CHAP. 8
트리(tree)

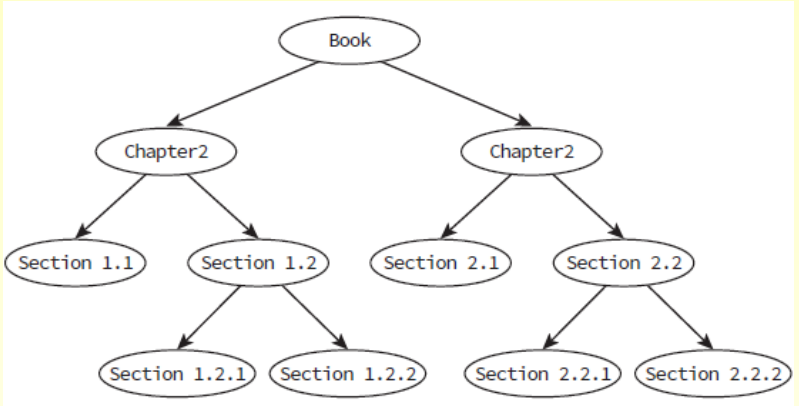
지금까지 학습했던 자료구조 유형

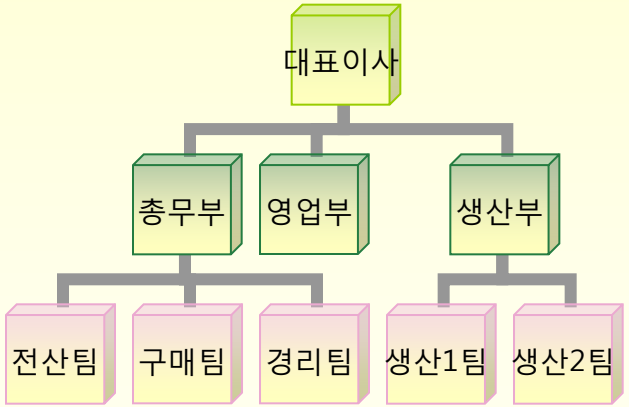
- 개별형 : 일반 변수에 1개 데이터만 저장
- 그룹형
  - 배열 : 같은 종류의 데이터들을 하나의 이름으로 저장
    - 검색 및 기타 연산은 "배열변수[첨자], 예: arr[8]" 이용하여 수행
  - 구조체 : 서로 다른 종류의 데이터들을 묶어서 하나의 이름으로 저장
    - 검색 및 기타 연산은 "구조체변수.필드이름, 예: inf.name" 이용하여 수행
  - 리스트 : 연속된 데이터들을 저장하는 방법으로 배열 또는 연결리스트로 구현
    - 저장된 결과는 순서를 가지고 있음
  - 스택 : LIFO 방식의 리스트 형식(배열 또는 연결리스트로 구현)
  - 큐 : FIFO 방식의 리스트 형식(배열 또는 연결리스트로 구현)

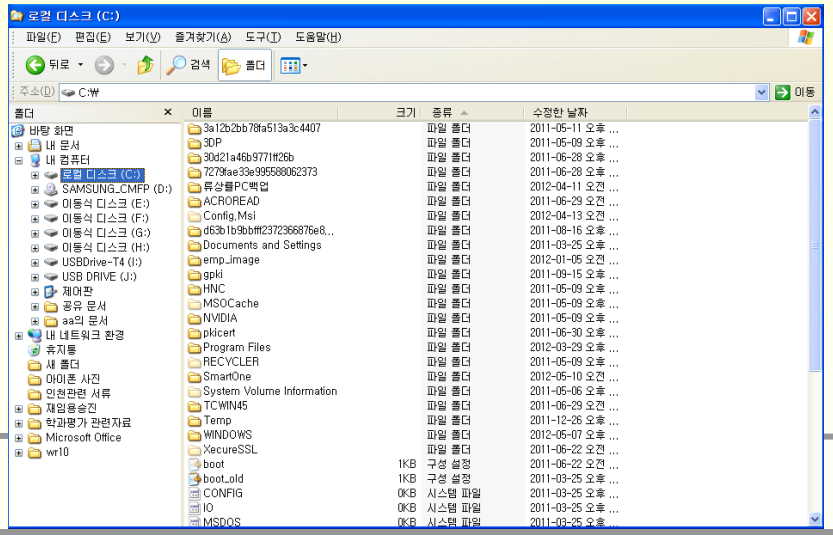
- 추가될 자료구조 : 트리(Tree) & 우선순위 큐(priority-Queue)

트리(TREE)

- 트리 : 계층적인 구조를 나타내는 자료구조(2차원적 시각의 구조)
- 트리 구성 : 부모-자식 관계의 노드 순서로 이루어진다.
- 응용분야
  - 계층적인 조직 표현
  - 파일 시스템
  - 인공지능에서의 결정트리

