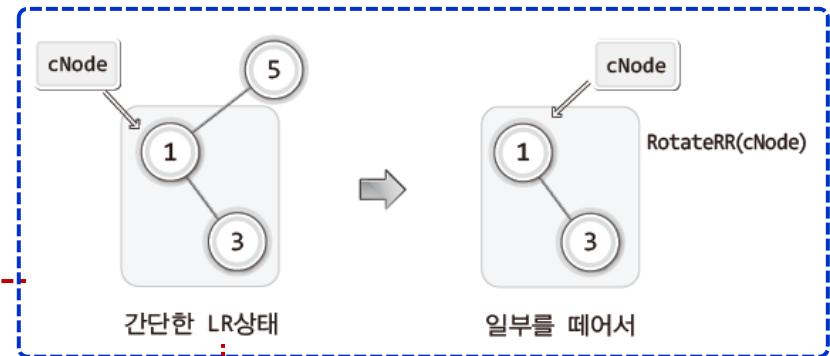
```
ChangeLeftSubTree(pNode, RotateRR(cNode));
```

```
return RotateLL(pNode);
```

```
}
```



다시 불인다



리밸런싱 도구: RL회전

```
// 부분적 LL회전에 이어서 RR회전을 진행
BTreeNode * RotateRL(BTreeNode * bst)
{
    BTreeNode * pNode;           // parent node
    BTreeNode * cNode;          // child node

    // pNode와 cNode가 RL회전을 위해 적절한 위치를 가리키게 한다.
    pNode = bst;
    cNode = GetRightSubTree(pNode); LR회전에서는 GetLeftSubTree 호출

    // 실제 RL회전을 담당하는 두 개의 문장
    ChangeRightSubTree(pNode, RotateLL(cNode));      // 부분적 LL회전
    return RotateRR(pNode);                          // RR회전
}

// 실제 LR회전을 담당하는 두 개의 문장
ChangeLeftSubTree(pNode, RotateRR(cNode));        // 부분적 RR회전
return RotateLL(pNode);                           // LL회전
```

방향에 있어서만 차이를 보인다.



리밸런싱 도구: Rebalance 함수

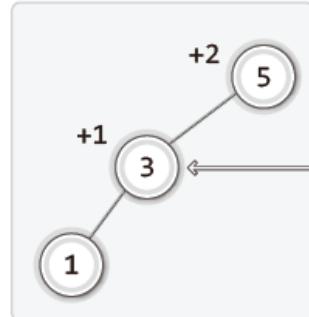
```
BTreeNode * Rebalance(BTreeNode ** pRoot)
{
    int hDiff = GetHeightDiff(*pRoot);      // 균형 인수 계산
    // 균형 인수가 +2 이상이면 LL상태 또는 LR상태이다.
    if(hDiff > 1)      // 왼쪽 서브 트리 방향으로 높이가 2 이상 크다면,
    {
        if(GetHeightDiff(GetLeftSubTree(*pRoot)) > 0)
            *pRoot = RotateLL(*pRoot);          // 균형 인수가 +2 이상이면 왼쪽으로 길게 불균형
        else
            *pRoot = RotateLR(*pRoot);          // 을 이룬 상태이므로 LL 또는 LR상태이다!
    }

    // 균형 인수가 -2 이하이면 RR상태 또는 RL상태이다.
    if(hDiff < -1)      // 오른쪽 서브 트리 방향으로 2 이상 크다면,
    {
        if(GetHeightDiff(GetRightSubTree(*pRoot)) < 0)
            *pRoot = RotateRR(*pRoot);          // 균형 인수가 -2 이하면 오른쪽으로 길게 불균형
        else
            *pRoot = RotateRL(*pRoot);          // 을 이룬 상태이므로 RR 또는 RL상태이다!
    }
}

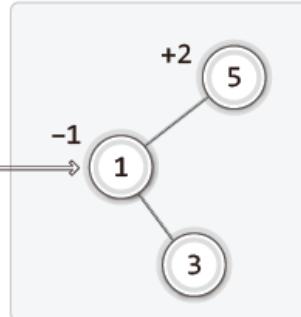
return *pRoot;
}
```



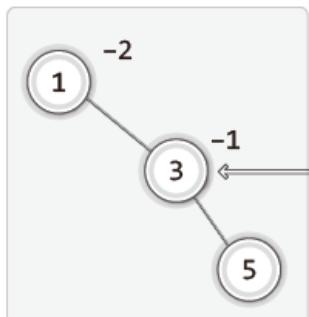
LL상태와 LR상태, 그리고 RR상태 RL상태의 구분



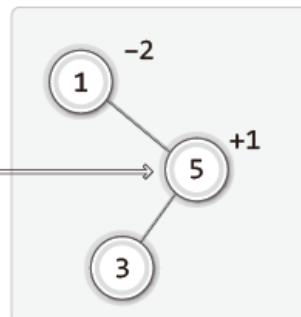
LL상태와 LR상태의
구분기준



GetHeightDiff(GetLeftSubTree(*pRoot))의 반환
값이 0보다 크면 LL상태 그렇지 않으면 LR상태



RR 상태와 RL상태의
구분기준



GetHeightDiff(GetRightSubTree(*pRoot))의 반환
값이 0보다 작으면 RR상태 그렇지 않으면 RL상태



BinarySearchTree2.c의 실질적 변화

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    ...
    Rebalance(pRoot); // 노드 추가 후 리밸런싱!
}

BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    ...
    Rebalance(pRoot); // 노드 제거 후 리밸런싱!
    return dNode;
}
```

앞서 소개한 함수의 확장 결과(예측)

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    ...
    *pRoot = Rebalance(pRoot); // 노드 추가 후 리밸런싱!
}

BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    ...
    *pRoot = Rebalance(pRoot); // 노드 제거 후 리밸런싱!
    return dNode;
}
```

실질적인 변화 결과



AVL 트리의 동작결과 확인: main 함수

```
int main(void)
{
    BTTreeNode * avlRoot;
    BTTreeNode * clNode;           // current left node
    BTTreeNode * crNode;           // current right node
    BSTMakeAndInit(&avlRoot);

    BSTInsert(&avlRoot, 1);
    BSTInsert(&avlRoot, 2);
    BSTInsert(&avlRoot, 3);
    BSTInsert(&avlRoot, 4);
    BSTInsert(&avlRoot, 5);
    BSTInsert(&avlRoot, 6);
    BSTInsert(&avlRoot, 7);
    BSTInsert(&avlRoot, 8);
    BSTInsert(&avlRoot, 9);

    printf("루트 노드: %d \n", GetData(avlRoot));
}
```

BinaryTree3.h
BinaryTree3.c
BinarySearchTree3.h
BinarySearchTree3.c
AVLRebalance.h
AVLRebalance.c

AVLTreeMain.c 파일구성

루트 노드: 5
왼쪽1: 4, 오른쪽1: 6
왼쪽2: 3, 오른쪽2: 7
왼쪽3: 2, 오른쪽3: 8
왼쪽4: 1, 오른쪽4: 9

실행결과

```
clNode = GetLeftSubTree(avlRoot);
crNode = GetRightSubTree(avlRoot);
printf("왼쪽1: %d, 오른쪽1: %d \n", GetData(clNode), GetData(crNode));

clNode = GetLeftSubTree(clNode);
crNode = GetRightSubTree(crNode);
printf("왼쪽2: %d, 오른쪽2: %d \n", GetData(clNode), GetData(crNode));

clNode = GetLeftSubTree(clNode);
crNode = GetRightSubTree(crNode);
printf("왼쪽3: %d, 오른쪽3: %d \n", GetData(clNode), GetData(crNode));

clNode = GetLeftSubTree(clNode);
crNode = GetRightSubTree(crNode);
printf("왼쪽4: %d, 오른쪽4: %d \n", GetData(clNode), GetData(crNode));
return 0;
```



수고하셨습니다~



Chapter 12에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 13. 테이블과 해쉬

Introduction To Data Structures Using C

Chapter 13. 테이블과 해쉬



Chapter 13-1:

빠른 탐색을 보이는 해쉬 테이블



테이블 자료구조의 이해

key 역시 의미 있는 데이터로 정의하는 것이 좋다!

사번 : key	직원 : value
99001	양현석 부장
99002	한상현 차장
99003	이현진 과장
99004	이수진 사원

데이터가 key와 value로 한 쌍을 이루며, key가 데이터의 저장 및 탐색의 도구가 된다.
즉 테이블 자료구조에서는 원하는 데이터를 단번에 찾을 수 있다.

테이블 자료구조의 예

AVL 트리의 탐색 연산이 $O(\log_2 n)$ 의 시간 복잡도를 보이는 반면, 테이블 자료구조의 탐색 연산은 $O(1)$ 의 시간 복잡도를 보인다!

테이블은 사전 구조 또는 맵(map)이라고도 불린다.



배열을 기반으로 하는 테이블

```
typedef struct _empInfo
{
    int empNum;           // 직원의 고유번호
    int age;              // 직원의 나이
} EmpInfo;               value

int main(void)
{
    EmpInfo empInfoArr[1000];
    EmpInfo ei;
    int eNum;

    printf("사번과 나이 입력: ");
    scanf("%d %d", &(ei.empNum), &(ei.age));
    empInfoArr[ei.empNum] = ei;      // 단번에 저장!
                                    Key에 해당하는 직원의 고유번호를 배열의 인덱스
                                    값으로 활용하고 있다.

    printf("확인하고픈 직원의 사번 입력: ");
    scanf("%d", &eNum);

    ei = empInfoArr[eNum];        // 단번에 탐색!
    printf("사번 %d, 나이 %d \n", ei.empNum, ei.age);
    return 0;
}
```

이는 테이블의 개념적 이해를 돋는 예제이다.
단 해쉬의 개념이 빠져있기 때문에 효율적인
테이블이라 할 수는 없다.

실행결과

```
사번과 나이 입력: 129 29
확인하고픈 직원의 사번 입력: 129
사번 129, 나이 29
```



테이블에 의미를 부여하는 해쉬 함수와 충돌문제

```
int main(void)
{
    EmpInfo empInfoArr[100];

    EmpInfo emp1={20120003, 42};
    EmpInfo emp2={20130012, 33};
    EmpInfo emp3={20170049, 27};

    EmpInfo r1, r2, r3;

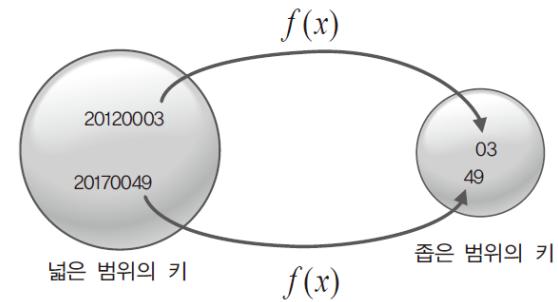
    // 키를 인덱스 값으로 이용해서 저장
    empInfoArr[GetHashCode(emp1.empNum)] = emp1;
    empInfoArr[GetHashCode(emp2.empNum)] = emp2;
    empInfoArr[GetHashCode(emp3.empNum)] = emp3;

    // 키를 인덱스 값으로 이용해서 탐색
    r1 = empInfoArr[GetHashCode(20120003)];
    r2 = empInfoArr[GetHashCode(20130012)];
    r3 = empInfoArr[GetHashCode(20170049)];

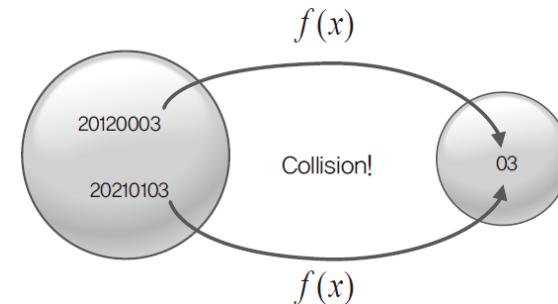
    // 탐색 결과 확인
    printf("사번 %d, 나이 %d \n", r1.empNum, r1.age);
    printf("사번 %d, 나이 %d \n", r2.empNum, r2.age);
    printf("사번 %d, 나이 %d \n", r3.empNum, r3.age);
    return 0;
}
```

```
int GetHashCode(int empNum)
{
    return empNum % 100;
}
```

해쉬 함수 $f(x)$



합리적인 메모리 공간의 할당을 돋는다.



데이터는 다른데 해쉬 값은 같은 충돌 발생 가능!



어느 정도 갖춰진 해쉬 테이블의 예: Person

```
typedef struct _person
{
    int ssn;           // 주민등록번호
    char name[STR_LEN]; // 이름
    char addr[STR_LEN]; // 주소
} Person;

int GetSSN(Person * p);
void ShowPerInfo(Person * p);
Person * MakePersonData(int ssn, char * name,
```

헤더파일

```
int GetSSN(Person * p)
{
    return p->ssn;
}
```

소스파일

```
void ShowPerInfo(Person * p)
{
    printf("주민등록번호: %d \n", p->ssn);
    printf("이름: %s \n", p->name);
    printf("주소: %s \n\n", p->addr);
}
```

```
Person * MakePersonData(int ssn, char * name, char * addr)
{
    Person * newP = (Person*)malloc(sizeof(Person));
    newP->ssn = ssn;
    strcpy(newP->name, name);
    strcpy(newP->addr, addr);
    return newP;
}
```

테이블의 저장 대상에 대한 정의!

- 키 : 주민등록 번호
- 값 : 구조체 변수의 주소 값

어느 정도 갖춰진 해쉬 테이블의 예: 슬롯

```
typedef int Key;           // 주민등록번호
typedef Person * Value;

enum SlotStatus {EMPTY, DELETED, INUSE};

typedef struct _slot
{
    Key key;
    Value val;
    enum SlotStatus status;
} Slot;
```

사번 : key	직원 : value
99001	양현석 부장
99002	한상현 차장
99003	이현진 과장
99004	이수진 사원

슬롯

슬롯의 상태

- EMPTY 이 슬롯에는 데이터가 저장된바 없다.
- DELETED 이 슬롯에는 데이터가 저장된바 있으나 현재는 비워진 상태다.
- INUSE 이 슬롯에는 현재 유효한 데이터가 저장되어 있다.

