<8장, 트리(TREES), 김성현, 2019107283>

8.1, 트리의 개념	8.2 이진트리소개 (binary tree)	8.3 이진 트리의 표현 (배열 vs 포인터)	8.4 이진 트리의 순회 (traversal)	8.5 반복적 순회
8.11 이진탐색트리 (search에 사용)	8.10 스레드 이진트리	8.9 이진트리의 기타연산	8.8 트리의 응용: 수식트리	8.7 트리의 응용	8.6 레벨 순회

8.12 이진탐색트리의 응용: 영어 사전.

<교재 순서>에 따라서 모든 트리를 배우고 교재로 하자.

8.1, 트리(tree)의 개념

- 트리란?
- 트리의 정의
- 트리의 용어
- 트리의 종류
  - 이진트리
  - 일반트리

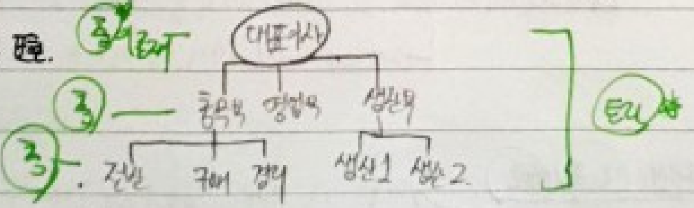
• 트리 (tree)란?

- 트리: 계층적인(hierarchical) 구조를 가지는 자료. (ex. 가계도)
  - 배열, 스택, 큐, 리스트 등은 선형(linear) 구조.
- 트리는 부모(parent) - 자식(child) 관계를 표현하는 노드(node)와 링크(link)로 이루어진다.
- 응용 분야:
  - 회사의 조직 표현
  - 컴퓨터 파일 시스템의 디렉토리 구조. (우리 수업할 때 리눅스의 directory 구조도 여기에 해당)
  - 인공지능에서의 결정 트리(decision tree).
  - ... 그 외 많은 구조

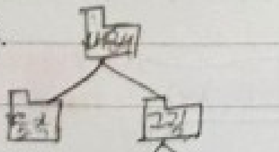
여기서 전체적으로 싹 보고 뒤에서 세부적으로 공부 할 예정이다.

• 트리 응용

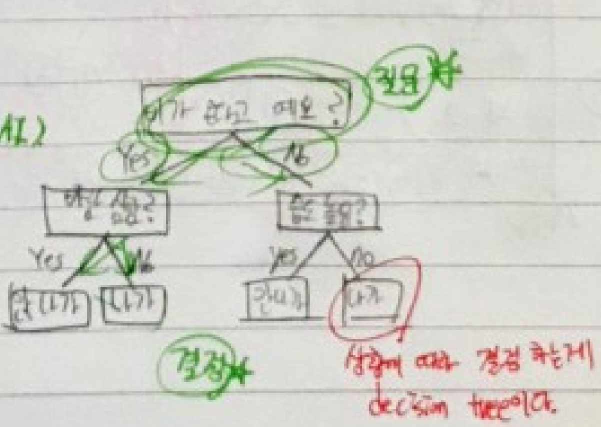
- 회사의 조직 표현.



- 디렉토리 구조.



- 결정 트리 (decision tree) (in 인공지능, AI)



p.236~256

• 트리의 정의

■ 트리(tree)는 노드(node)와 링크(link)로 구성되며 자료구조(특징: 사이클이 없음)

■ 노드(node): 데이터를 저장하는 장소

■ 링크(link): 부모(parent)노드에서 자식(child)노드로 연결되는 선.

■ 노드의 종류(정의보다 중요!!)

■ 루트(root)노드: 부모가 없는 노드. → (가장 위에 있는 노드)

■ 단말(terminal 혹은 leaf)노드: 자식이 없는 노드 → (가장 밑에 있는 가장 끝 노드)

■ 내부(internal)노드: 하나 이상의 자식이 있는 노드. → (중간에 자식 있는 노드)

■ 형제(sibling)노드: 같은 부모를 가지는 노드들. → (같은 부모에서 뻗어 나온 자식 노드들)

p.257

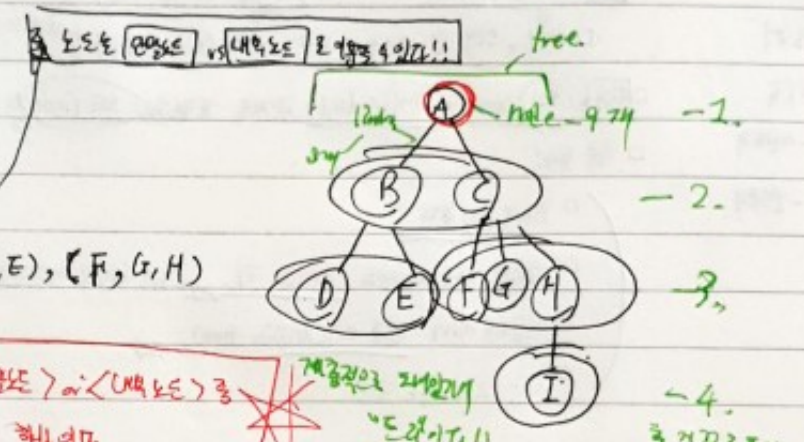
• 노드의 종류(예시)

■ 루트 노드: A

■ 단말 노드: D, E, F, G, I

■ 내부 노드: A, B, C, H

■ 형제 노드: (B, C), (D, E), (F, G, H)



정리하자면, 모든 노드는 <단말노드> or <내부노드> 중 하나이다.

↳ 특별하게 정해 놓은 parent 없는 노드를 "루트 노드"라 한다!!

↳ 상황에 따라서 <단말>이 될 수도 있고, <내부>가 될 수도 있다!!