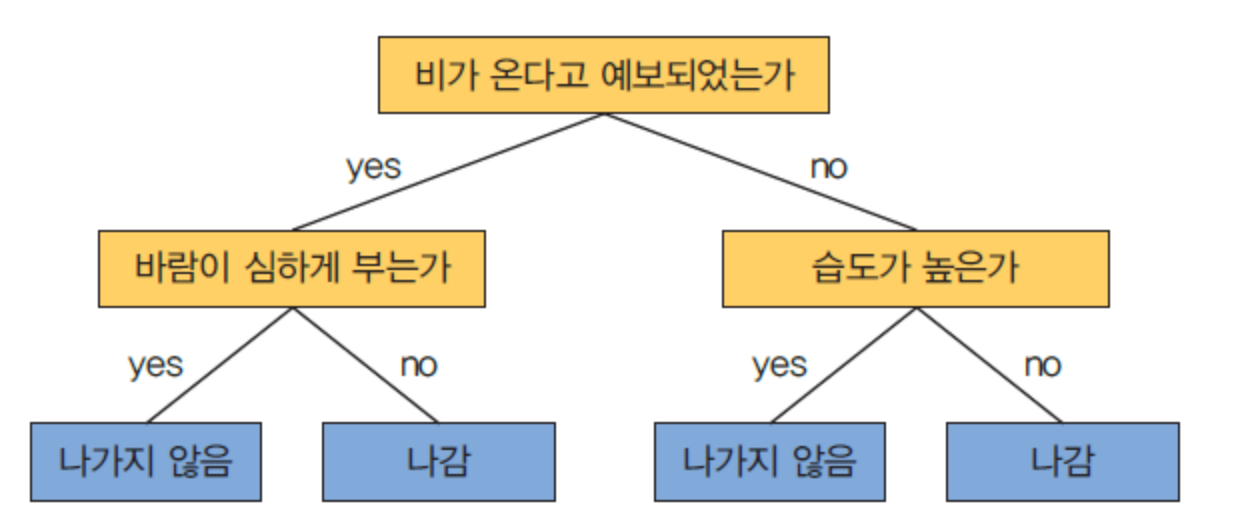




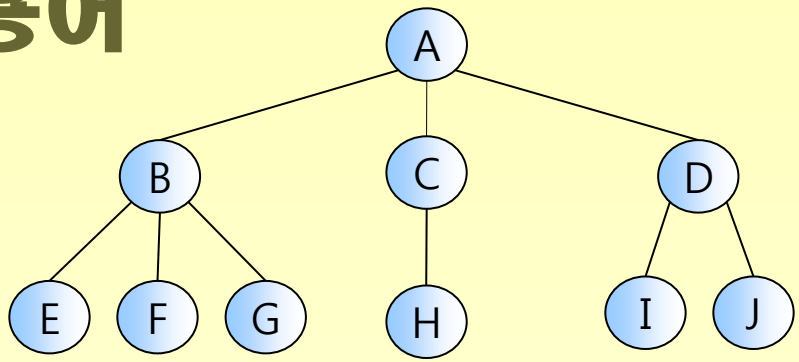


결정 트리(Dicision Tree)

- 결정 트리 : 판단(if문)을 하고 이에 대한 결정(then/else)을 하는 구조



주요 용어

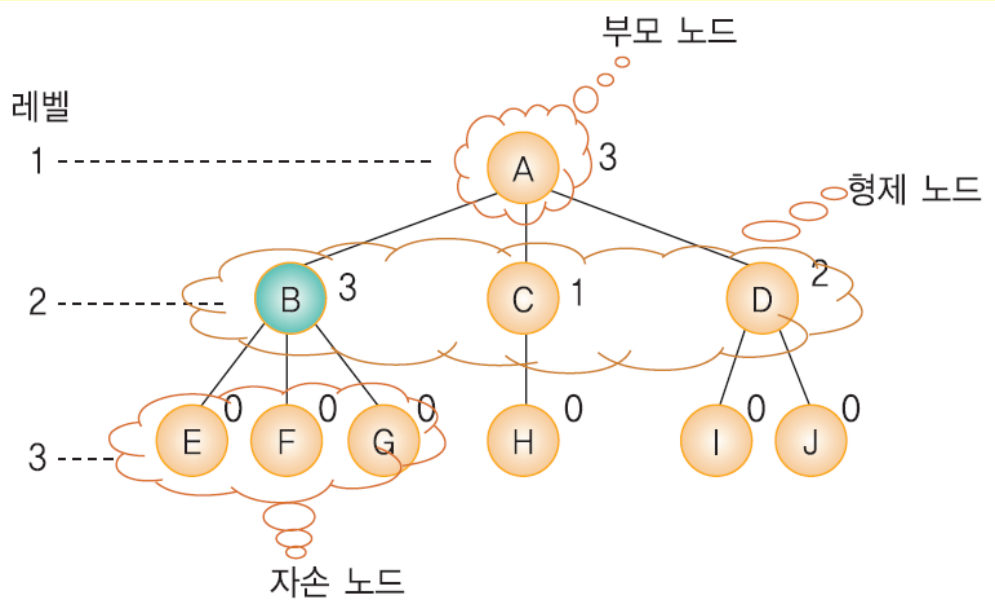


- 노드(node): 트리의 구성요소
- 루트(root): 부모가 없는 노드(A)
- 서브트리(subtree): 하나의 노드와 그 노드들의 자손들로 이루어진 트리