지금 당장은 EMPTY와 INUSE면 충분하다. 그러나 충돌 문제의 해결을 감안하여 DELETED를 슬롯의 상태에 포함시킨다.



해쉬 테이블의 헤더파일과 초기화 함수

```
typedef int HashFunc(Key k);

typedef struct _table
{
    Slot tbl[MAX_TBL];
    HashFunc * hf;
} Table;

// 테이블의 초기화
void TBLInit(Table * pt, HashFunc * f);

// 테이블에 키와 값을 저장
void TBLInsert(Table * pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table * pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table * pt, Key k);
```

```
void TBLInit(Table * pt, HashFunc * f)
{
    int i;

    // 모든 슬롯 초기화
    for(i=0; i<MAX_TBL; i++)
        (pt->tbl[i]).status = EMPTY;

    pt->hf = f;      // 해쉬 함수 등록
}
```

해쉬 함수는 등록 또는 변경이 가능하도록 정의하는 것이 좋다!

그리고 일반적으로 삽입, 삭제 및 탐색의 과정에서 키를 별도로 전달하도록 함수가 정의된다.



헤더파일에 선언된 함수들의 정의

```
void TBLInsert(Table * pt, Key k, Value v)
{
    int hv = pt->hf(k); 해시 값을 얻는다!
    pt->tbl[hv].val = v;
    pt->tbl[hv].key = k;
    pt->tbl[hv].status = INUSE;
}
```

```
Value TBLDelete(Table * pt, Key k)
{
    int hv = pt->hf(k); 해시 값을 얻는다!

    if((pt->tbl[hv]).status != INUSE)
    {
        return NULL;
    }
    else
    {
        (pt->tbl[hv]).status = DELETED;
        return (pt->tbl[hv]).val;
    }           삭제되는 데이터 반환
}
```

```
Value TBLSearch(Table * pt, Key k)
{
    int hv = pt->hf(k); 해시 값을 얻는다!

    if((pt->tbl[hv]).status != INUSE)
        return NULL;
    else
        return (pt->tbl[hv]).val;
}           탐색 대상 반환
```



해쉬 함수의 정의와 main 함수

```
int MyHashFunc(int k)
{
    return k % 100;    매우 단순한 해쉬 함수의 정의
}
```

```
int main(void)
{
    Table myTbl;                                Person.h, Person.c,
    Person * np;                                 Slot.h, Table.h, Table.c,
    Person * sp;                                 SimpleHashMain.c
    Person * rp;                                 실행 위한 파일의 구성

    TBLInit(&myTbl, MyHashFunc);

    // 데이터 입력
    np = MakePersonData(20120003, "Lee", "Seoul");
    TBLInsert(&myTbl, GetSSN(np), np);

    np = MakePersonData(20130012, "KIM", "Jeju");
    TBLInsert(&myTbl, GetSSN(np), np);

    np = MakePersonData(20170049, "HAN", "Kangwon");
    TBLInsert(&myTbl, GetSSN(np), np);
```

```
// 데이터 탐색
sp = TBLSearch(&myTbl, 20120003);
if(sp != NULL)
    ShowPerInfo(sp);

sp = TBLSearch(&myTbl, 20130012);
if(sp != NULL)
    ShowPerInfo(sp);

sp = TBLSearch(&myTbl, 20170049);
if(sp != NULL)
    ShowPerInfo(sp);

// 데이터 삭제
rp = TBLDelete(&myTbl, 20120003);
if(rp != NULL)
    free(rp);

rp = TBLDelete(&myTbl, 20130012);
if(rp != NULL)
    free(rp);

rp = TBLDelete(&myTbl, 20170049);
if(rp != NULL)
    free(rp);

return 0;
}
```



좋은 해쉬 함수의 조건



데이터의 저장 위치가 적당히 분산되어 있다.

- ▶ [그림 13-4: 좋은 해쉬 함수를 사용한 결과]



데이터가 특정 위치에 몰려 있다.

- ▶ [그림 13-5: 좋지 않은 해쉬 함수를 사용한 결과]

"좋은 해쉬 함수는 키의 일부분을 참조하여 해쉬 값을 만들지 않고, 키 전체를 참조하여 해쉬 값을 만들어 낸다."

이는 많은 수의 데이터를 조합하여(키 전부를 조합하여) 해쉬 값 생성시 다양한 값의 생성을 기대할 수 있을 것이라는 단순한 생각을 근거로 한다.



자릿수 선택 방법과 자릿수 폴딩 방법

자릿수 선택 방법

“여덟 자리의 수로 이루어진 키에서 다양한 해수 값 생성에 도움을 주는 네 자리의 수를 뽑아서 해수 값을 생성한다.”

키의 특정 위치에서 중복의 비율이 높거나 아예 공통으로 들어가는 값이 있다면 이들을 제외시키는 방법

자릿수 폴딩 방법

2 7 3 4 1 9

$$27 + 34 + 19$$

폴딩의 방법에 있어서 꼭 지켜야 하는 정해진 규칙이 있는 것은 아니다! 여러 가지 사고를 근거로 다양한 적용이 가능하다.



Chapter 13. 테이블과 해쉬



Chapter 13-2:

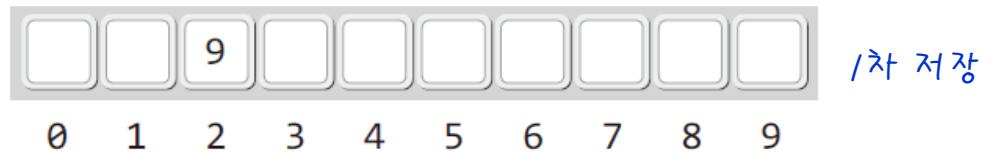
충돌 문제의 해결책



선형 조사법(Linear Probing)

- 해쉬 함수

key % 7



- 해쉬 함수

key % 7



$$f(k)+1 \rightarrow f(k)+2 \rightarrow f(k)+3 \rightarrow f(k)+4 \dots$$

선형 조사법에서의 빈자리 찾는 과정

선형 조사법은 단순하지만, 충돌의 횟수가 증가함에 따라서 클러스터 현상(특정 영역에 데이터가 몰리는 현상)이 발생한다는 단점이 있다.



이차 조사법과 슬롯의 상태 DELETED

$$f(k)+1^2 \rightarrow f(k)+2^2 \rightarrow f(k)+3^2 \rightarrow f(k)+4^2 \dots$$

이차 조사법에서의 빈자리 찾는 과정, 선형 조사법보다 멀리서 빈자리를 찾는다.



1차, 저장



2차, 저장(충돌 발생 & 충돌 해결)



3차, 9의 삭제

이렇듯 DELETED 상태로 별도 표시 해 두어야 동일한 해쉬 값의 데이터 저장을 의심할 수 있다.



이중 해쉬: 이해

해쉬 값이 같으면, 충돌 발생시 빈 슬롯을 찾기 위한 접근 위치가 늘 동일하다는 문제점을 해결한 방법으로 총 두 개의 해쉬 함수를 활용하는 방법이다.

- 1차 해쉬 함수 $h1(k) = k \% 15$ 배열의 길이가 15인 경우의 예

- 2차 해쉬 함수 $h2(k) = 1 + (k \% c)$ 15보다 작은 소수로 c 를 결정한다.



C의 결정 예

- 1차 해쉬 함수 $h1(k) = k \% 15$

- 2차 해쉬 함수 $h2(k) = 1 + (k \% 7)$

- 1을 더하는 이유: 2차 해쉬 값이 0이 되는 것을 막기 위해서
- c 를 15보다 작은 값으로 하는 이유: 배열의 길이가 15이므로
- c 를 소수로 결정하는 이유: 클러스터 현상을 낮춘다는 통계를 근거로!



이중 해쉬: 적용

- 1차 해쉬 함수 $h1(k) = k \% 15$
- 2차 해쉬 함수 $h2(k) = 1 + (k \% 7)$

• $h1(3) = 3 \% 15 = 3$ /차, 저장

• $h1(18) = 18 \% 15 = 3$

2차, 충돌

• $h1(33) = 33 \% 15 = 3$

3차, 충돌

실제로는 2차 해쉬 값을 근거로 빈 자리를 찾기 때문에 1차 해쉬 값이 같아도 빈 자리를 찾는 위치는 달라지게 된다.

→ • $h2(18) = 1 + 18 \% 7 = 5$

18에 대한 2차 해쉬 값

→ • $h2(33) = 1 + 33 \% 7 = 6$

33에 대한 2차 해쉬 값

• $h2(18) \rightarrow h2(18) + 5 \times 1 \rightarrow h2(18) + 5 \times 2 \rightarrow h2(18) + 5 \times 3 \dots$

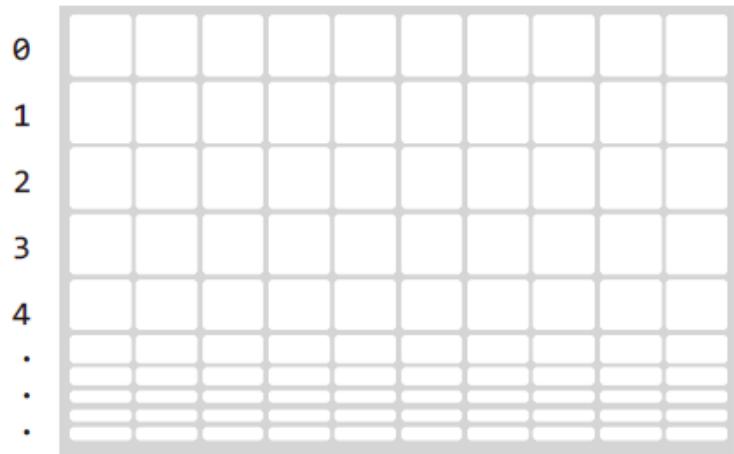
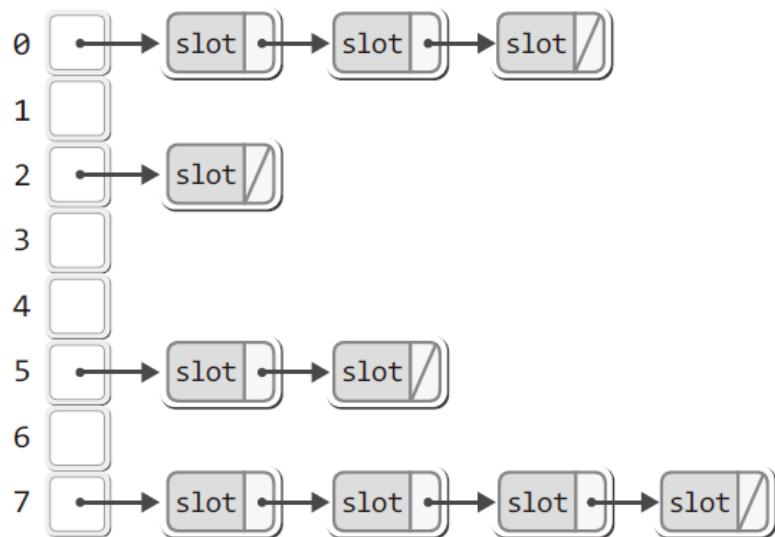
• $h2(33) \rightarrow h2(33) + 6 \times 1 \rightarrow h2(33) + 6 \times 2 \rightarrow h2(33) + 6 \times 3 \dots$

2차 해쉬 값을 근거로 빈자리 찾기!



체이닝(닫힌 어드레싱 모델)

한 해쉬 값에 다수의 데이터를 저장할 수 있도록 배열을
2차원의 형태로 선언하는 모델!



한 해쉬 값에 다수의 데이터를 저장할 수 있도록 각
해쉬 값 별로 연결 리스트를 구성하는 모델



체이닝의 구현: 구현의 방법

앞서 구현한 테이블을 변경 및 확장하는 형태로 구현하기로 결정!

- Slot.h → 변경 및 확장 후 Slot2.h로 이름 변경
 - Table.h → 변경 및 확장 후 Table2.h로 이름 변경
 - Table.c → 변경 및 확장 후 Table2.c로 이름 변경

- DLinkedList.h, DLinkedList.c 연결 리스트의 구현 결과

해수 값 별 연결 리스트 구성을 위해 Ch 04에서 구현한 연결 리스트를 활용!



체이닝의 구현: 슬롯의 변경

이전 구현: *Slot.h*

```
typedef int Key;           // 주민등록번호
typedef Person * Value;

enum SlotStatus {EMPTY, DELETED, INUSE};

typedef struct _slot
{
    Key key;
    Value val;
    enum SlotStatus status;
} Slot;
```

체이닝 기반에서는 슬롯의 상태 정보를
유지하지 않아도 된다.

체이닝 기반 구현: *Slot2.h*

```
typedef int Key;
typedef Person * Value;

typedef struct _slot
{
    Key key;
    Value val;
} Slot;
```



체이닝의 구현: 테이블 구조체의 변경

이전 구현: Table.h

```
typedef int HashFunc(Key k);

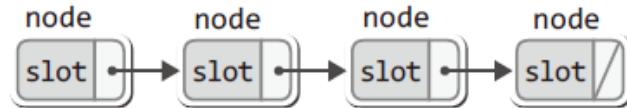
typedef struct _table
{
    Slot tbl[MAX_TBL];
    HashFunc * hf;           바뀐 부분!
} Table;

// 테이블의 초기화
void TBLInit(Table * pt, HashFunc * f);

// 테이블에 키와 값을 저장
void TBLInsert(Table * pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table * pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table * pt, Key k);
```



노드의 데이터 부분이 슬롯이 되게 한다!