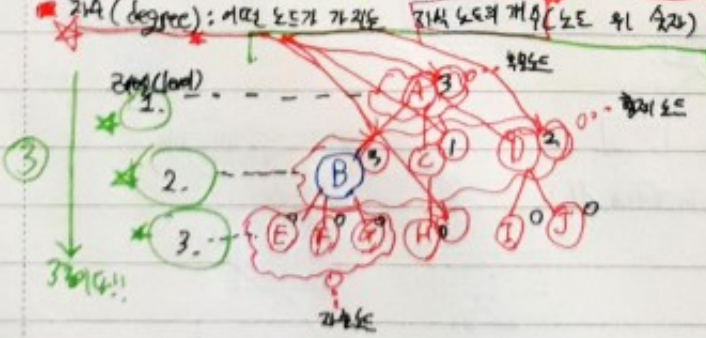
p.255~256

• 트리의 용어

■ 레벨(Level): 트리의 각 층의 번호 (계층)

■ 높이(height): 트리의 최대 레벨 (아래 트리에서는 3일)

■ 차수(degree): 어떤 노드가 가지는 자식 노드의 개수(노드 위 숫자)



p.257 • 트리의 종류.

- 트리
  - 이진(binary) 트리 — 우리 앞으로 배울 것!!
  - 일반 트리

→ 모든 노드의 child가 최대 2개 (2개!!) 0, 1, 2 를 하나를 만족.

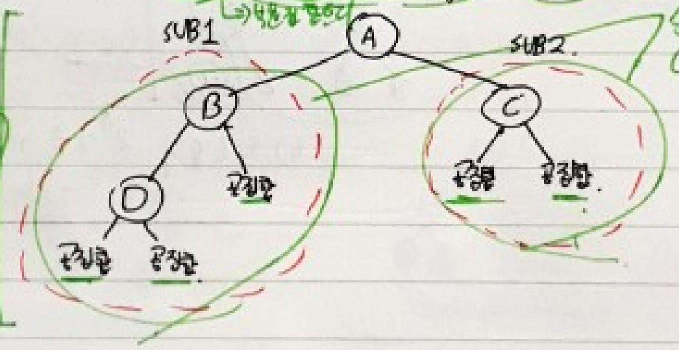
→ 노드 child 갯수 제한 X (2개 이상도 가능).

p.258~263. 8.2 이진트리(binary tree)

- 이진트리의 정의
- 이진트리의 성질
- 이진트리의 분류
- • 교재에서의 정의 ⇒ (1) 공집합이거나, (2) 루트와 왼쪽 서브트리, 오른쪽 서브트리로 구성된 노드들의 유한 집합

p.258~259 • 이진 트리(binary tree)의 정의.

- 이진트리: 모든 노드가 2개의 서브트리(subtree)를 가지는 트리 (서브트리는 공집합일 수도 있음) → 서브트리가 없을 수도 있다! ⇒ 2개를 가질 수 있거나 각각을 없을 수도 있다! 2개, 1개, 0개일 수도 있다
- (요점) 이진 트리의 노드에는 최대 2개까지의 자식 노드가 존재.
- 모든 노드의 차수가 2 이하임 (구현이 편리)
- 이진 트리에는 서브트리 간의 순서 (left 및 right)가 존재 → 일반 트리

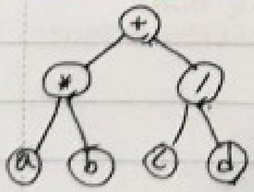


p.259~261 • 이진 트리의 성질 (1번)

- 노드의 개수가 n이면 링크의 개수는 n-1임.

노드: n
링크: n-1

→ 이진트리는 노드 하나 말고는 하나의 링크가 필요한데, 루트노드는 링크 필요 X 그래서 이렇다!!! → 필요 X



• 노드의 개수: n=7

• 링크의 개수: n-1=6.

모든 노드는 하나 있으며 parent에 연결되어야 하는데, 루트노드만 parent가 없으니 링크가 노드랑 같은데 필요없어지면 이 때문에 -1 되어서 링크가 하나 적어지는 것이다!!

최소 h개 => 일렬 최소 연결
최대 2^h-1개 => 모두 꽉 채워 연결
(완전)

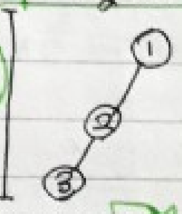
• 이진 트리의 성질 (2/3)

□ 높이(height)가 h인 이진 트리의 경우, 최소 h개의 노드를 가지며, 최대 2^h-1개의 노드를 가진다.

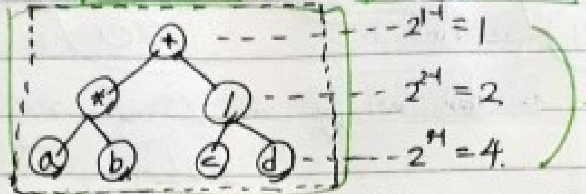
h ☆ 높이=3

- 노드 : n
- → 링크 : n-1

- 높이 : h
- → 최소노드 : h개
- → 최대노드 : 2^h-1개



최소 노드 개수 = 3 ☆



최대 노드 개수 $= 2^{1-1}+2^{2-1}+2^{3-1} = 1+2+4 = 7.$

☆ h=3

7개 → 2^3-1 ☆

• 이진 트리의 성질 (3/3)

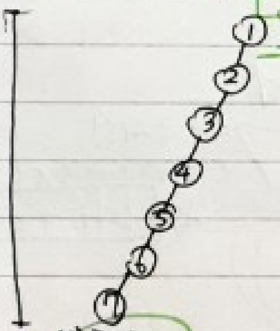
□ n개의 노드를 가지는 이진 트리의 높이

□ 최대 높이 : n

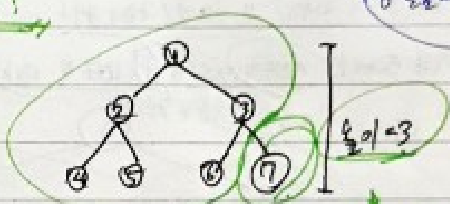
□ 최소 높이 : $\lceil \log_2(n+1) \rceil$ $\lceil 3.7 \rceil \Rightarrow 4$ (가장 가까운 정수로 올림!!)

☆ 올림해준다!! 반올림 X, 다음 정수로!!

높이=7



(a) 최대 높이.



(b) 최소 높이.

2^h-1을 거꾸로 한다!!
$\Rightarrow \log_2(n+1)$ ☆

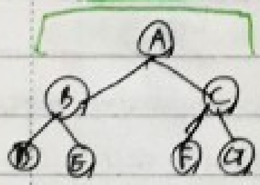
p.261~263 • 이진 트리의 분류.