- 단말노드(terminal node): 자식이 없는 노드(E,F,G,H,I,J)
- 비 단말노드: 적어도 하나의 자식을 가지는 내부 노드(A,B,C,D)

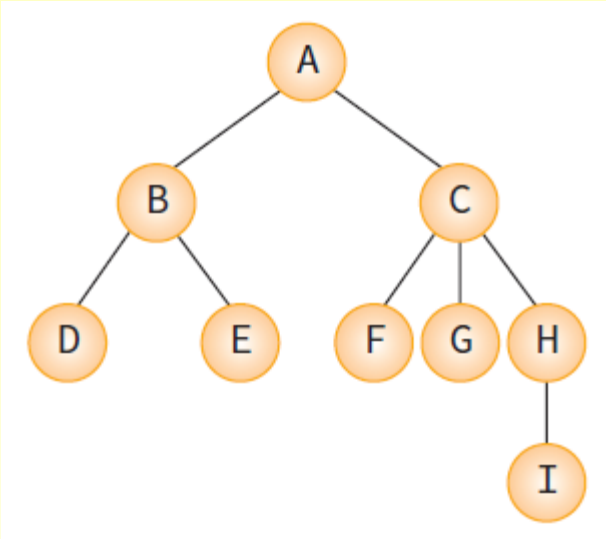
※ 자식, 부모, 형제, 조상, 자손 노드 : 인간 관계와 비슷한 개념

주요 용어



- **레벨(level)**: 트리의 각 레벨 번호(1~3)
- **높이(height)**: 트리의 최대 레벨(3)
- **차수(degree)**: 노드가 가지고 있는 자식 노드의 개수(0~3)

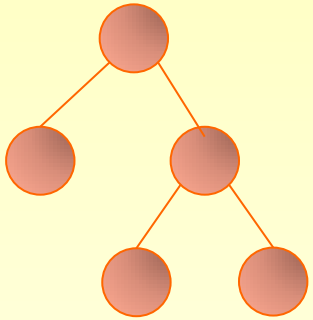
주요 용어

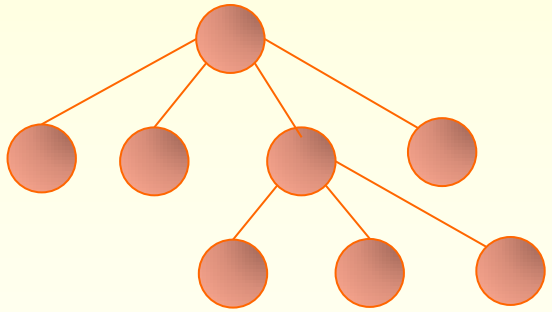


- A : 루트(root) 노드
- B : D와 E의 부모(parent) 노드
- C : B의 형제(sibling) 노드
- D와 E : B의 자식(child) 노드
- B의 차수(degree) = 2
- 이 트리의 높이(height) = 4

트리 종류

- 트리
  - 이진 트리
  - 일반 트리

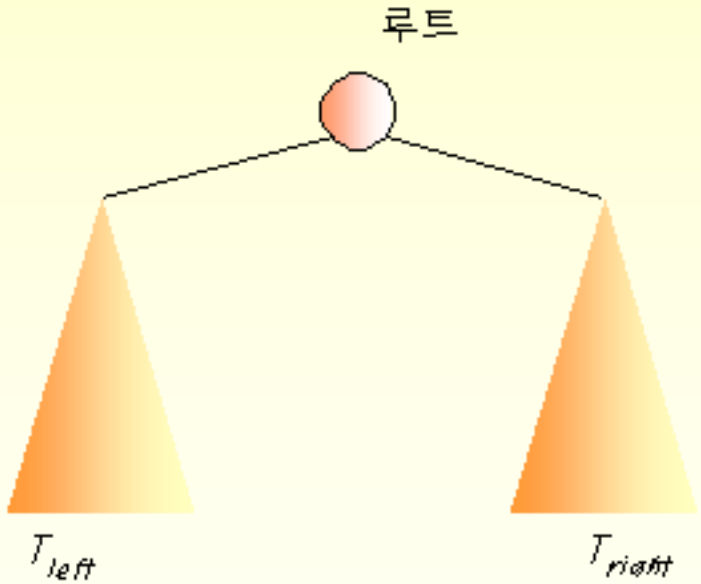


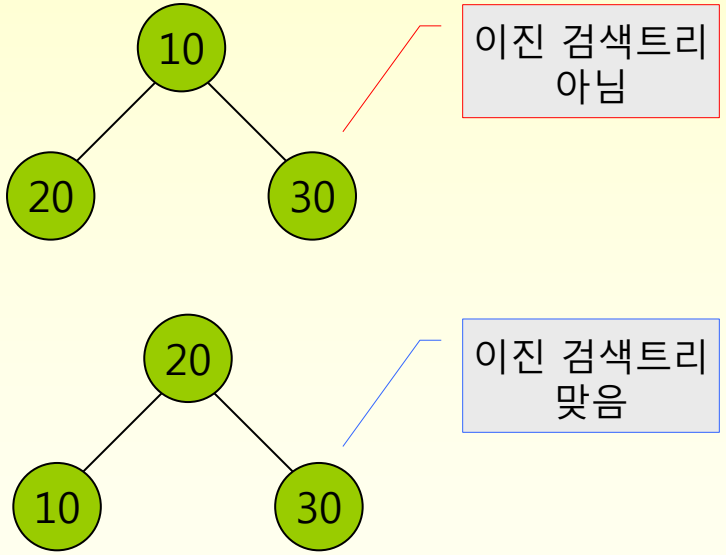


이진트리(binary tree)