연결 리스트를 그대로 활용하는 좋은 모델

체이닝 기반 구현: Table2.h

```
typedef int HashFunc(Key k);

typedef struct _table
{
    List tbl[MAX_TBL];
    HashFunc * hf;           해쉬 값 별로
} Table;                  연결 리스트를 구성해야 한다!

void TBLInit(Table * pt, HashFunc * f);
void TBLInsert(Table * pt, Key k, Value v);
Value TBLDelete(Table * pt, Key k);
Value TBLSearch(Table * pt, Key k);
```



체이닝의 구현: 연결 리스트의 선언 변경

```
#ifndef __D_LINKED_LIST_H__
#define __D_LINKED_LIST_H__

#include "Slot2.h"      // 추가된 헤더파일 선언문
. . . . 중간 생략 . . . .

typedef Slot LData;    // 변경된 typedef 선언문

typedef struct _node
{
    LData data;
    struct _node * next;
} Node;

typedef struct _linkedList
{
    Node * head;
    Node * cur;
    Node * before;
    int numOfData;
    int (*comp)(LData d1, LData d2);
} LinkedList;

. . . . 이하 생략 . . . .
```

데이터가 슬롯이니 LData를 Slot으로 typedef 선언한다!



체이닝의 구현: 초기화와 삽입 연산

```
void TBLInit(Table * pt, HashFunc * f)
{
    int i;

    for(i=0; i<MAX_TBL; i++)
        ListInit(&(pt->tbl[i]));

    pt->hf = f;
}
```

연결 리스트 각각에 대해서 초기화를 진행

```
void TBLInsert(Table * pt, Key k, Value v)
{
    int hv = pt->hf(k);
    Slot ns = {k, v};

    if(TBLSearch(pt, k) != NULL)      // 키가 중복되었다면
    {
        printf("키 중복 오류 발생 \n");
        return;
    }
    else
    {
        LInsert(&(pt->tbl[_]), ns);
    }
}
```

해ش 값 기반 삽입!

테이블에 저장되는 데이터의 키 값은 유일해야 한다!

따라서 중복 여부를 확인하고 삽입을 진행한다.



체이닝의 구현: 삭제와 탐색

```
Value TBLDelete(Table * pt, Key k)
{
    int hv = pt->hf(k);
    Slot cSlot;

    if(LFirst(&(pt->tbl[hv]), &cSlot))
    {
        if(cSlot.key == k)
        {
            LRemove(&(pt->tbl[hv]));
            return cSlot.val;
        }
        else
        {
            while(LNext(&(pt->tbl[hv]), &cSlot))
            {
                if(cSlot.key == k)
                {
                    LRemove(&(pt->tbl[hv]));
                    return cSlot.val;
                }
            }
        }
    }
    return NULL;
}
```

삽입과 탐색이 해시 값을 기반으로 진행되기 때문에
코드의 구성이 유사하다!

```
Value TBLSearch(Table * pt, Key k)
{
    int hv = pt->hf(k);
    Slot cSlot;

    if(LFirst(&(pt->tbl[hv]), &cSlot))
    {
        if(cSlot.key == k)
        {
            return cSlot.val;
        }
        else
        {
            while(LNext(&(pt->tbl[hv]), &cSlot))
            {
                if(cSlot.key == k)
                    return cSlot.val;
            }
        }
    }
    return NULL;
}
```



실행 프로그램의 구성과 읽을거리

실행을 위한 파일의 구성

- Person.h, Person.c
- Slot2.h
- Table2.h, Table2.c
- DLinkedList.h, DLinkedList.c
- ChainedTableMain.c

Page 288의 우리가 구현한 테이블과 관련해서 반성할 점이라는 주제의 내용은 개인적으로
읽어 보시길 권해 드립니다.. ^ ^



수고하셨습니다~



Chapter 13에 대한 강의를 마칩니다!



윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 14. 그래프

Introduction To Data Structures Using C

Chapter 14. 그래프



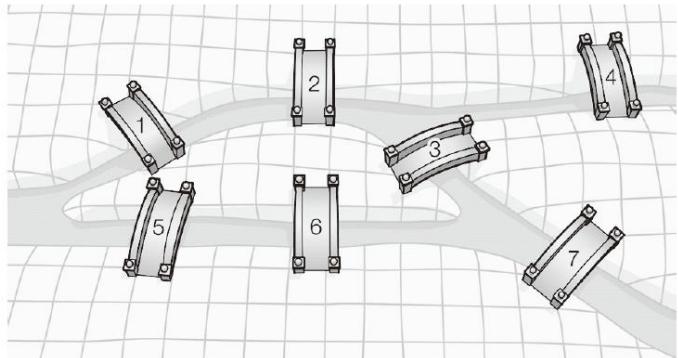
Chapter 14-1:

그래프의 이해와 종류

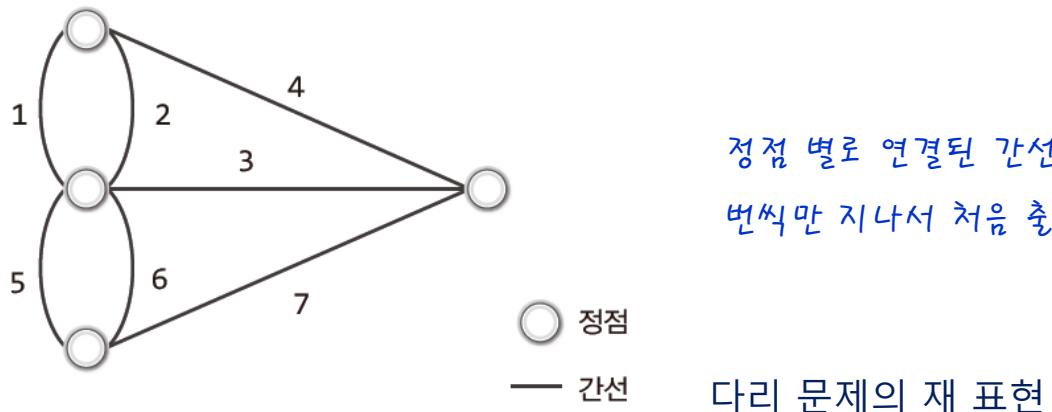


그래프의 역사와 이야기거리

모든 다리를 한 번씩만 건너서 처음 출발했던 장소로 돌아올 수 있는가?



쾨니히스베르크의 다리 문제

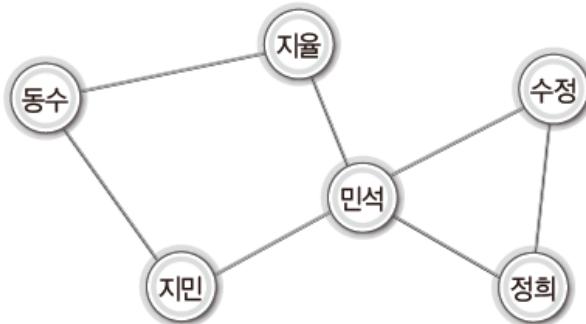


정점 별로 연결된 간선의 수가 모두 짝수 이어야 간선을 한번씩만 지나서 처음 출발했던 정점으로 돌아올 수 있다.

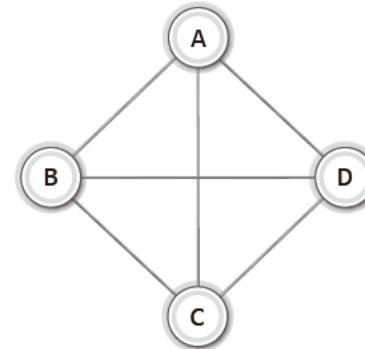


그래프의 이해와 종류

5학년 3반 어린이들의 비상 연락망: 연락의 방향성이 없다.

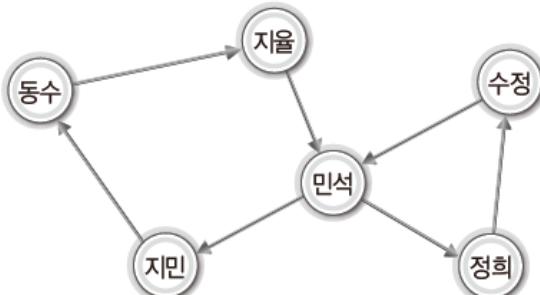


무방향 그래프의 예

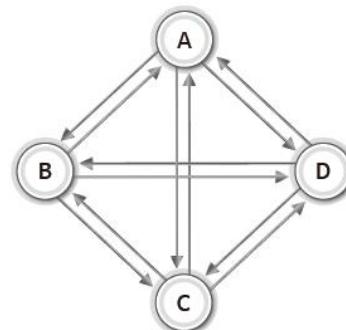


무방향 완전 그래프의 예

5학년 3반 어린이들의 비상 연락망: 방향성이 있다.



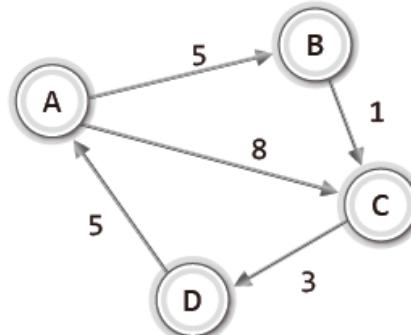
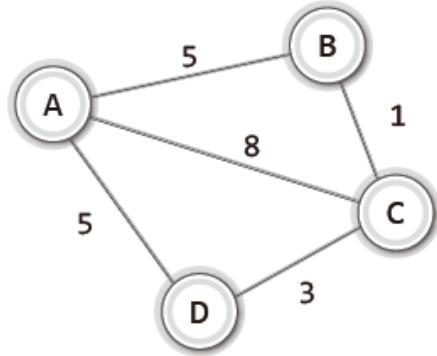
방향 그래프의 예



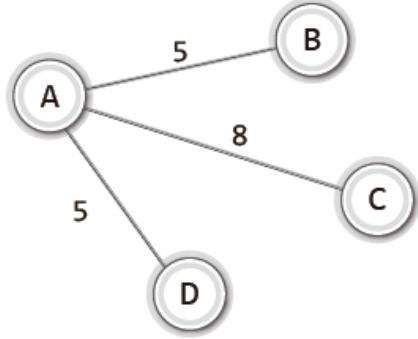
방향 완전 그래프의 예



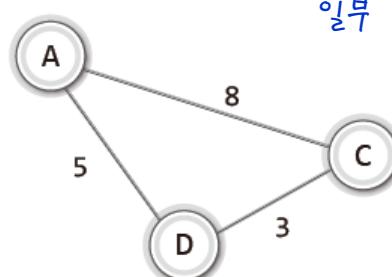
가중치 그래프와 부분 그래프



▼ 부분 그래프



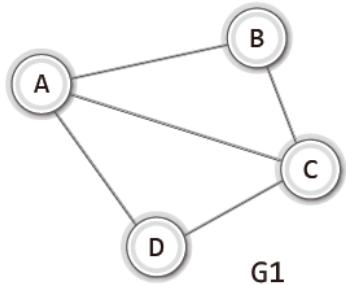
▶ 부분 그래프



일부 정점과 간선으로 구성이 된 그래프



그래프의 집합 표현



$$V(G_1) = \{A, B, C, D\}$$

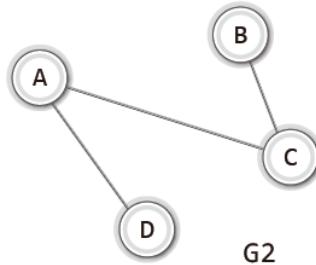
$$E(G_1) = \{(A, B), (A, C), (A, D), (B, C), (C, D)\}$$

- 그래프 G의 정점 집합

$V(G)$ 로 표시함

- 그래프 G의 간선 집합

$E(G)$ 로 표시함

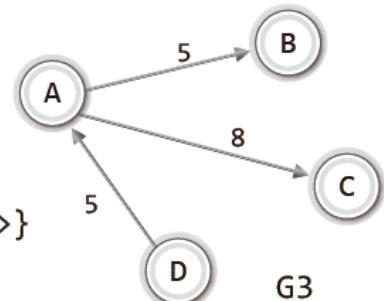


$$V(G_2) = \{A, B, C, D\}$$

$$E(G_2) = \{(A, C), (A, D), (B, C)\}$$

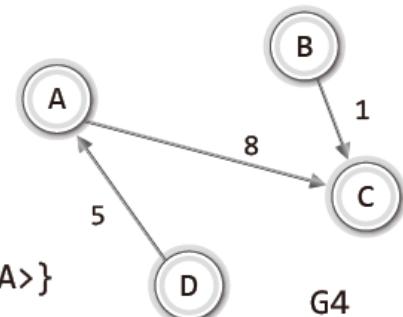
$$V(G_3) = \{A, B, C, D\}$$

$$E(G_3) = \{(A, B), (A, C), (D, A)\}$$



$$V(G_4) = \{A, B, C, D\}$$

$$E(G_4) = \{(A, C), (B, C), (D, A)\}$$



그래프의 ADT

- void GraphInit(UALGraph * pg, int nv);
 - 그래프의 초기화를 진행한다.
 - 두 번째 인자로 정점의 수를 전달한다.

```
enum {A, B, C, D, E, F, G, H, I, J};
```

```
enum {SEOUL, INCHEON, DAEGU, BUSAN, KWANGJU};
```

- void GraphDestroy(UALGraph * pg);
 - 그래프 초기화 과정에서 할당한 리소스를 반환한다.

정점의 이름을 선언하는 방법

- void AddEdge(UALGraph * pg, int fromV, int toV);
 - 매개변수 fromV와 toV로 전달된 정점을 연결하는 간선을 그래프에 추가한다.

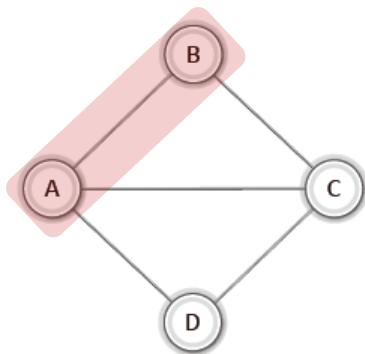
- void ShowGraphEdgeInfo(UALGraph * pg);
 - 그래프의 간선정보를 출력한다.