□ 포화(full) 이진 트리 — 꽉 찬 거

□ 완전(complete) 이진 트리. — 규칙적인

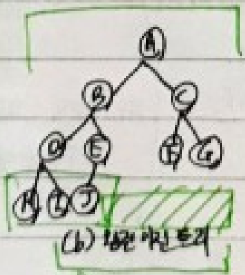
□ 기타 이진 트리. — 기타

트리는 무조건 '왼쪽'부터 그리니까, full 채워서 '또 그려서' '오른쪽'이 빈 것이다!!



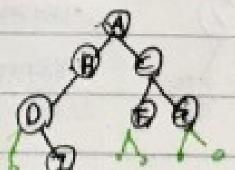
(a) 포화 이진 트리

노드에 1개씩 꽉 찼다!!
(h까지 그래서 꽉 찼다!)



(b) 완전 이진 트리

full X,
포화 이진트리에서 '오른쪽'이 다 빠진 거,
왼쪽부터 그리기에 '규칙적' 나오는, ~덜 그린 것이라고
표시한다.



(c) 기타 이진 트리.

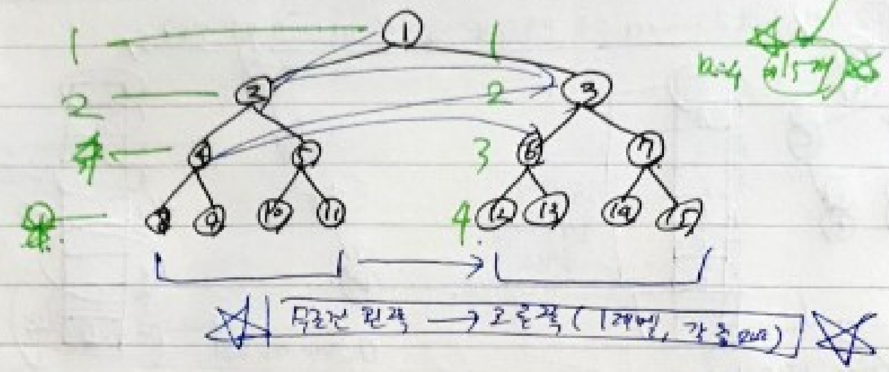
기타 것들 모든 거.

- 포화(full) 이진 트리.

■ 트리의 각 레벨에 노드가 꽉 차있는 이진 트리를 의미한다. (한 줄이 전부 꽉 차있다!!)

전체노드 개수 : $2^{1-1} + 2^{2-1} + 2^{3-1} + \cdots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = 2^k - 1.$

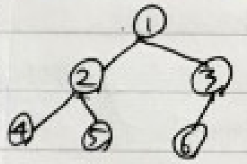
■ 포화 이진 트리에는 다음과 같이 각 노드에 번호를 붙일 수 있다.



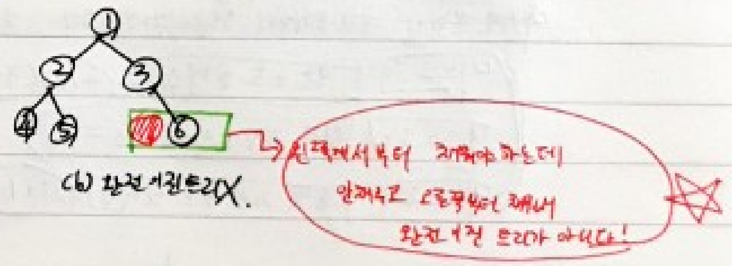
- 완전 (complete) 이진 트리.

■ 레벨 1부터 k-1까지는 노드가 모두 채워져 있고 마지막 레벨 k에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진 트리

■ 포화 이진 트리와 노드 번호가 일치.



(a) 완전 이진 트리



(b) 완전 이진트리X.

p.263~266 8.3 이진 트리의 표현

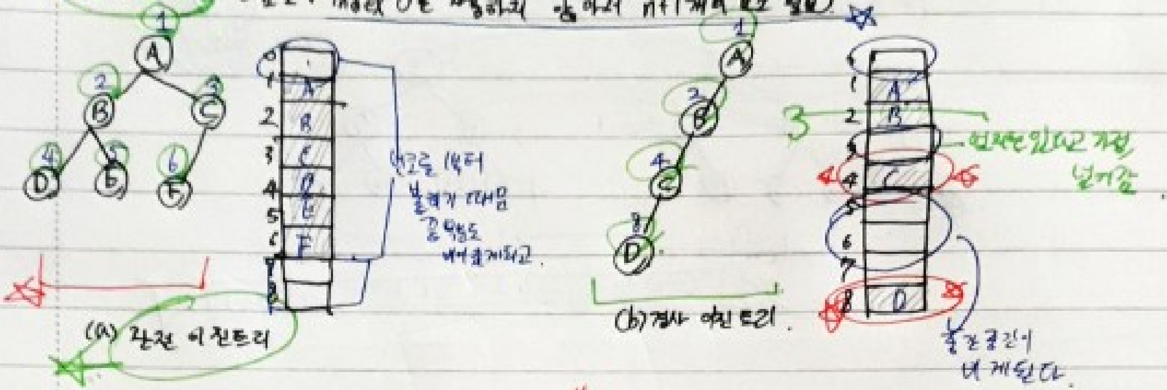
- 배열(array)을 이용
- 링크 표현(pointer)을 이용

• 트리의 배열 표현

□ 배열 표현: 모든 이진 트리를 n개의 노드를 가지는 포화 이진 트리라고 가정하고 (이해 해보고 넘어가야 한다!!)

각 노드에 번호를 붙여서 그 번호를 배열의 인덱스로 삼아 노드의 데이터를 배열에 저장하는 방법.

★ (full-이진트리 가정) (참고: index 0은 사용하지 않아서 n+1개의 요소 필요) ☆



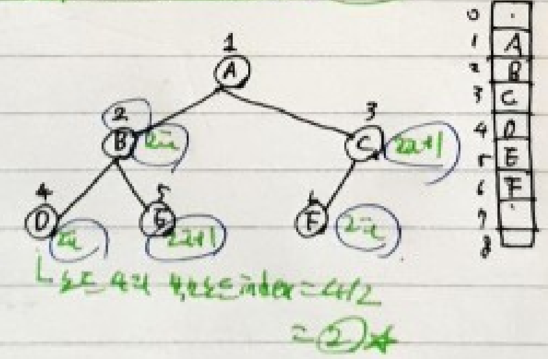
배열로 트리를 표현할 때에는
반드시 "포화 이진 트리"라는 가정이 필요!!
└ 기억공간 낭비 ↑

• 트리의 배열 표현 - 인덱스 특징.