- 이진 트리(binary tree) :모든 노드가 2개의 서브 트리를 가지고 있는 트리
  - 서브트리는 자식 노드가 없는 공집합도 가능
- 이진트리에서 각 노드에 연결된 노드 개수 : 최대 2개의 자식 노드

  ➔ 모든 노드의 차수가 2 이하로 구성 ➔ 구현하기 편리(if 문 적용)
- 이진 검색트리는 서브트리의 크기 순서에 의해 좌우로 배치





이진트리 생성과정

- 입력 : E C H F B K A D J

- 입력 : A B C D E F H J K

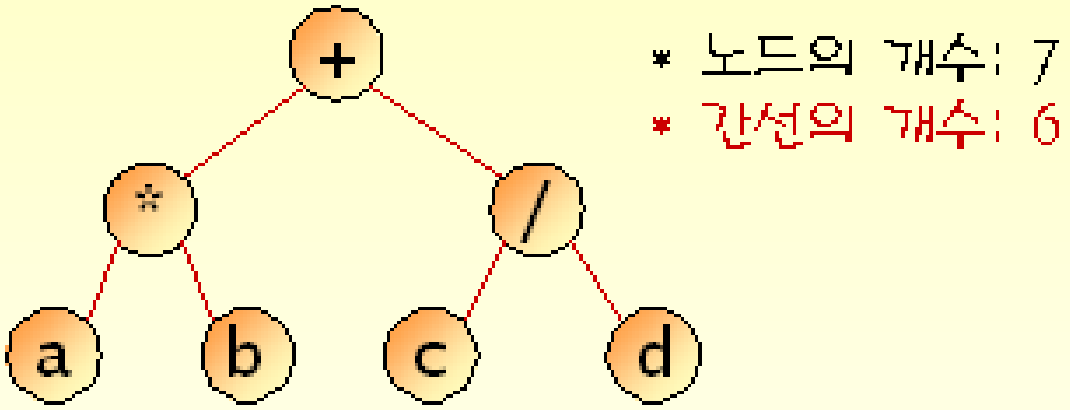