모든 기능과 가능성을 담아서 ADT를 정의하는 것이 능사는 아니다!

특정 그래프를 대상으로 ADT를 제한하여 정의하는 것이 오히려 현명할 수 있다!

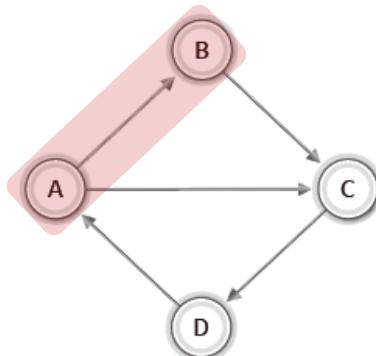


그래프를 구현하는 두 가지 방법: 인접 행렬 기반



	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0

정방 행렬을 이용하는 '인접 행렬 기반 그래프'의 예 1

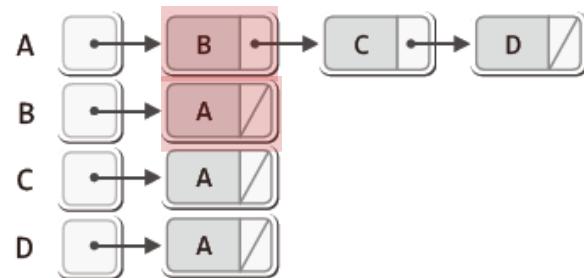
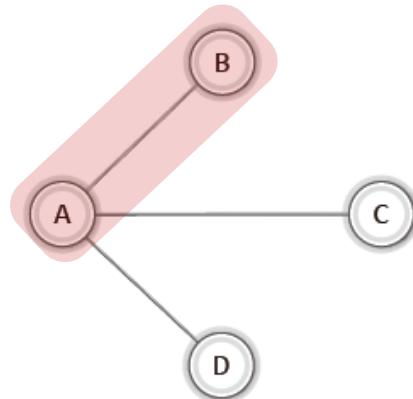


	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	0	0	1
D	1	0	0	0

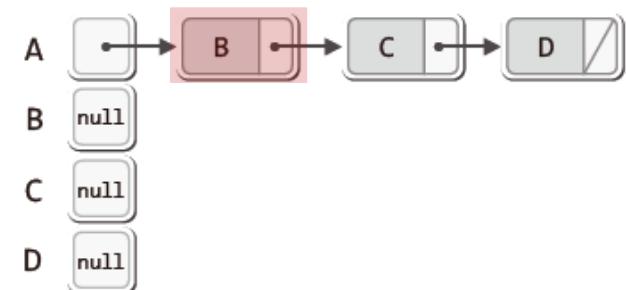
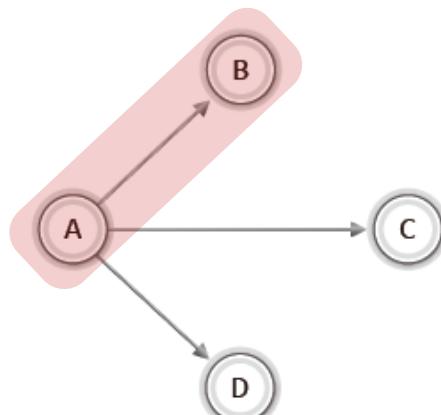
정방 행렬을 이용하는 '인접 행렬 기반 그래프'의 예 2



그래프를 구현하는 두 가지 방법: 인접 리스트 기반



연결 리스트를 이용하는 '인접 리스트 기반 그래프'의 예 1



연결 리스트를 이용하는 '인접 리스트 기반 그래프'의 예 2



Chapter 14. 그래프



Chapter 14-2:

인접 리스트 기반의 그래프 구현



그래프의 헤더파일 정의

```
// 연결 리스트를 가져다 쓴다!
```

```
#include "DLinkedList.h"
```

앞서 구현한 연결 리스트를 그대로 활용하여 구현하기 위한 선언!

```
// 정점의 이름을 상수화
```

```
enum {A, B, C, D, E, F, G, H, I, J};
```

```
typedef struct _ual
```

정점의 이름을 선언하는 방법!

```
{
```

```
    int numV;           // 정점의 수
```

```
    int numE;           // 간선의 수
```

```
    List * adjList;     // 간선의 정보
```

```
} ALGraph;
```

```
// 그래프의 초기화
```

```
void GraphInit(ALGraph * pg, int nv);
```

```
// 그래프의 리소스 해제
```

```
void GraphDestroy(ALGraph * pg);
```

```
// 간선의 추가
```

```
void AddEdge(ALGraph * pg, int fromV, int toV);
```

```
// 간선의 정보 출력
```

```
void ShowGraphEdgeInfo(ALGraph * pg);
```



선언된 함수의 이해를 돋기 위한 main 함수

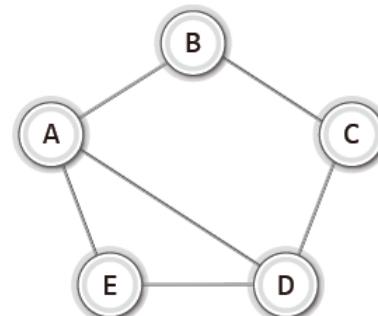
```
int main(void)
{
    ALGraph graph;           // 그래프의 생성
    GraphInit(&graph, 5);    // 그래프의 초기화
    초기화 과정에서 정점의 수를 결정한다.
    AddEdge(&graph, A, B);   // 정점 A와 B를 연결
    AddEdge(&graph, A, D);   // 정점 A와 D를 연결
    AddEdge(&graph, B, C);   // 정점 B와 C를 연결
    AddEdge(&graph, C, D);   // 정점 C와 D를 연결
    AddEdge(&graph, D, E);   // 정점 D와 E를 연결
    AddEdge(&graph, E, A);   // 정점 E와 A를 연결

    ShowGraphEdgeInfo(&graph); // 그래프의 간선정보 출력
    GraphDestroy(&graph);     // 그래프의 리소스 소멸
    return 0;
}
```

A와 연결된 정점: B D E
B와 연결된 정점: A C
C와 연결된 정점: B D
D와 연결된 정점: A C E
E와 연결된 정점: A D

실행결과

ALGraph.h
ALGraph.c
ALGraphMain.c
DLinkedList.h
DLinkedList.c **파일구성**



main 함수를 통해서 생성한 그래프



그래프의 구현: 초기화와 소멸

```

void GraphInit(ALGraph * pg, int nv)           // 그래프의 초기화
{
    int i;

    // 정점의 수에 해당하는 길이의 리스트 배열을 생성한다.
    pg->adjList = (List*)malloc(sizeof(List)*nv);    // 간선정보를 저장할 리스트 생성

    pg->numV = nv;          // 정점의 수는 nv에 저장된 값으로 결정
    pg->numE = 0;           // 초기의 간선 수는 0개

    // 정점의 수만큼 생성된 리스트들을 초기화한다.
    for(i=0; i<nv; i++)
    {
        ListInit(&(pg->adjList[i]));
        SetSortRule(&(pg->adjList[i]), WhoIsPrecede);
    }
}

int WhoIsPrecede(int data1, int data2)
{
    if(data1 < data2)
        return 0;
    else
        return 1;
}

그래프와 연관성 없다! 다만 연결 리스트가 요구하므로 적당한 함수를 등록하였다.

void GraphDestroy(ALGraph * pg)           // 그래프 리소스의 해제
{
    if(pg->adjList != NULL)
        free(pg->adjList);               // 동적으로 할당된 연결 리스트의 소멸
}

```



그래프의 구현: 간선의 추가와 간선 정보 출력

```
// 간선의 추가
void AddEdge(ALGraph * pg, int fromV, int toV)
{
    LIInsert(&(pg->adjList[fromV]), toV);
    LIInsert(&(pg->adjList[toV]), fromV);

    pg->numE += 1;
}
```

무방향 그래프의 구현을 보여준다.

방향 그래프의 구현이라면 LIInsert의 함수 호출이 1회로 끝이 난다.

```
// 간선의 정보 출력
void ShowGraphEdgeInfo(ALGraph * pg)
{
    int i;
    int vx;

    for(i=0; i<pg->numV; i++)
    {
        printf("%c와 연결된 정점: ", i + 65);

        if(LFirst(&(pg->adjList[i]), &vx))
        {
            printf("%c ", vx + 65);
            while(LNext(&(pg->adjList[i]), &vx))
                printf("%c ", vx + 65);
        }
        printf("\n");
    }
}
```



Chapter 14. 그래프

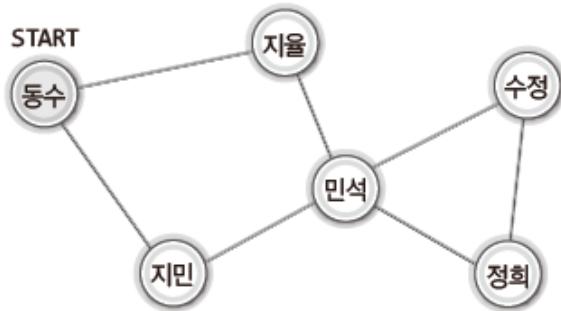


Chapter 14-3:

그래프의 탐색

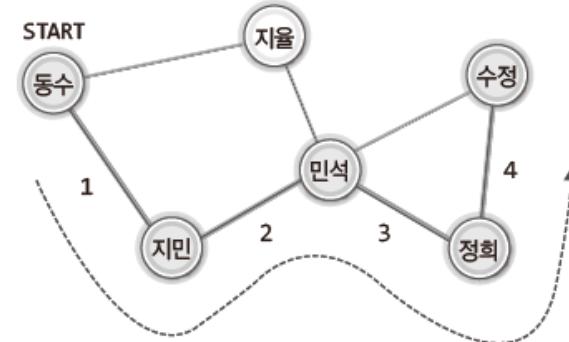


깊이 우선 탐색: Depth First Search



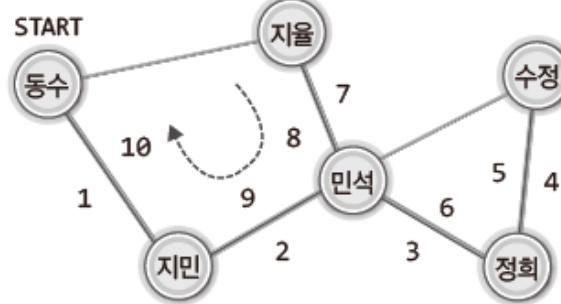
▶ [그림 14-16: DFS의 과정 1/4]

동수로부터 비상 연락망 가동!



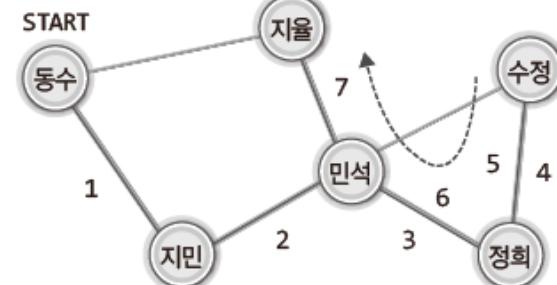
한 사람에게만
연락을 취해 나간다!

▶ [그림 14-17: DFS의 과정 2/4]



▶ [그림 14-20: DFS의 과정 4/4]

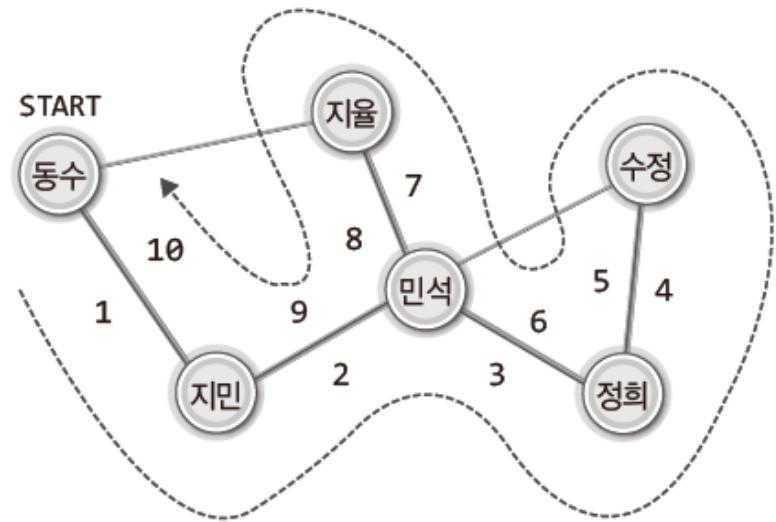
시작 점으로 되돌아 오면 연락 끝!



▶ [그림 14-18: DFS의 과정 3/4]

연락 할 곳이 없으면 역으로 되돌아 가면서
연락 취할 곳을 찾는다.