□ 배열 표현의 이진 트리에서 부모와 자식 노드의 인덱스(index) 관계.

- □ 노드 i의 부모 노드 인덱스 = i/2 (주의: integer 값)
- □ 노드 i의 왼쪽 자식 노드 인덱스 = 2i
- □ 노드 i의 오른쪽 자식 노드 인덱스 = 2i+1.

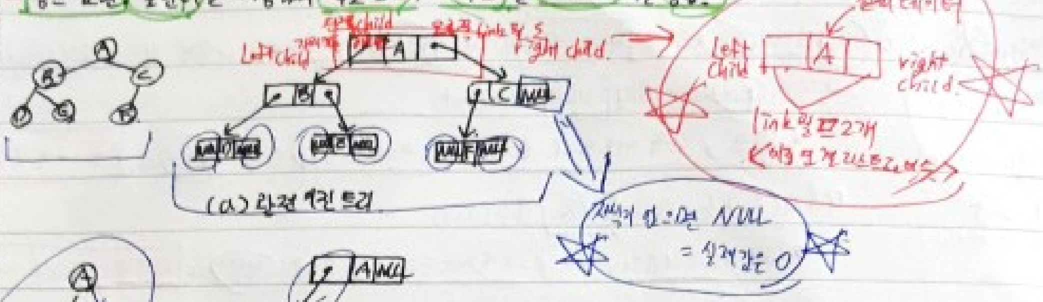
└ 꼭, 암기! → 규칙!!



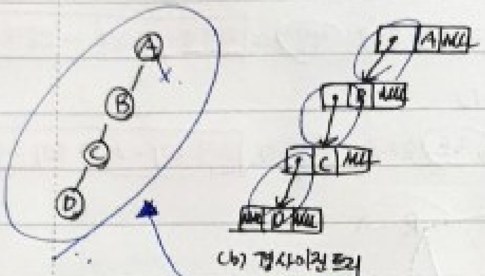
어떤 노드의 왼쪽 / 오른쪽 자식 노드를 찾아갈 수도 있고,
부모노드를 곧바로 찾아갈 수도 있다.
<배열의 장점>

• 트리의 링크 표현, 그래서 링크로 많은 프로그램에서 표현!!

□ 링크 표현: 포인터를 사용하여 부모 노드가 자식 노드를 가리키게 하는 방법.



(a) 완전 이진 트리



(b) 경사 이진 트리

왼쪽 child만 있어서 "경사져 있다" 해서 (경사 이진 트리다).

• 트리의 링크 표현 - 구조체

□ 트리의 각 노드는 구조체로 표현.

□ 링크는 구조체 내 포인터로 표현 (이진 트리의 경우 left 및 right로 표현되는 2개 링크가 필요).

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

총 3개의 필드 (data + link 2개)

struct TreeNode, 즉 그기 자신 노드의 구조체로 선언하여 똑같은 형식의 양쪽 자식들을 또 가리킨다.

이런걸 여러 개 만들어서 다닥 연결해주는 거임!

• 트리의 링크 표현 - 프로그램 보기.

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

typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right;
} TreeNode;
//      n1(10)
//     /     \
//  n2(20)  n3(30)
void main()
{
    TreeNode *n1, *n2, *n3;
```

노드 3개 트리

```
    n1 = (TreeNode *)malloc(sizeof(TreeNode));
    n2 = (TreeNode *)malloc(sizeof(TreeNode));
    n3 = (TreeNode *)malloc(sizeof(TreeNode));
    n1->data = 10;
    n1->left = n2; n1->right = n3;

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

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

p.267~276 8.4 이진 트리의 순회(traversal)

- 순회?
- 전위순회 root먼저
- 중위순회 root중간
- 후위순회 root나중에
- 까지. ☆

이진트리 다 다시해야!!

• 순회(traversal)란?

□ 순회: 트리의 모든 노드들을 어떤 규칙에 따라 방문하는 것. → 방문 or 출력 시 이 순회(traversal)이 필요하다.

□ 전위(preorder)순회: VLR 먼저(root)

■ 루트, 왼쪽 서브트리 순회, 오른쪽 서브트리 순회 순으로 방문한다. 루트 - 왼쪽순회 - 오른쪽 순회

□ 중위(inorder)순회: LVR 중간(root)

■ 왼쪽 서브트리순회, 루트, 오른쪽 서브트리 순회 순으로 방문한다. 왼쪽순회 - 루트 - 오른쪽순회

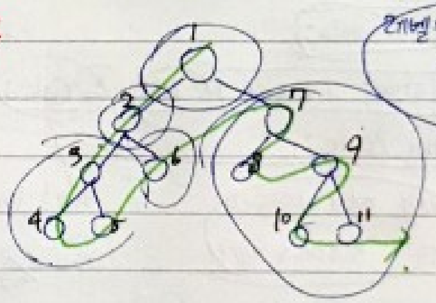
□ 후위(postorder)순회: LRV 마지막(root)

■ 왼쪽 서브트리순회, 오른쪽 서브트리 순회, 루트 순으로 방문한다. 왼쪽 순회 - 오른쪽 순회 - 루트

☆ 기준: root ☆

• 전위(preorder) 순회

1. 루트 노드를 방문
2. 왼쪽 서브트리를 (전위 순회로) 순회. ☆
3. 오른쪽 서브트리를 (전위 순회로) 순회. ☆



레벨을 내려갈 때마다 그 레벨에서의 root를 기준으로 순회해야한다!!

• 전위순회 - 프로그램

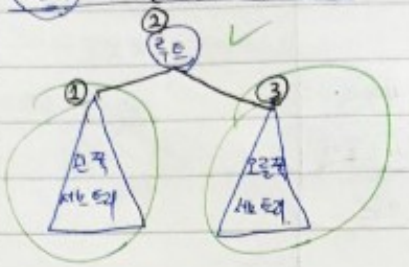
□ 순환(recursive) 호출을 이용한다!