이진 검색트리 생성과정

- 입력 : E C H F B K A D J

- 입력 : A B C D E F H J K

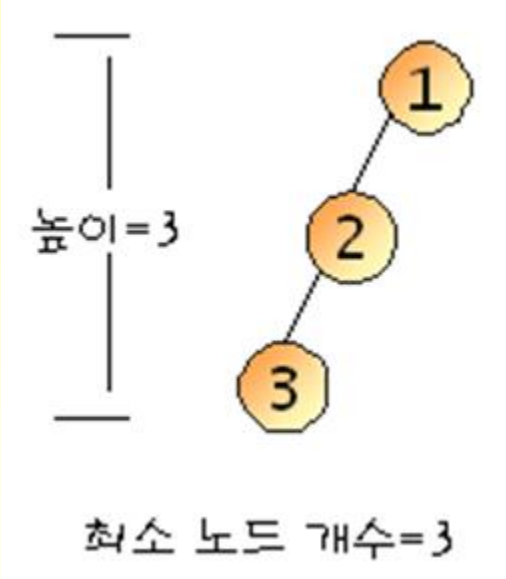
이진트리의 성질

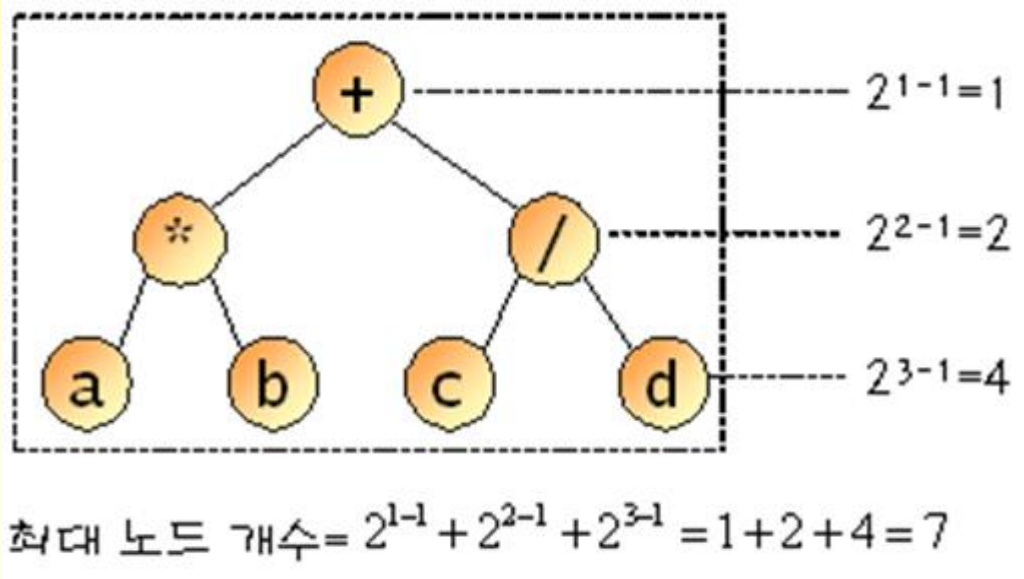
- 노드 개수가 n개 ➔ 간선(edge) 개수는 n-1



이진트리의 성질

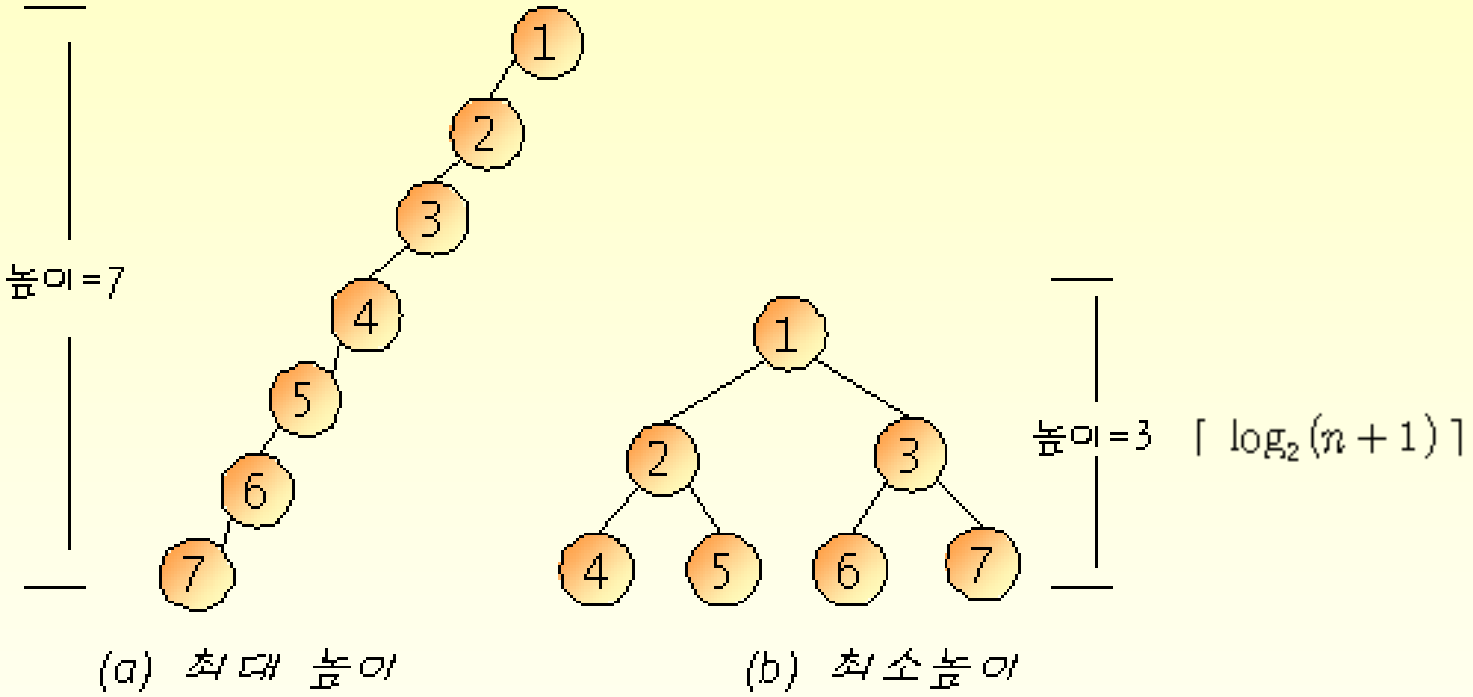
- 높이가 h인 이진트리 : 최소 h개 ~ 최대 2^h-1개 노드를 가진다.