깊이 우선 탐색: 정리

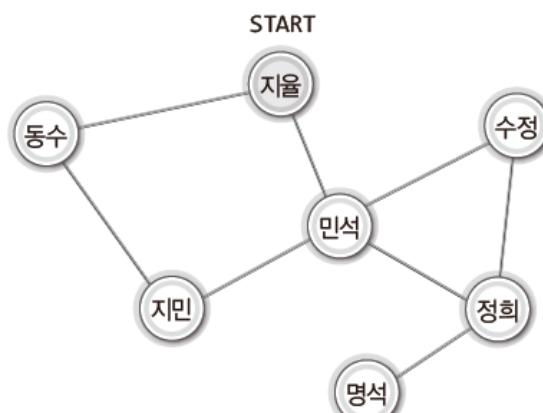


깊이 우선 탐색 과정의 핵심 세 가지

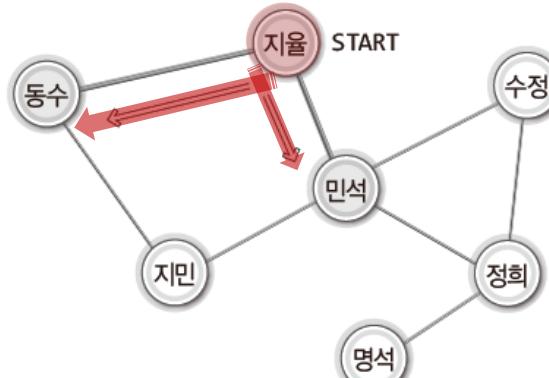
- 한 사람에게만 연락을 한다.
- 연락할 사람이 없으면, 자신에게 연락한 사람에게 이를 알린다.
- 처음 연락을 시작한 사람의 위치에서 연락은 끝이 난다



너비 우선 탐색: Breadth First Search

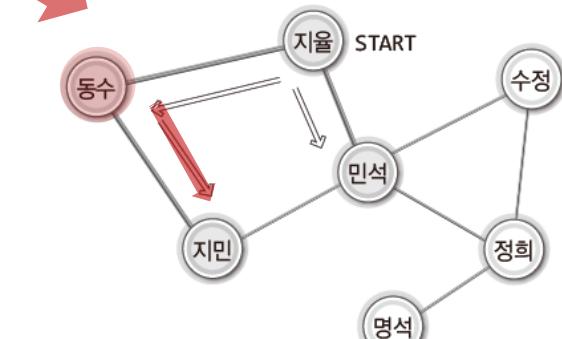


▶ [그림 14-22: BFS의 과정 1/5]

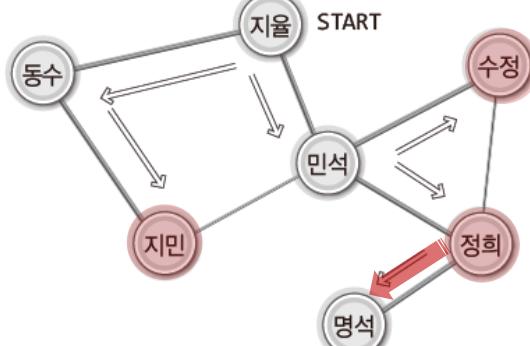


▶ [그림 14-23: BFS의 과정 2/5]

연결 된 모든 이에게 연락을!

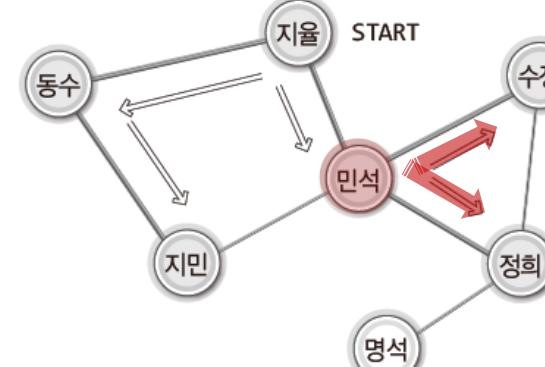


▶ [그림 14-24: BFS의 과정 3/5]



▶ [그림 14-26: BFS의 과정 5/5]

마지막으로 명석 또한 전달의 기회를 갖는다.



▶ [그림 14-25: BFS의 과정 4/5]



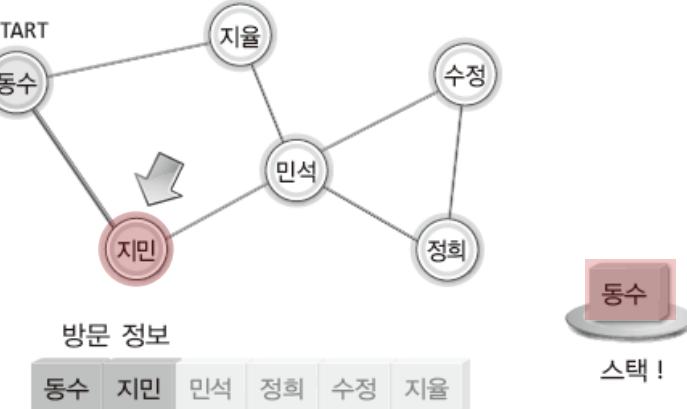
깊이 우선 탐색의 구현 모델: 과정 1~



경로 정보의 추적을 목적으로!
스택!

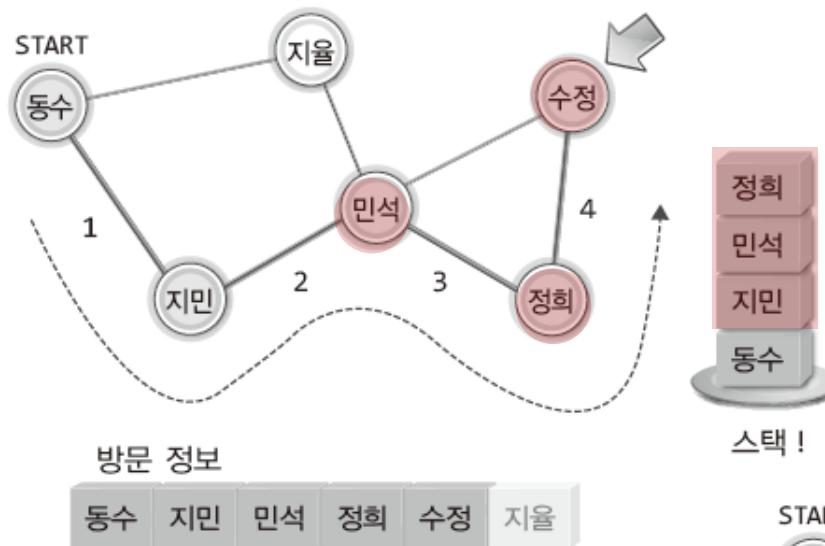
▶ [그림 14-28: DFS의 구현 1/7]

 동수를 떠나 지민에게 연락이 취해질 때
동수의 정보가 스택으로 이동한다!

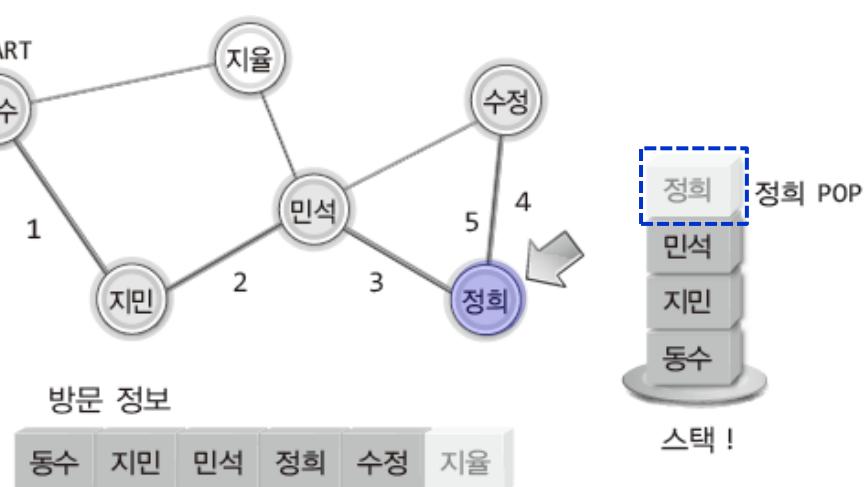


▶ [그림 14-29: DFS의 구현 2/7]

깊이 우선 탐색의 구현 모델: 과정 3~

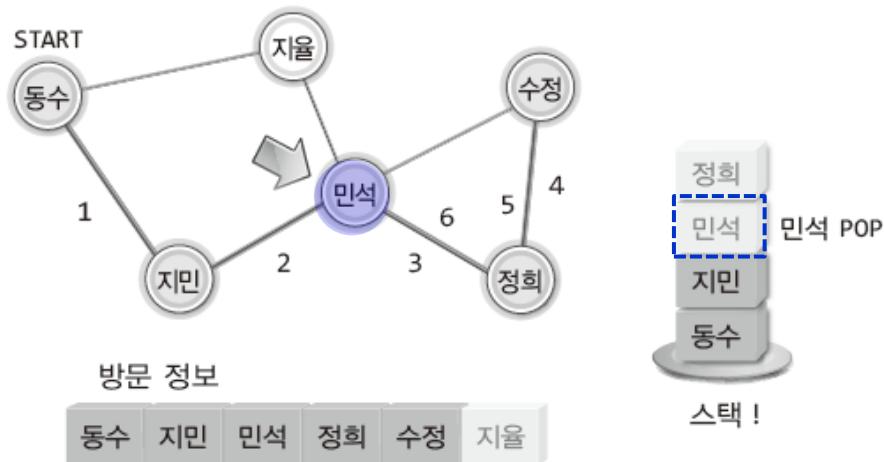


▶ [그림 14-30: DFS의 구현 3/7]



▶ [그림 14-31: DFS의 구현 4/7]

깊이 우선 탐색의 구현 모델: 과정 5~

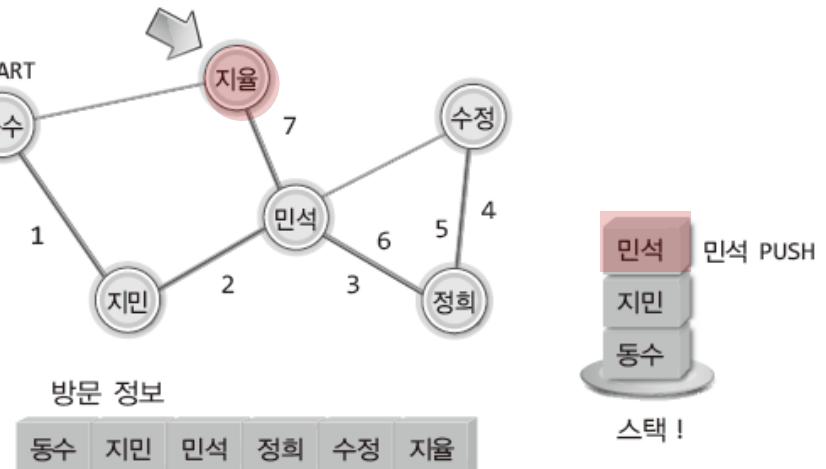


▶ [그림 14-32: DFS의 구현 5/7]



민석은 이전에 방문이 이뤄졌지만, 이와 상관 없이

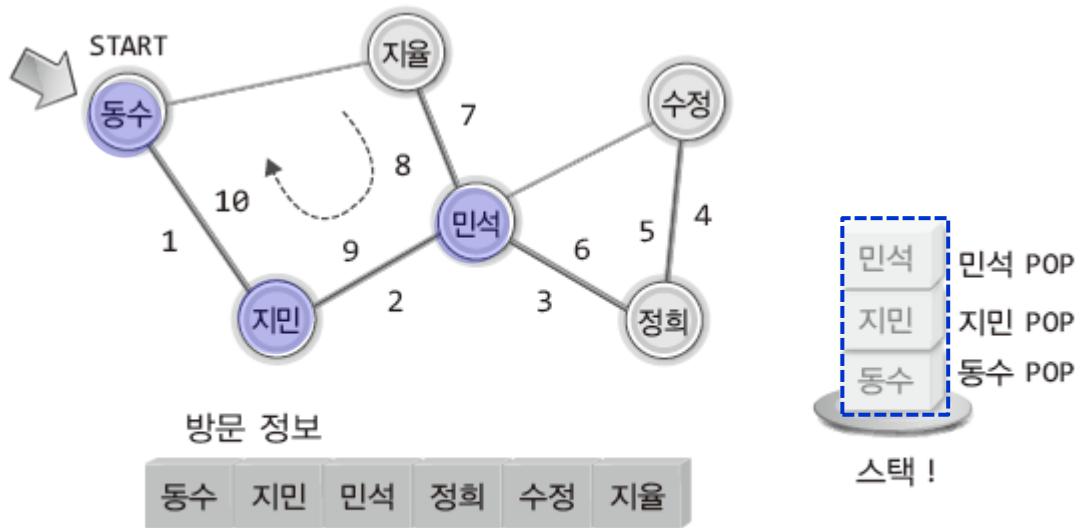
민석을 떠날 때 민석의 정보는 스택에 저장된다.



▶ [그림 14-33: DFS의 구현 6/7]



깊이 우선 탐색의 구현 모델: 과정 7



▶ [그림 14-34: DFS의 구현 7/7]

스택에 저장된 정보를 마지막까지 꺼내어 역으로 그 경로를 추적하다 보면 시작 위치로 이동이 가능하다!