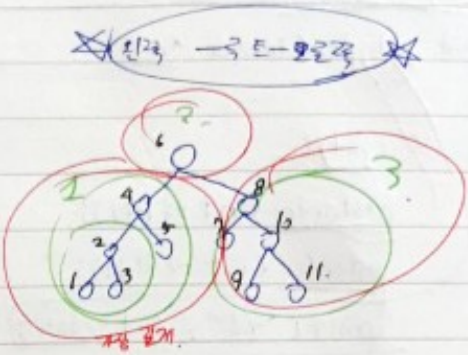
```
void preorder(TreeNode *root)
{
    if (root) {
        printf("%d", root->data);   //루트 먼저
        preorder(root->left);  다시 preorder 순환 //왼쪽 순환(recall)
        preorder(root->right); 다시 preorder 순환 //오른쪽 순환(recall)
    }
}
```

• 중위(inorder) 순회.

1. 왼쪽 서브트리를 (중위 순회로) 순회한다.
2. 루트 노드를 방문한다.
3. 오른쪽 서브트리를 (중위 순회로) 순회한다.

왼쪽 → 루트 → 오른쪽





• 중위 순회 - 프로그램.

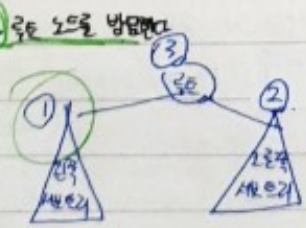
□ 순환(recursive) 호출을 이용한다.

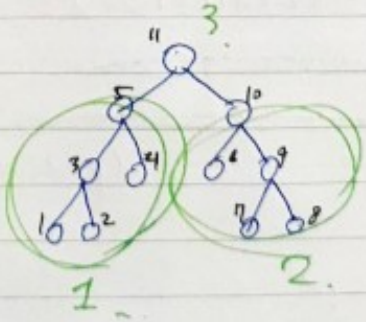
```
void inorder (TreeNode *root)
{
    if (root) {
        inorder (root->left);          // 왼쪽 먼저
        printf("%d", root->data);      // 루트 중간
        inorder (root->right);         // 오른쪽 마지막.
    }
}
```

• 후위(postorder) 순회

1. 왼쪽 서브트리를 (후위 순회로) 순회한다.
2. 오른쪽 서브트리를 (후위 순회로) 순회한다.
3. 루트 노드를 방문한다.

왼쪽 → 오른쪽 → 루트





post • 후위 순회_프로그램.

□ 순환 (recursive) 프로그램 이용함.

```
void postorder(TreeNode *root)
{
  if (root) {
    postorder(root->left);   //왼쪽
    postorder(root->right);  //오른쪽
    printf("%d", root->data); //루트.
  }
}
```

p.277~278 8.5, 반복적인 순회.

• 보통은 순회를 "순환"으로 구현 <더욱 효율적인 방법>

⇒ but, 우리가 만든 "스택"을 이용해서도 구현할 수 있다!!

↳ 이 경우 "스택"을 통해 루트를 넘어왔다 갈 수 있다!!

(p.277~278 자세한 코드 참조)

자료 비슷한 맥락으로 뒤에 나온 탐색트리에서

두 가지 방법을 살펴볼 것이다.

p.279~282 8.6. 레벨 순회.

• 효율적인 것 같지는 않지만 자주 사용

- 레벨 순회 (level order) ⇒ 각 노드를 레벨 순으로 검사하는 순회 방법.

└ 동일 레벨은 좌 → 우.

다른 레벨 → 1 ~ 높은 순으로

(p.280~282 코드 참고)

8.7~8.8 트리응용

- 수식 트리
- 이진트리 용량 계산

a+b

p.283~285. 8.7 트리의응용 : 수식 트리 처리.

• 수식 트리

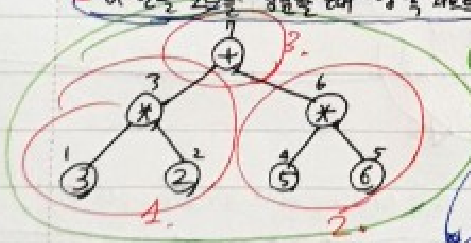
- 수식 트리 : 산술 식을 트리 형태로 표현한 것. 식 → 트리
  - 비 단말 노드 : 연산자(operator)
  - 단말 노드 : 피연산자(operand) ☆ 먼저 연산자 → 마지막에 피연산자 ☆
- 예)

수식	a+b	a-(b×c)	(a<b) or (c<d)
전위순회 preorder	+ab	-a×bc	or <ab <cd
중위순회 inorder	a+b	a-(b×c)	(a<b) or (c<d)
후위순회 postorder	ab+	abc×-	ab <cd <or

☆☆ 수식 계산할 때는 후위순회 사용 ☆☆☆

• 수식 트리 - 계산

- 후위 순회를 사용 → 계산할 때 사용!!
- 서브트리의 값을 순환(recursive) 호출로 계산.
- 비 단말 노드를 방문할 때 양쪽 서브트리의 값을 노드에 저장된 연산자를 이용하여 계산한다.



(3×2)+(5×6)

⇒ 계산 postorder traversal 하면서

피연산자 단말일 경우 그대로 두고,

(연산자) 비단말일 경우, 양쪽 서브트리의 노드의 값을 비단말의 operator로 계산!!

• 수식트리 - 계산 프로그램.