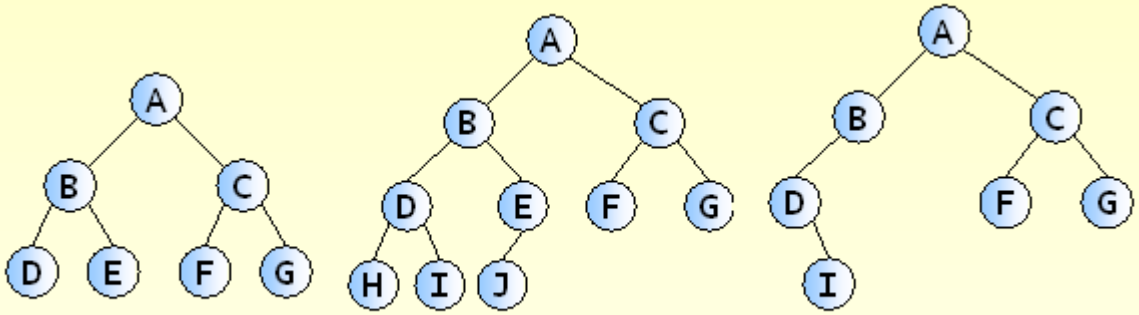
이진트리의 성질

- n개의 노드를 가지는 이진트리의 높이는 최대 n ~ 최소 $(\log_2 n)+1$



이진트리의 분류

- **포화 이진 트리(full binary tree)**: 트리의 각 레벨에 노드가 꽉 차있는 형태
- **완전 이진 트리(complete binary tree)**: 높이가 h일 때 레벨 1부터 h까지는 노드가 모두 채워져 있고, 마지막 레벨 h에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진트리



(a) 포화이진트리 (b) 완전이진트리 (c) 기타이진트리

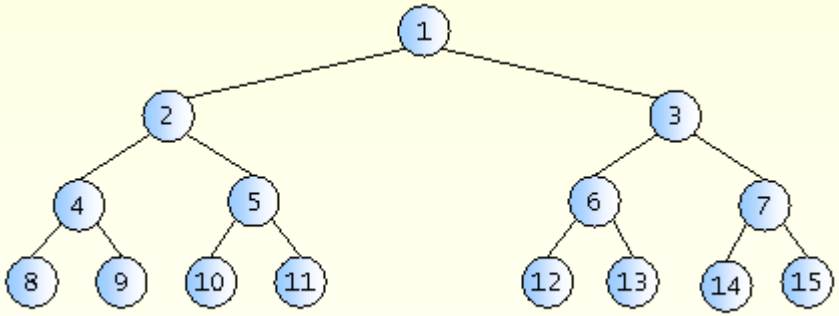
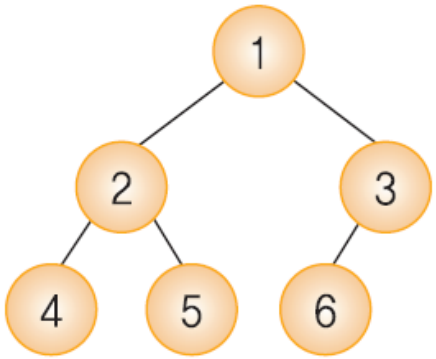


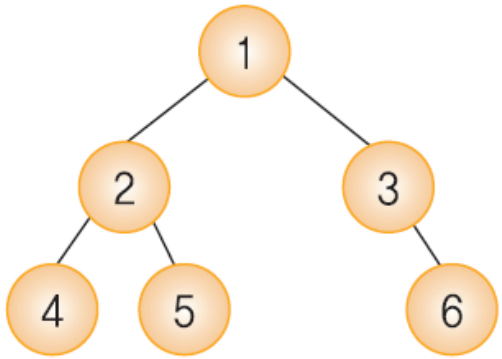
그림 7.15 포화이진트리에서의 노드의 번호

이진트리의 분류

- 완전 이진 트리(complete binary tree) ⊂ 포화 이진 트리(full binary tree)