깊이 우선 탐색의 실제 구현: 파일의 구성



깊이 우선 탐색의 실제 구현

void DFSShowGraphVertex(ALGraph * pg, int startV);

- 그래프의 모든 정점 정보를 출력하는 함수
- DFS를 기반으로 정의가 된 함수

구현 결과를 반영한 파일의 구성

- | | |
|--------------------------------------|----------------------------|
| · ALGraphDFS.h, ALGraphDFS.c | 그래프 관련 |
| · ArrayBaseStack.h, ArrayBaseStack.c | 스택 관련(Chapter 06에서 구현) |
| · DLinkedList.h, DLinkedList.c | 연결 리스트 관련(Chapter 04에서 구현) |
| · DFSMain.c | |



깊이 우선 탐색의 실제 구현: ALGraphDFS.h

```
// 정점의 이름들을 상수화  
enum {A, B, C, D, E, F, G, H, I, J};
```

정점의 이름을 결정하는 방법

```
typedef struct _ual  
{  
    int numV;          // 정점의 수  
    int numE;          // 간선의 수  
    List * adjList;    // 간선의 정보  
    int * visitInfo;   // 탐색과정에서 탐색이 진행된  
                       // 정점 정보를 담기 위한 멤버 추가!  
} ALGraph;  
  
// 그래프의 초기화  
void GraphInit(ALGraph * pg, int nv);  
  
// 그래프의 리소스 해제  
void GraphDestroy(ALGraph * pg);  
  
// 간선의 추가  
void AddEdge(ALGraph * pg, int fromV, int toV);  
  
// 간선의 정보 출력  
void ShowGraphEdgeInfo(ALGraph * pg);  
  
// 정점의 정보 출력: Depth First Search 기반  
void DFShowGraphVertex(ALGraph * pg, int startV);
```

멤버 visitInfo 관련 추가 코드

```
void GraphInit(ALGraph * pg, int nv)  
{  
    ....  
    // 정점의 수를 길이로 하여 배열을 할당  
    pg->visitInfo = (int *)malloc(sizeof(int) * pg->numV);  
  
    // 배열의 모든 요소를 0으로 초기화!  
    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);  
}
```

멤버 visitInfo 관련 추가 코드

```
void GraphDestroy(ALGraph * pg)  
{  
    ....  
    // 할당된 배열의 소멸!  
    if(pg->visitInfo != NULL)  
        free(pg->visitInfo);  
}
```



깊이 우선 탐색의 실제 구현: Helper Func

방문한 정점의 정보를 기록 및 출력

```
int VisitVertex(ALGraph * pg, int visitV)
{
    if(pg->visitInfo[visitV] == 0)          // visitV에 처음 방문일 때 '참'인 if문
    {
        pg->visitInfo[visitV] = 1;          // visitV에 방문한 것으로 기록
        printf("%c ", visitV + 65);        // 방문한 정점의 이름을 출력
        return TRUE;                      // 방문 성공!
    }
    return FALSE;                         // 방문 실패!
}
이미 방문한 정점이라면 FALSE가 반환된다!
```

DFShowGraphVertex 함수의 구현에 필요한, DFShowGraphVertex 함수 내에서 호출이 되는 함수로 써 방문한 정점의 정보를 그래프의 멤버 visitInfo가 가리키는 배열에 등록하는 기능을 제공한다.



깊이 우선 탐색의 실제 구현: DFShow~ 함수의 정의

```
void DFShowGraphVertex(ALGraph * pg, int startV)
```

```
{
```

.... 초기화 영역

```
while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
```

{ 연결된 정점의 정보를 얻어서!

```
int visitFlag = FALSE;
```

```
if(VisitVertex(pg, nextV) == TRUE) {
```

.... 방문을 시도했는데 방문에 성공하면

```
} else {
```

.... 방문을 시도했는데 방문한적 있는 곳이라면

```
}
```

```
if(visitFlag == FALSE) { 연결된 정점과의 방문이 모두 완료되었다면,
```

```
if(SIsEmpty(&stack) == TRUE)
```

break; 스택이 비면! 종료!

```
else
```

```
visitV = SPop(&stack);
```

되돌아 가기 위한 POP 연산!

```
}
```

.... 마무리 영역

```
}
```

Stack stack;

int visitV = startV;

int nextV;

StackInit(&stack);

VisitVertex(pg, visitV); 시작 정점 방문!

SPush(&stack, visitV);

시작 정점 떠나면서

스택으로!

memset(
pg->visitInfo, 0, sizeof(int) * pg->numV);

깊이 우선 탐색의 실제 구현: DFShow~ 함수의 정의

```
void DFShowGraphVertex(ALGraph * pg, int startV)
{
    .... 초기화 영역 ....
    while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        int visitFlag = FALSE;
        if(VisitVertex(pg, nextV) == TRUE) {
            .... 방문을 시도했는데 방문에 성공하면
        } else {
            .... 방문을 시도했는데 방문한적 있는 곳이라면
        }

        if(visitFlag == FALSE) {
            if(SIsEmpty(&stack) == TRUE)
                break;
            else
                visitV = SPop(&stack);
        }
    }
    .... 마무리 영역 ....
}
```

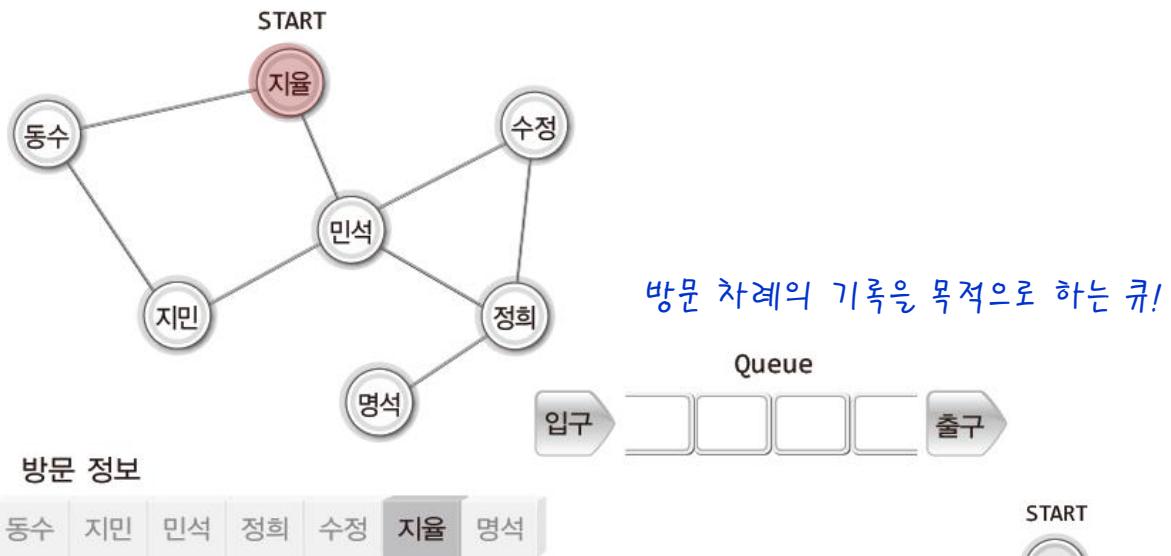
방문한 정점을 떠나야 하니 해당
정보 스택으로!

{
SPush(&stack, visitV);
visitV = nextV;
visitFlag = TRUE;

연결된 다른 정점을 찾아서 방문
을 시도하는 일련의 과정!

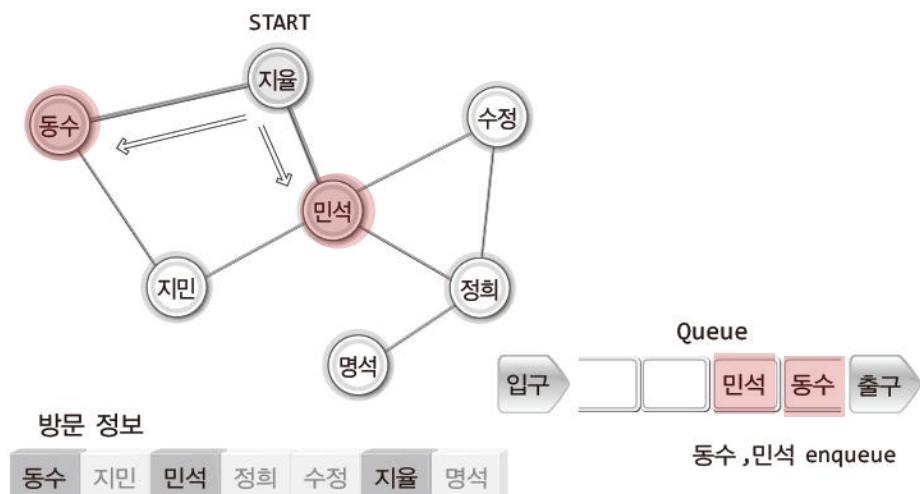
while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE)
{
 if(VisitVertex(pg, nextV) == TRUE)
 {
 SPush(&stack, visitV);
 visitV = nextV;
 visitFlag = TRUE;
 break;
 }
}

너비 우선 탐색의 구현 모델: 과정 1~



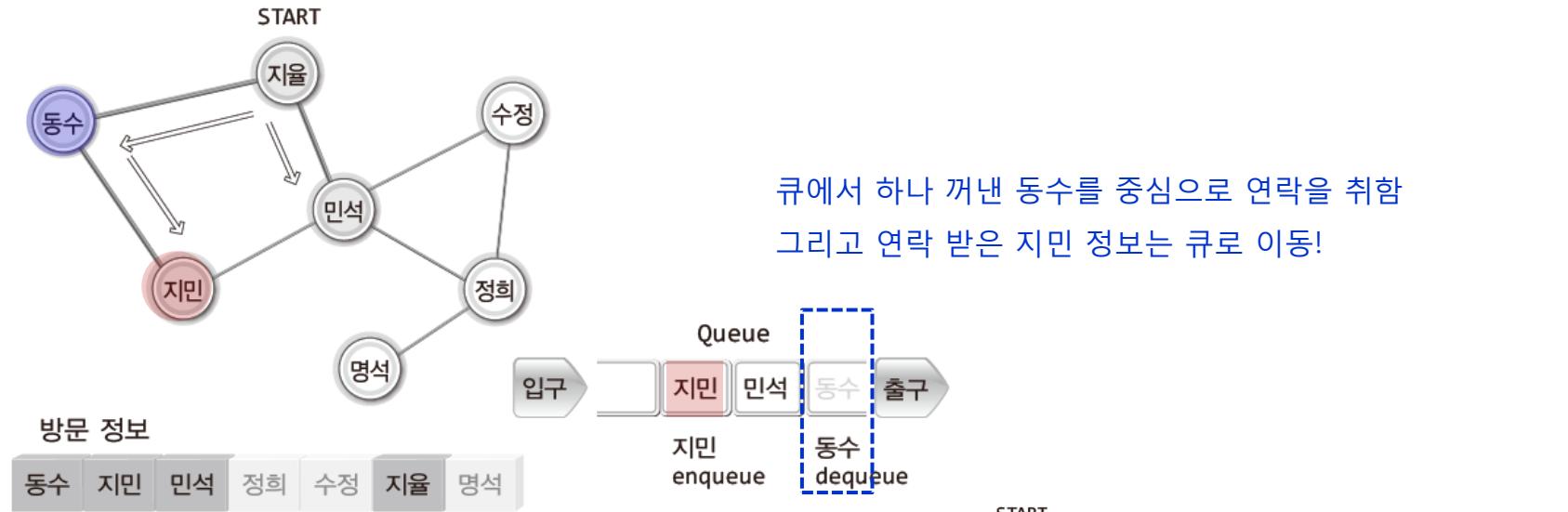
▶ [그림 14-35: BFS의 구현 1/5]

동수와 민석은 연락을 받기만 했을 뿐 연락을 취하지
는 않은 대상! 이러한 대상의 정보를 큐에 저장!



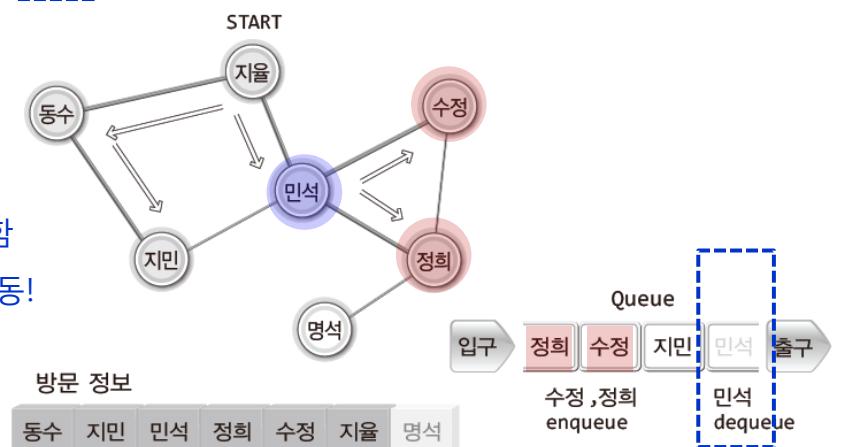
▶ [그림 14-36: BFS의 구현 2/5]

너비 우선 탐색의 구현 모델: 과정 3~



▶ [그림 14-37: BFS의 구현 3/5]

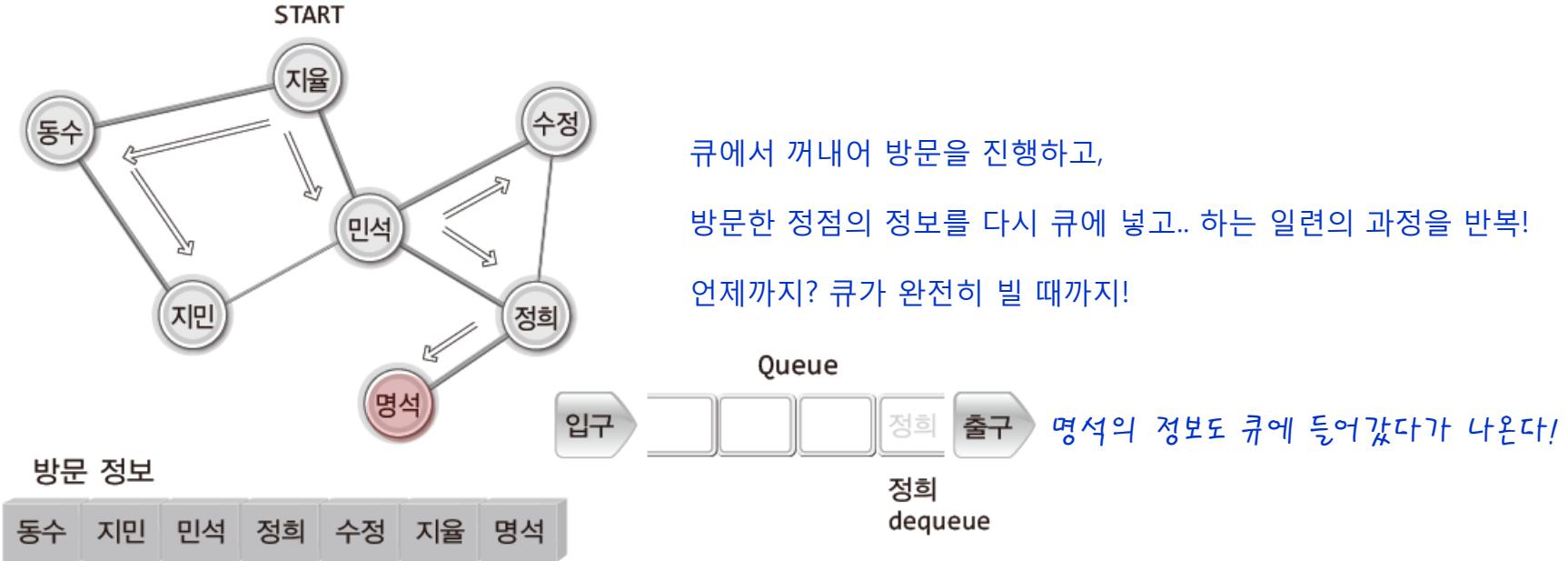
큐에서 하나 꺼낸 민석을 중심으로 연락을 취함
그리고 연락 받은 수정과 정희 정보는 큐로 이동!