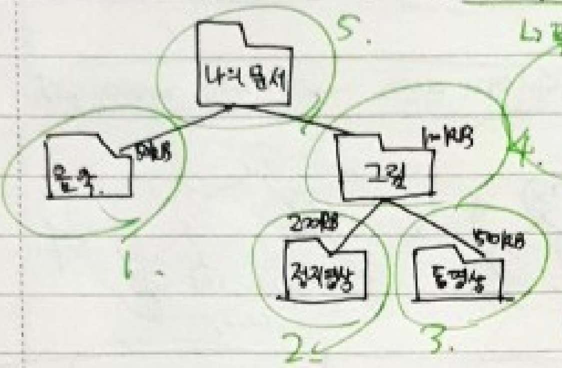
• 순환(recursive)호출을 이용한다. (계산)

```
int evaluate(TreeNode *root){
    if (root==NULL) return 0;
    if (root -> left ==NULL && root -> right)==NULL)   (단말)
        return root -> data;        ← 단말 = value.
    else {
        int op1 = evaluate (root -> left);   왼쪽
        int op2 = evaluate (root -> right);  오른쪽   → 이 순서에 의해 postorder 성립!!
        switch (root -> data){   연산자 → 비단말 = operator   (비단말)
            case '+' : return op1+op2;
            case '-' : return op1-op2;
            case '*' : return op1 * op2;
            case '/' : return op1 / op2;
```

p.236~237 8.8 트리의 응용: 디렉토리 용량 계산

• 디렉토리 용량 계산

□ 어떤 디렉토리의 총 용량을 계산하는데 후위(postorder)순회 사용.



└ 꼭 postorder로만 가능한 건 아닌데,
postorder가 가장 자연스럽다!
⇒ 용량 계산시, 가장 밑에서부터 올라오면서 계산한다.

(50) + (200 + 500) + (100) = 나의문서

p.287~290 8.9 이진 트리의 (추가)연산

□ 노드개수

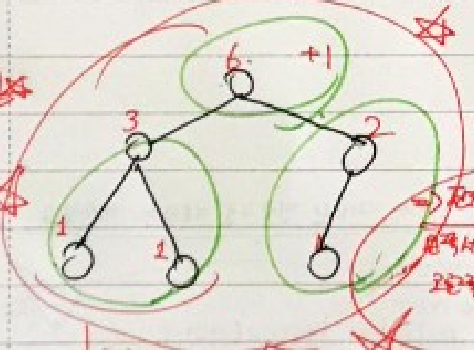
□ 높이(height)

• 트리 노드 개수

□ 이진 트리의 모든 노드의 개수를 계산.

□ 좌우 서브트리에 대하여 순환 호출(recursive call)하여 얻은 두 결과 값을 더한 후 1을 더 더함.

순환 호출



⇒ 전체트리의 노드개수 = 왼쪽서브트리노드개수 + 오른쪽서브트리노드개수 + 1

└ 3 + 2 + 1 = 6

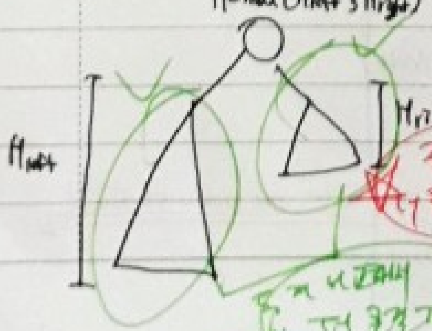
```
int get_node_count(TreeNode *node)
{
    int count = 0;
    if (node != NULL)
        count = 1 + get_node_count(node->left) +
            get_node_count(node->right);
    return count;
}
```

후위!! 왼쪽 └ 오른쪽

• 트리 높이(height)

□ 좌우 서브트리에 대하여 순환호출하여 얻은 두 결과 값 중 최대 값을 구한 후 1 증가.

H = max(Hleft, Hright)



작은거 X, 더 큰 거를 선택!!

둘 다 비교해서 더 큰걸 가짐!!!

순환호출하면 좌우 서브트리에 대한 높이 값이 나온다!!

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

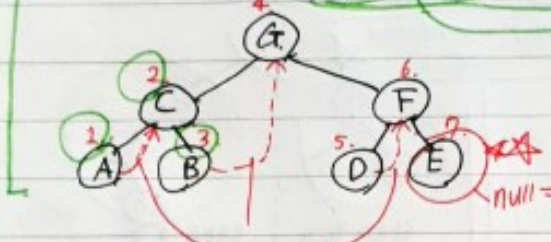
큰 값을 골라주는 max 함수

순환 호출

p.290~293 8.10 스레드 이진 트리.

• 스레드 (threaded) 이진 트리.

□ 이진 트리의 단말 노드의 오른쪽 링크는 NULL 인데, 여기에 중위(inorder) 순회시 후속 (successor) 노드의 주소를 저장하여 활용 (이 때 단말 노드의 오른쪽 링크가 스레드임을 추가로 표시함.



스레드 (threaded), 실

null ⇒ 중위순회 마지막 node

스레드 이진 트리에서는 오른쪽 link가 null이다.

(inorder의 다음이 없기 때문!!)

└ 오른쪽 노드 링크가 다음 순회의 노드인지, 오른쪽 자식 노드인지 구별하기 위해 필요하기 때문에 '스레드 (threaded)'가 실제로 필요한 것이다.

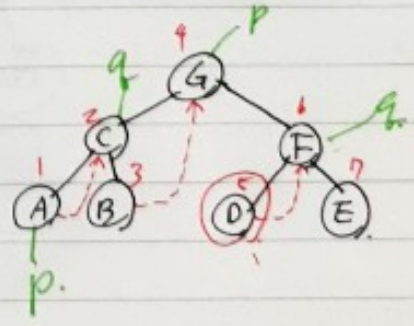
• 스레드 이진 트리의 노드 구조체.

□ 스레드 이진 트리에서의 노드 구조체는 오른쪽 링크가 스레드임을 추가로 표시할 필요가 있음.

```
typedef struct TreeNode {
    int data;
    struct TreeNode *left, *right;
    int is_thread;   // 오른쪽 링크가 스레드이면 1
} TreeNode;
```