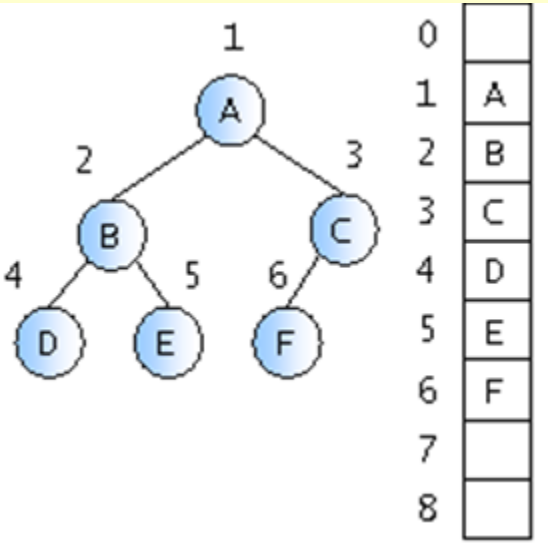
(a) 완전 이진 트리

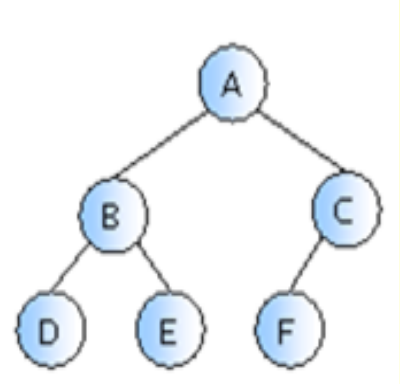


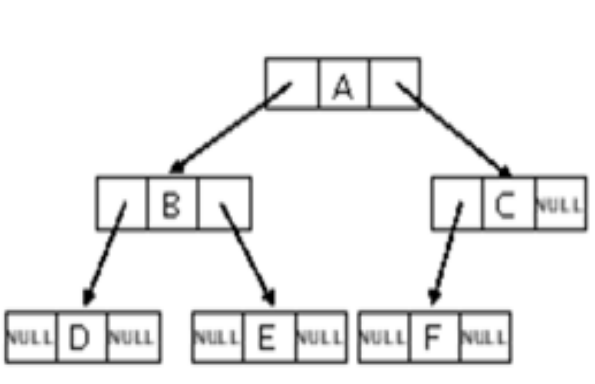
(b) 완전 이진 트리가 아님

이진트리를 구현하는 방법

- 배열을 이용하는 방법
- 구조체(포인터)를 이용하는 방법

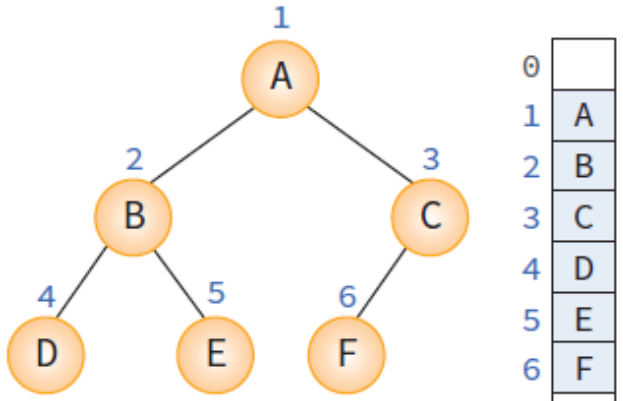




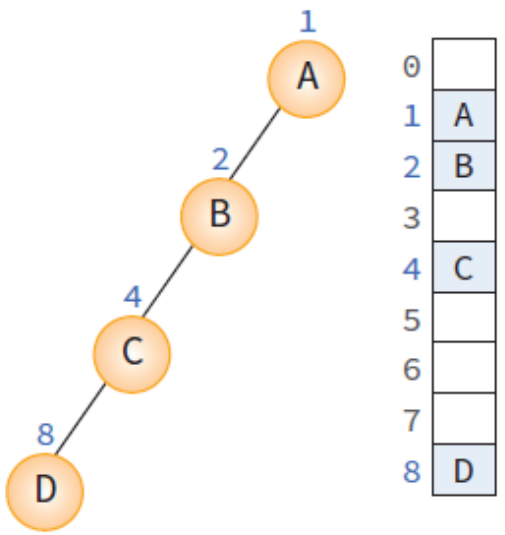


이진트리를 구현하는 방법

- **배열**로 표현
  - 모든 이진트리를 포화 이진트리라고 가정하고, 각 노드에 번호를 붙여서 그 번호를 배열의 첨자로 적용하여 노드를 배열요소에 저장하는 방법