▶ [그림 14-38: BFS의 구현 4/5]



너비 우선 탐색의 구현 모델: 과정 5



▶ [그림 14-39: BFS의 구현 5/5]

너비 우선 탐색의 실제 구현

ALGraph.h
ALGraph.c

BFShowGraphVertex 함수의
선언 및 정의 추가하여!

ALGraphBFS.h
ALGraphBFS.c

너비 우선 탐색의 실제 구현

void BFShowGraphVertex(ALGraph * pg, int startV);

- 그래프의 모든 정점 정보를 출력하는 함수
- BFS를 기반으로 정의가 된 함수

구현 결과를 반영한 파일의 구성

- ALGraphBFS.h, ALGraphBFS.c
- CircularQueue.h, CircularQueue.c
- DLinkedList.h, DLinkedList.c
- BFSain.c

그래프 관련
큐 관련(Chapter 07에서 구현)
연결 리스트 관련(Chapter 04에서 구현)



너비 우선 탐색의 실제 구현: ALGraphBFS.h

```
enum {A, B, C, D, E, F, G, H, I, J}; // 정점의 이름들을 상수화
```

```
typedef struct _ual
{
    int numV;      // 정점의 수
    int numE;      // 간선의 수
    List * adjList; // 간선의 정보
    int * visitInfo;
} ALGraph;
```

```
// 그래프의 초기화
```

ALGraphDFS.h와 동일하다!

```
void GraphInit(ALGraph * pg, int nv);
```

```
// 그래프의 리소스 해제
```

```
void GraphDestroy(ALGraph * pg);
```

```
// 간선의 추가
```

```
void AddEdge(ALGraph * pg, int fromV, int toV);
```

```
// 그래프의 간선 정보 출력
```

```
void ShowGraphEdgeInfo(ALGraph * pg);
```

```
// BFS 기반 그래프의 정점 정보 출력
```

```
void BFShowGraphVertex(ALGraph * pg, int startV);
```



너비 우선 탐색의 실제 구현: Helper Func

```
void BFShowGraphVertex(ALGraph * pg, int startV)
{
    Queue queue;
    int visitV = startV;
    int nextV;

    QueueInit(&queue);
    VisitVertex(pg, visitV);

    시작점 방문!
}

while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
{
    visitV에 연결된 정점 정보 얻음
    if(VisitVertex(pg, nextV) == TRUE)
        Enqueue(&queue, nextV);

    while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        계속해서 visitV에 연결된 정점 정보 얻음
        if(VisitVertex(pg, nextV) == TRUE)
            Enqueue(&queue, nextV);
    }

    if(QIsEmpty(&queue) == TRUE)
        break;    큐가 비면 탈출 조건이 성립!
    else
        visitV = Dequeue(&queue);
    }

    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
}
```

코드의 전체적인 느낌이 DFShowGraphVertex와 유사하다. 그리고 그 함수보다 간결하다!

Chapter 14. 그래프

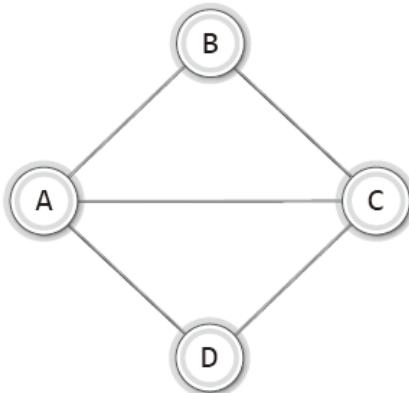


Chapter 14-4:

최소 비용 신장 트리



사이클의 이해



정점 B에서 점점 D에 이르는 단순 경로

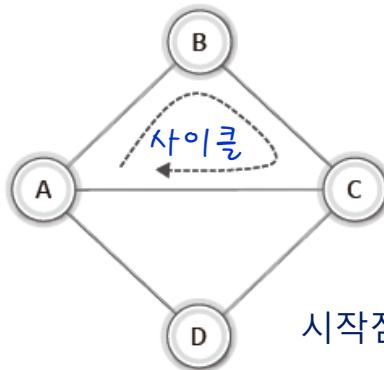
- B-A-D 단순 경로
- B-C-D 단순 경로
- B-A-C-D 조금 돌아가는 단순 경로
- B-C-A-D 조금 돌아가는 단순 경로

단순 경로는 간선을 중복 포함하지 않는다.

단순 경로가 아닌 정점 B에서 점점 D에 이르는 경로

- B-A-C-B-A-D

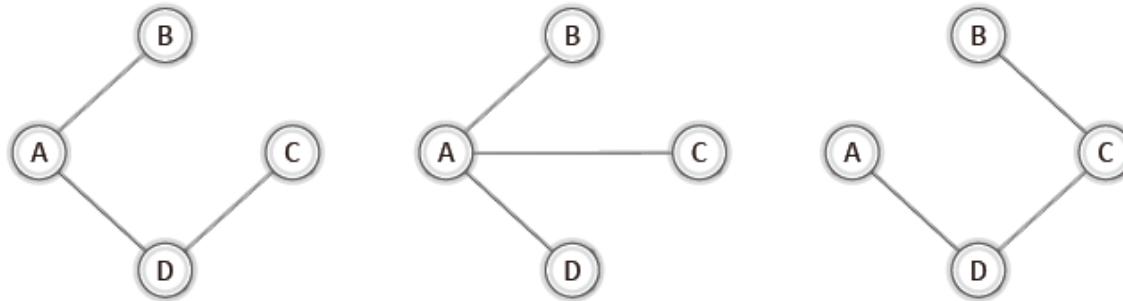
B와 A를 잇는 간선이 두 번 포함됨!



시작점과 끝점이 같은 단순 경로를 가리켜 '사이클' 이라 한다.



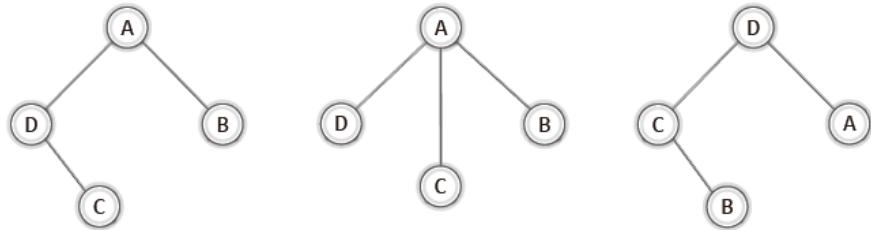
사이클을 형성하지 않는 그래프



어떻게 경로를 구성하더라도 '사이클'을 형성하지 않는 그래프!

이러한 종류의 그래프를 가리켜 '신장 트리'라 한다.

[위의 그래프를 회전시킨 결과](#)



사이클을 형성하지 않는 그래프들은 일종의 트리로 볼 수 있다.

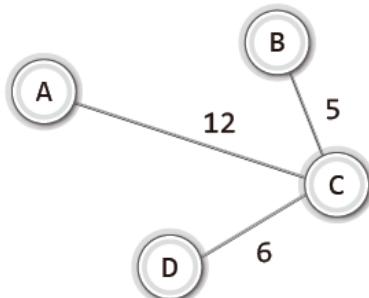
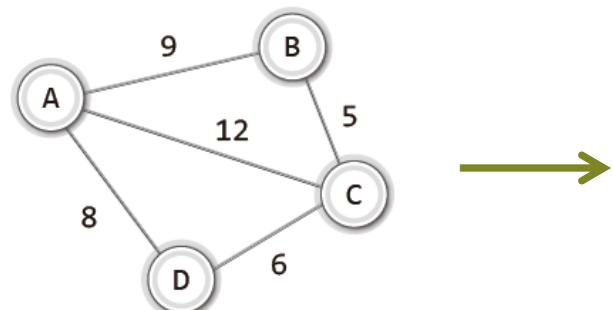
그래서 이들을 가리켜 신장 그래프가 아닌 신장 트리라 한다.



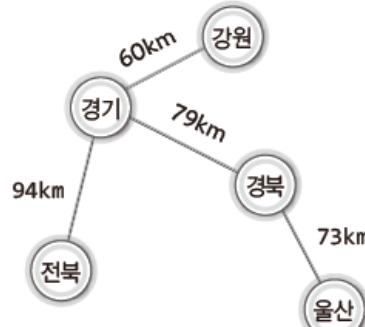
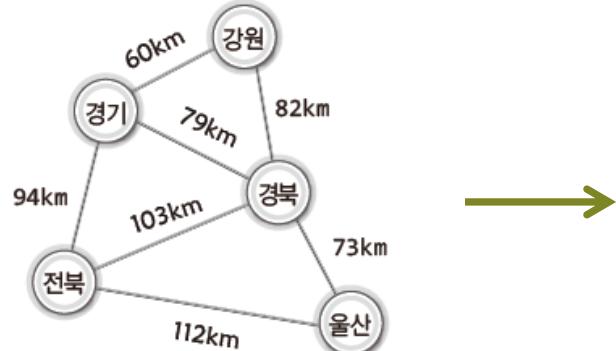
최소 비용 신장 트리의 이해와 적용

- 그리프의 모든 정점이 간선에 의해서 하나로 연결되어 있다.
- 그리프 내에서 사이클을 형성하지 않는다.

신장 트리의 특징



'최소 비용 신장 트리' 구성의 예

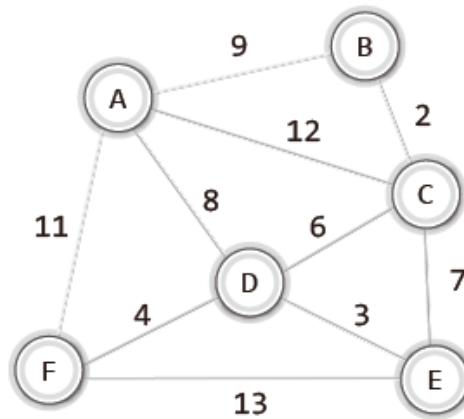


'최소 비용 신장 트리' 구성의 예



크루스칼 알고리즘 1: 과정 1~

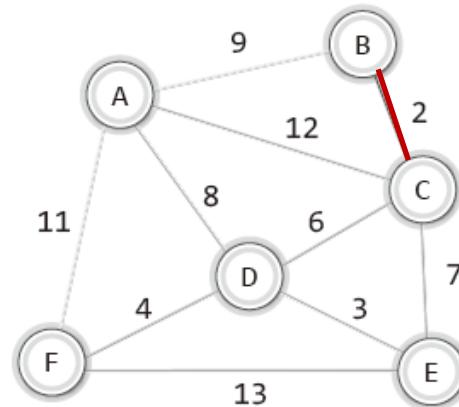
가중치를 기준으로 간선을 정렬한 후에 MST가 될 때까지 간선을 하나씩 선택 또는 삭제해 나가는 방식



▶ [그림 14-49: 크루스칼 알고리즘 1의 1/4]

2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬



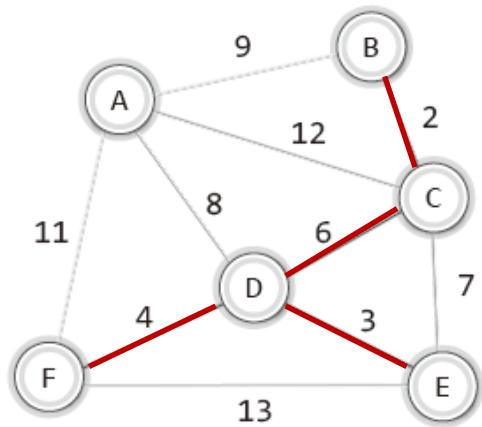
2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

▶ [그림 14-50: 크루스칼 알고리즘 1의 2/4]



크루스칼 알고리즘 1: 과정 3~



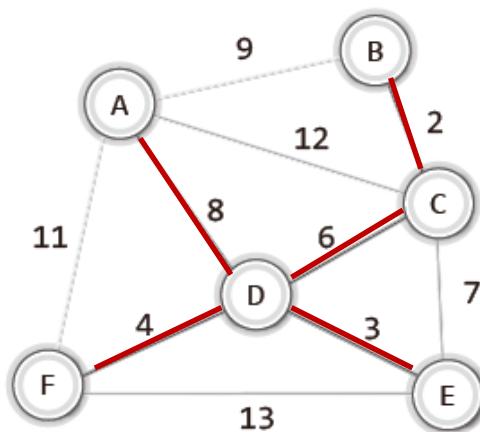
이동
↓↓↓
2, 3, 4, 6, 7, 8, 9, 11, 12, 13
가중치의 오름차순 정렬

▶ [그림 14-51: 크루스칼 알고리즘 1의 3/4]

최소 비용 신장 트리의 조건인

간선의 수 + 1 = 정점의 수를 만족하니

이것으로 최소 비용 신장 트리 형성 완료!