• 스레드 이진 트리의 후속 노드

□ 스레드 이진 트리에서 중위(inorder) 후속 노드를 찾는 함수

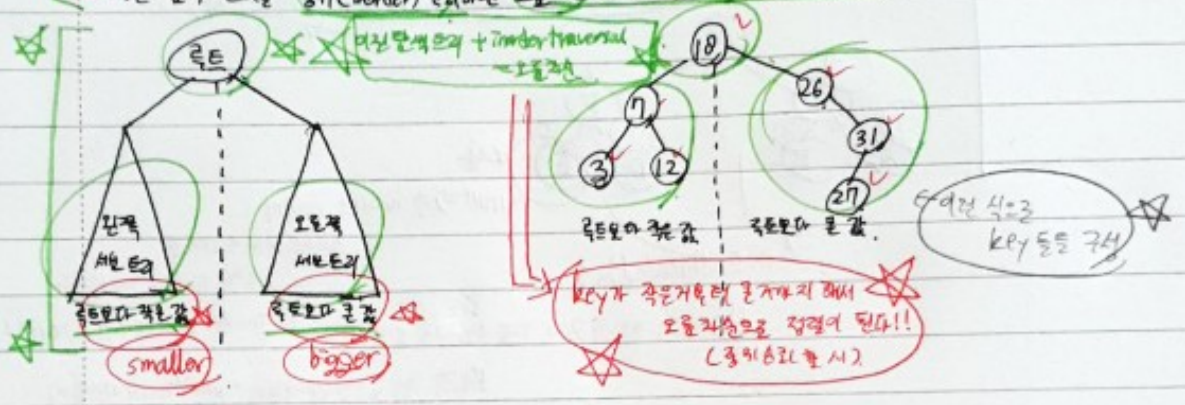



```
TreeNode *Find_successor(TreeNode *p)
{
    // q는 p의 오른쪽 링크
    TreeNode *q = p->right;
    // 오른쪽 링크가 스레드이거나
    // NULL이면 오른쪽 링크를 반환
    if (p->is_thread == TRUE || q == NULL)
        return q;
    // 그렇지 않으면 오른쪽 링크의
    // 가장 왼쪽 노드를 찾음
    while (q->left != NULL) q = q->left;
    return q;
}
```

p294~30 8.11 이진 탐색 트리.

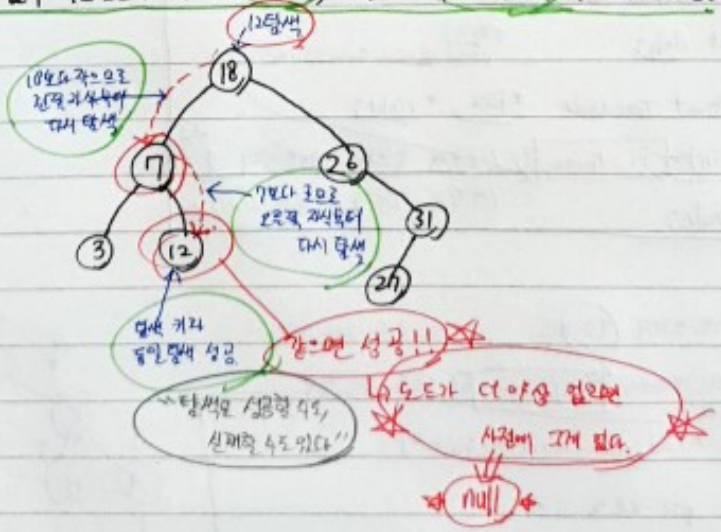
• 이진 탐색(binary search) 트리. search를 빨리하기 위한 것!!

- 탐색(search) 작업을 효율적으로 수행하기 위한 자료구조
- key(왼쪽 서브트리) ≤ key(루트노드) ≤ key(오른쪽 서브트리)
- 이진 탐색 트리를 중위(inorder) 순회하면 오름차순으로 정렬된 값을 얻을 수 있다.



• 이진 탐색 트리에서의 탐색

- 주어진 키 값이 루트 노드의 키 값과 같으면 탐색 성공.
- 주어진 키 값이 루트노드의 키값보다 작으면, 이 루트 노드의 왼쪽 자식에서 다시 탐색.
- 주어진 키 값이 루트 노드의 키 값보다 크면, 이 루트 노드의 오른쪽 자식에서 다시 탐색.



• 이진 탐색 트리에서의 탐색.

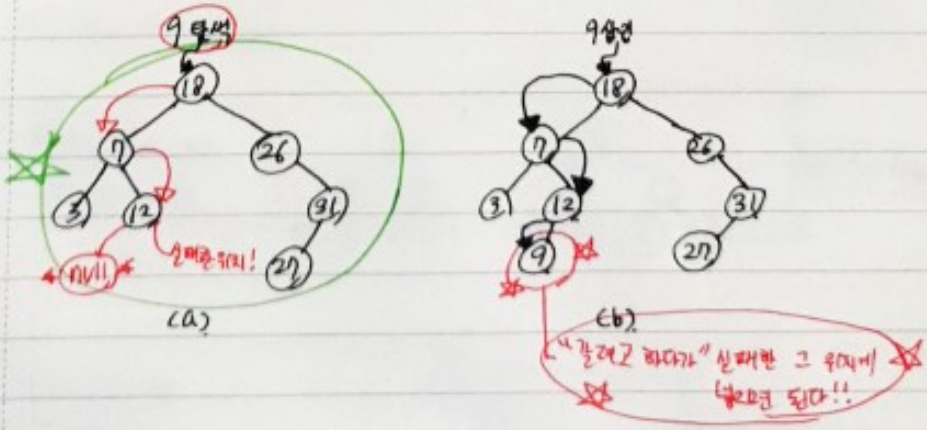
```
//순환적인(recursive) 탐색 함수
TreeNode *search(TreeNode *node, int key) {
    if (node == NULL) return NULL;          // 탐색 실패
    if (key == node->key) return node;      // 탐색 성공
    else if (key < node->key) return search(node->left, key);
    else return search(node->right, key);
}
```

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

• 이진 탐색 트리에서의 삽입.

□ 이진 탐색 트리에 원소(여기에서 9)를 삽입하기 위해서는 먼저 탐색을 수행하는 것이 필요.

□ 탐색에 실패한 위치(여기에서 9 탐색 시 12 위치에서 실패함)가 바로 새로운 노드를 삽입할 위치.



• 이진 탐색 트리에서의 삭제 ← 좀 복잡함.

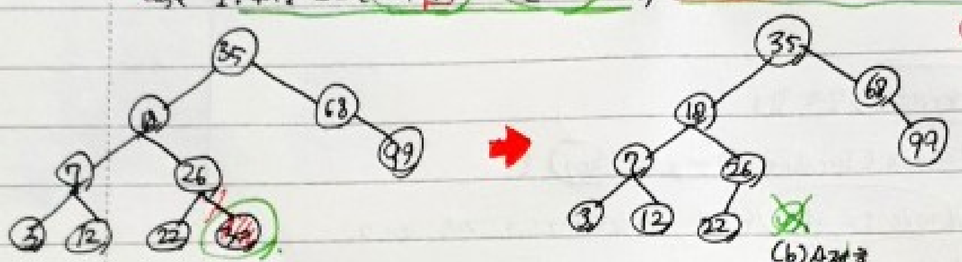
□ 아래 3가지 경우에 따라 다르게 처리함. 케이스 case 분리.

- Case 1: 삭제할 노드가 단말 노드인 경우 ⇒ 그냥 삭제, 바로 없애버리면 됨 (자식 노드 없음 X 때문)
- Case 2: 삭제할 노드가 하나의 서브트리를 가지고 있는 경우 ┐ 하나 vs 두 개.
- Case 3: 삭제할 노드가 두개의 서브트리를 가지고 있는 경우 ┘

• 이진 탐색 트리에서의 삭제 - case 1

□ Case 1: 삭제할 노드 (아래 30)가 단말 노드인 경우, 그 노드의 부모 노드를 찾아서 자식 연결을 끊으면 된다.
(terminal / Leaf 노드)

그냥 빼버리면 됨
조건 위반 X.

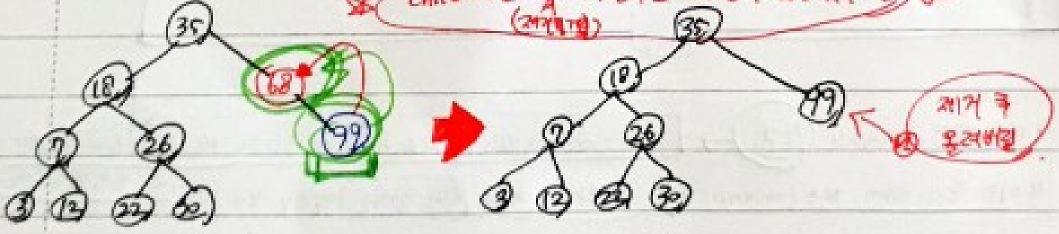


(a) 삭제 전 (b) 삭제 후

• 이진 탐색 트리에서의 삭제 - case 2.

□ Case 2: 삭제할 노드 (아래 68)가 하나의 서브트리를 가지고 있는 경우, 그 노드를 삭제하고 하나의 서브트리는 부모 노드에 붙여준다.

☆ Child 하나를 자식 빼면서 그 자리로 올려버리는 것이죠. ☆ (제거될 것)



(a) 삭제 전. (b) 삭제 후.

• 이진 탐색 트리에서의 삭제 - case 3.

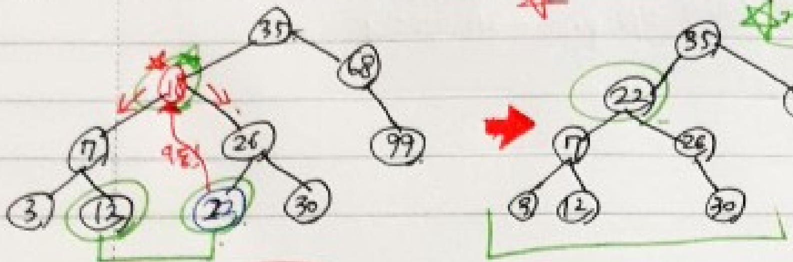
□ case 3: 삭제할 노드 (아래 18)가 두 개의 서브트리를 가지고 있는 경우,

가장 근접한 값

삭제할 노드와 가장 가까운 값을 가진 노드 (아래 22)를 삭제할 노드 위치로 이동한다.

□ 가장 가까운 값은 왼쪽 서브트리의 가장 큰 값(아래 12)과 오른쪽 서브트리의 가장 작은 값(아래 22) 중 하나임.

가장 차이가 작은 순서를 데려온다!!



왼쪽/오른쪽 정할 수 있다!!

왼쪽, 오른쪽 둘 다 가능하다!!

왼쪽에서 가장 큰 놈 vs 오른쪽에서 가장 작은 놈

• 이진 탐색 트리의 성능 분석

□ 이진 탐색 트리에서의 탐색, 삽입, 삭제 연산의 시간 복잡도(time complexity)는 트리의 높이(height) h에 비례한다.

□ 최선(best)의 경우 → 성능 가장 굿!

■ 이진 탐색 트리가 포화 이진 트리일 경우

■ $h = \log_2 n$ (여기서 n은 전체 노드 수) → 컴퓨터과학에서 성능 최고는 $\log_2$ 이다!!

□ 최악(worst)의 경우 → 성능 가장 쓰레기

■ 이진 탐색 트리가 경사 이진 트리일 경우

■ $h = n$ (여기서 n은 전체 노드 수).

→ 두가지 경우에 따라 성능이 다르다는 걸 우리는 알고 있어야 된다.

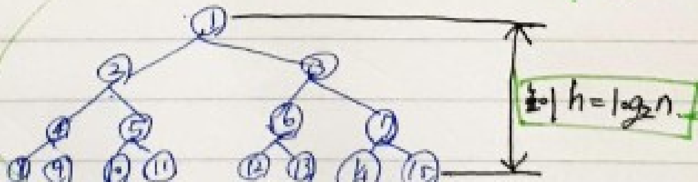
성능 좋은 트리 만드는 데 필요한 기초 이론들임

⇓

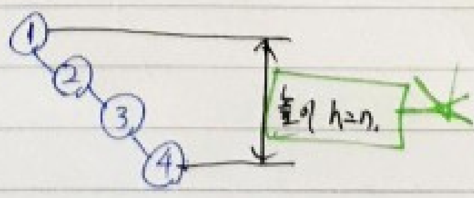
이진 트리는 성능이 좋은 것처럼 나올 때도 있어야 된다

• 이진 탐색 트리의 성능 분석.

최선(best)의 경우
(포화 이진 트리).



최악(worst)의 경우
(경사 이진 트리)



p.310~315 8.12 이진탐색 트리의 응용: 영어 사전.

- 사전 구축하는 데 도움되게 쓰는 이진 탐색 트리

☆ 삽입 / 탐색 / 삭제가 가장 중요 ☆

↳ 하나씩 반복하면서 단어단상 후 탐색 ☆

(자세한 것은 p.310~315 확인!!)