가중치가 7인 간선을 포함시키면
사이클이 형성된다! 따라서 건너 뛴다!

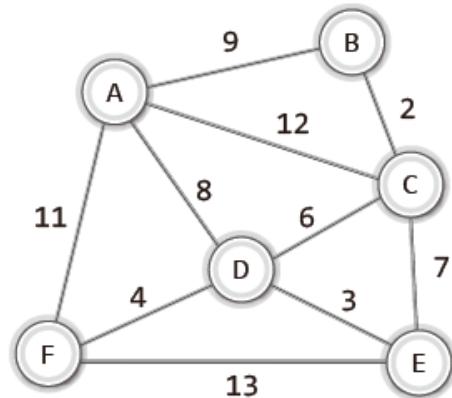
↓
2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

▶ [그림 14-52: 크루스칼 알고리즘 1의 4/4]

크루스칼 알고리즘 2: 과정 1~

높은 가중치의 간선을 하나씩 빼는 방식의 크루스칼 알고리즘

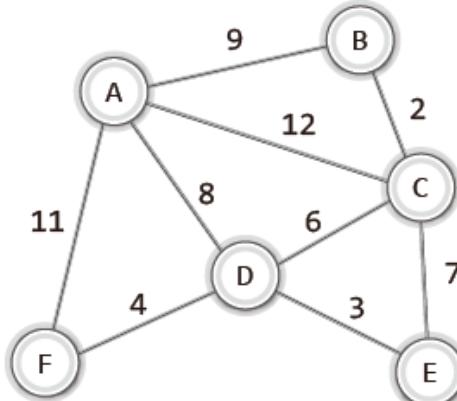


▶ [그림 14-54: 크루스칼 알고리즘 2의 1/4]

가중치가 13인 간선이 없어도 모든 정점은 연결
이 되므로 이를 삭제한다.

13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

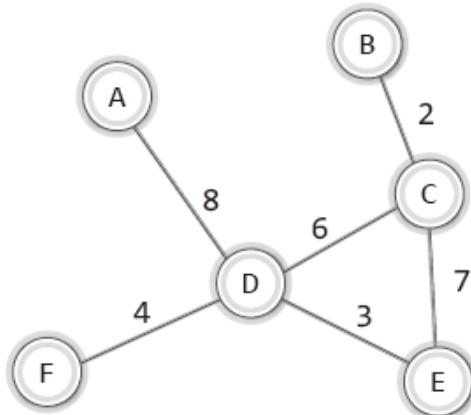


▶ [그림 14-55: 크루스칼 알고리즘 2의 2/4]

13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

크루스칼 알고리즘 2: 과정 3~



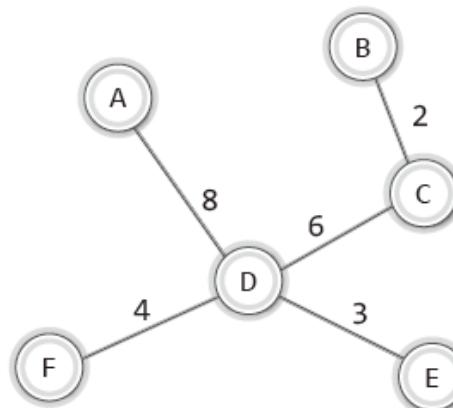
▶ [그림 14-56: 크루스칼 알고리즘 2의 3/4]

최소 비용 신장 트리의 조건인

간선의 수 + 1 = 정점의 수를 만족하니

이것으로 최소 비용 신장 트리 형성 완료!

가중치가 8인 간선이 없으면 정점 A와 정점 D가
연결되지 않는다. 따라서 통과~!
이동
13, 12, 11, 9, 8, 7, 6, 4, 3, 2
가중치의 내림차순 정렬



▶ [그림 14-57: 크루스칼 알고리즘 2의 4/4]

13, 12, 11, 9, 8, 7, 6, 4, 3, 2
가중치의 내림차순 정렬

크루스칼 알고리즘의 구현을 위한 계획1

우리가 선택한 구현 방식

가중치를 기준으로 간선을 내림차순으로 정렬한 다음 높은 가중치의 간선부터 시작해서 하나씩 그래프에서 제거하는 방식

구현에 사용할 도구들

- DLinkedList.h, DLinkedList.c
- ArrayBaseStack.h, ArrayBaseStack.c
- ALGraphDFS.h, ALGraphDFS.c

연결 리스트
배열 기반 스택
깊이 우선 탐색을 포함하는 그래프



크루스칼 알고리즘이 담기는 파일들

- ALGraphKruskal.h, ALGraphKruskal.c

가중치 그래프의 구현 결과

그리고 가중치 그래프의 구현을 위해서는 가중치가 포함된 간선을 표현한 구조체가 정의되어야 한다.
헤더파일 ALEdge.h를 만들어서 해당 구조체를 정의한다.



크루스칼 알고리즘의 구현을 위한 계획2

다음 질문에 답을 하는 함수가 필요하다!

이 간선을 삭제한 후에도 이 간선에 의해 연결된 두 정점을 연결하는 경로가 있는가?

이를 위해 DFS 알고리즘을 활용! DFS의 구현결과인 DFShowGraphVertex 함수를 확장하여 이 질문에 답을 하도록 한다!

이는 크루스칼 알고리즘의 일부이다.

그래프를 구성하는 간선들을 가중치를 기준으로 정렬할 수 있어야 한다.

이를 위해서 앞서 구현한 우선순위 큐를 활용

- PriorityQueue.h, PriorityQueue.c 우선순위 큐
- UsefulHeap.h, UsefulHeap.c 우선순위 큐의 기반이 되는 힙



크루스칼 알고리즘의 구현을 위한 계획3

- | | |
|--------------------------------------|-------------------------|
| • DLinkedList.h, DLinkedList.c | 연결 리스트 |
| • ArrayBaseStack.h, ArrayBaseStack.c | 배열 기반 스택 |
| • ALGraphKruskal.h, ALGraphKruskal.c | 크루스칼 알고리즘 기반의 그래프 |
| • PriorityQueue.h, PriorityQueue.c | 우선순위 큐 |
| • UsefulHeap.h, UsefulHeap.c | 우선순위 큐의 기반이 되는 힙 |
| • ALEdge.h | 가중치가 포함된 간선의 표현을 위한 구조체 |

최종적으로 크루스칼 알고리즘의 구현을 보이기 위한 프로젝트의 헤더파일과 소스파일의 구성



크루스칼 알고리즘의 구현: 헤더파일

```
enum {A, B, C, D, E, F, G, H, I, J};
```

```
typedef struct _ual
```

```
{
```

```
    int numV;
```

```
    int numE;
```

```
    List * adjList;
```

```
    int * visitInfo;
```

```
    PQueue pqueue; // 간선의 가중치 정보 저장
```

```
} ALGraph;
```

```
void GraphInit(ALGraph * pg, int nv);
```

```
void GraphDestroy(ALGraph * pg);
```

```
void AddEdge(ALGraph * pg, int fromV, int toV, int weight);
```

```
void ShowGraphEdgeInfo(ALGraph * pg);
```

```
void DFShowGraphVertex(ALGraph * pg, int startV);
```

```
void ConKruskalMST(ALGraph * pg); // 최소 비용 신장 트리의 구성
```

```
void ShowGraphEdgeWeightInfo(ALGraph * pg); // 가중치 정보 출력
```

```
typedef struct _edge
```

```
{
```

```
    int v1; // 간선이 연결하는 첫 번째 정점
```

```
    int v2; // 간선이 연결하는 두 번째 정점
```

```
    int weight; // 간선의 가중치
```

```
} Edge;
```

ALEdge.h

ALGraphKruskal.h



크루스칼 알고리즘을 구현한 함수: 수정된 함수들

```
void GraphInit(ALGraph * pg, int nv)
{
    . . . . 여기까지는 ALGraphDFS.c의 GraphInit 함수와 동일 . . .

    // 우선순위 큐의 초기화
    PQueueInit(&(pg->pqueue), PQWeightComp); // 추가된 문장
}
```

```
int PQWeightComp(Edge d1, Edge d2)
{
    return d1.weight - d2.weight;
}
```

가중치 기준 내림차순으로
간선 정보 꺼내기 위한 정의!

```
void AddEdge(ALGraph * pg, int fromV, int toV, int weight)
{
    Edge edge = {fromV, toV, weight}; // 간선의 가중치 정보를 담음
    LInsert(&(pg->adjList[fromV]), toV);
    LInsert(&(pg->adjList[toV]), fromV);
    pg->numE += 1;

    // 간선의 가중치 정보를 우선순위 큐에 저장
    PEnqueue(&(pg->pqueue), edge);
}
```



크루스칼 알고리즘을 구현한 함수: ConKruskalMST

```
void ConKruskalMST(ALGraph * pg)      // 크루스칼 알고리즘 기반 MST의 구성
{
    Edge recvEdge[20];    // 복원할 간선의 정보 저장
    Edge edge;
    int eidx = 0;
    int i;

    // MST를 형성할 때까지 아래의 while문을 반복
    while(pg->numE+1 > pg->numV)      // MST 간선의 수 + 1 == 정점의 수
    {
        edge = PDequeue(&(pg->pqueue)); 가중치 순으로 간선 정보 획득!
        RemoveEdge(pg, edge.v1, edge.v2); 획득한 정보의 간선 실제 삭제!

        if(!IsConnVertex(pg, edge.v1, edge.v2)) 삭제 후 두 정점 연결 경로 있는지 확인!
        {
            RecoverEdge(pg, edge.v1, edge.v2, edge.weight); 연결 경로 없으면 간선 복원!
            recvEdge[eidx++] = edge;
        }
    }

    // 우선순위 큐에서 삭제된 간선의 정보를 회복
    for(i=0; i<eidx; i++)
        PE enqueue(&(pg->pqueue), recvEdge[i]);
}
```

- RemoveEdge 그래프에서 간선을 삭제한다.
- IsConnVertex 두 정점이 연결되어 있는지 확인한다.
- RecoverEdge 삭제된 간선을 다시 삽입한다.



크루스칼 알고리즘의 완성을 돋는 함수들 1

```
// 간선의 소멸
void RemoveEdge(ALGraph * pg, int fromV, int toV)
{
    RemoveWayEdge(pg, fromV, toV);
    RemoveWayEdge(pg, toV, fromV);
    (pg->numE)--;
}
```

인접 리스트 기반 무방향 그래프인 관계로 하나의 간선을 완전히 소멸하기 위해서는 두 개의 간선 정보를 소멸시켜야 한다.

```
void RecoverEdge(ALGraph * pg, int fromV, int toV, int weight)
{
    LInsert(&(pg->adjList[fromV]), toV);
    LInsert(&(pg->adjList[toV]), fromV);
    (pg->numE)++;
}
```

AddEdge 함수와 달리 간선의 가중치 정보를 별도로 저장하지 않는다. 이렇듯 가중치 정보를 별도로 저장하지 않는 이유는 크루스칼 알고리즘의 구현 내용을 통해 이해할 수 있다.



크루스칼 알고리즘의 완성을 돋는 함수들 2

```
// 한쪽 방향의 간선 소멸
void RemoveWayEdge(ALGraph * pg, int fromV, int toV)
{
    int edge;
    if(LFirst(&(pg->adjList[fromV]), &edge))
    {
        if(edge == toV) {
            LRemove(&(pg->adjList[fromV]));
            return;
        }
        while(LNext(&(pg->adjList[fromV]), &edge))
        {
            if(edge == toV) {
                LRemove(&(pg->adjList[fromV]));
                return;
            }
        }
    }
}
```

이렇듯 RemoveEdge 함수의 완성을 돋는 RemoveWayEdge 함수를 별도로 정의하면 방향 그래프의 구현을 위한 확장이 용이하다!



크루스칼 알고리즘의 완성을 돋는 함수들 3

```
// 인자로 전달된 두 정점이 연결되어 있다면 TRUE, 그렇지 않다면 FALSE 반환
int IsConnVertex(ALGraph * pg, int v1, int v2)
{
    Stack stack;
    int visitV = v1;
    int nextV;

    StackInit(&stack);
    VisitVertex(pg, visitV);
    SPush(&stack, visitV);

    while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        int visitFlag = FALSE;
        // 정점을 돌아다니는 도중에 목표를 찾는다면 TRUE를 반환한다.
        if(nextV == v2) {
            // 함수가 반환하기 전에 초기화를 진행한다.
            memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
            return TRUE;      // 목표를 찾았으니 TRUE를 반환!
        }
    }
}
```

```
if(VisitVertex(pg, nextV) == TRUE)
{
    SPush(&stack, visitV);
    visitV = nextV;
    visitFlag = TRUE;
}
else
{
    while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        // 정점을 돌아다니는 도중에 목표를 찾는다면 TRUE를 반환한다.
        if(nextV == v2) {
            // 함수가 반환하기 전에 초기화를 진행한다.
            memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
            return TRUE;      // 목표를 찾았으니 TRUE를 반환!
        }

        if(VisitVertex(pg, nextV) == TRUE) {
            SPush(&stack, visitV);
            visitV = nextV;
            visitFlag = TRUE;
            break;
        }
    }
}
```

DFShowGraphVertex 함수와의 비교를 통해서 어떻게 수정되었고 또 그 결과 어떻게 두 정점의 연결을 확인하는지 이해하자!



크루스칼 알고리즘의 완성을 돋는 함수들 3

```
if(visitFlag == FALSE)
{
    if(SIsEmpty(&stack) == TRUE)
        break;
    else
        visitV = SPop(&stack);
}
}

memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
return FALSE;      // 여기까지 왔다는 것은 목표를 찾지 못했다는 것!
}
```

이로써 부분적으로 필요한 모든 설명이 완료 되었으니 전체 코드를 확인하고 교재에서 제공하는 main 함수의 실행 결과도 직접 확인해보자!



수고하셨습니다~



Chapter 14에 대한 강의를 마칩니다!