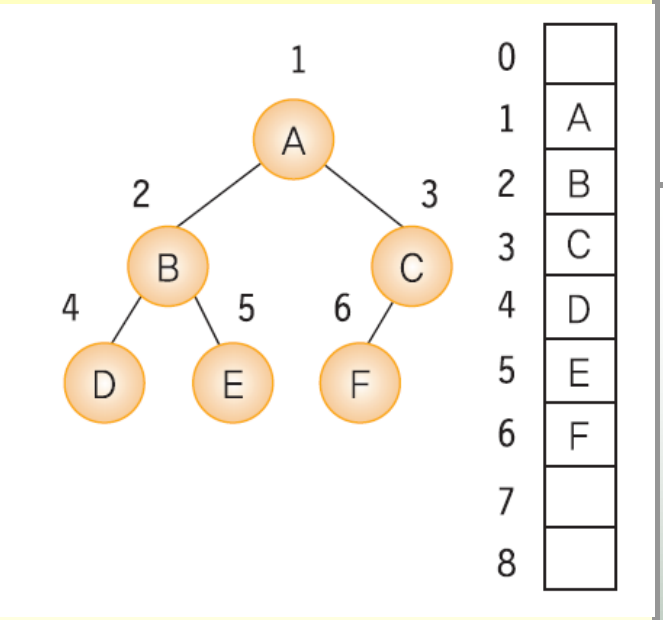
(a) 완전 이진트리



(b) 경사 이진트리

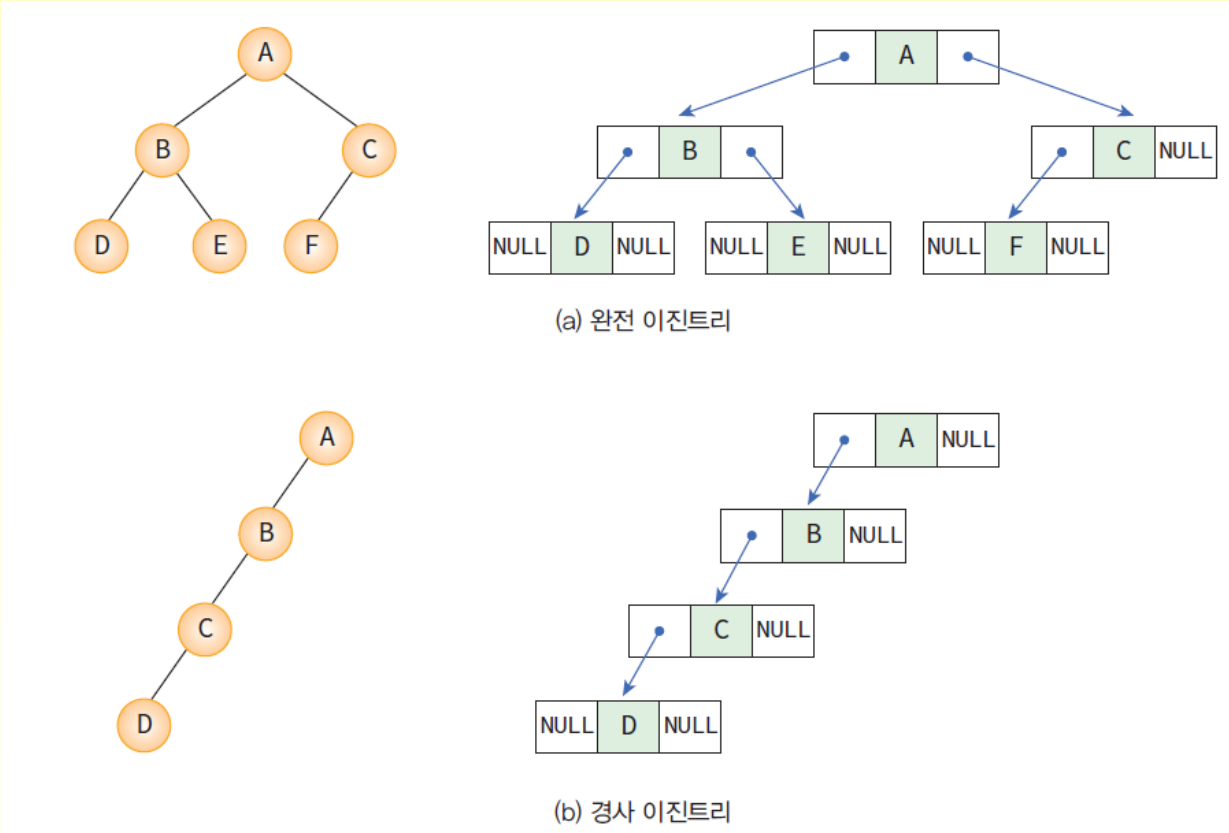
이진트리를 구현하는 방법

- **배열로 표현**
  - 루트는 무조건 1번 인덱스에 저장
  - 노드 i의 부모노드의 인덱스 = i/2
  - 노드 i의 왼쪽 자식노드의 인덱스 = 2*i
  - 노드 i의 오른쪽 자식노드의 인덱스 = 2*i+1



이진트리를 구현하는 방법

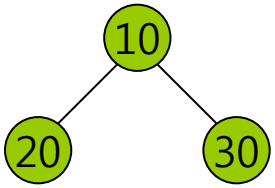
- 연결리스트로 표현 : 2개 포인터를 가진 노드 구조
  - 부모 노드가 왼쪽/오른쪽 자식 노드의 주소를 가지는 방법



이진 트리(검색트리 X)의 표현 by 연결리스트

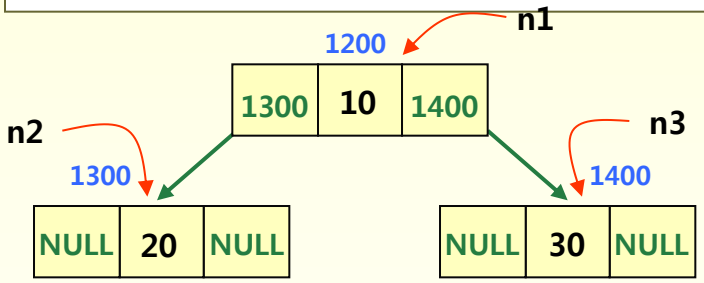
```
#include <stdio.h>
#include <stdlib.h>
typedef struct TreeNode {
    int       data;
    struct  TreeNode  *llink, *rlink;
} TreeNode;

void main()
{
   TreeNode  *n1, *n2, *n3;
   n1 = (TreeNode *)malloc(sizeof(TreeNode));
   n2 = (TreeNode *)malloc(sizeof(TreeNode));
   n3 = (TreeNode *)malloc(sizeof(TreeNode));
```

트리 모양



주소


```
   n1->data = 10;
   n1->llink = n2;
   n1->rlink = n3;

   n2->data = 20;
   n2->llink = NULL;
   n2->rlink = NULL;

   n3->data = 30;
   n3->llink = NULL;
   n3->rlink = NULL; // 일방적 삽입
}
```



이진 트리의 검색/출력 순서 정하기

- 이진트리 검색(retrieval) : 루트의 주소만 아는 상태에서 검색값 노드 찾기
  - 검색값 입력 ➔ 루트에서 출발 ➔ 서브트리들 검색 ➔ 있으면 주소, 없으면 NULL

- 이진트리 출력 : 루트 주소가 주어지면, 서브트리의 모든 노드를 출력(검색)
  - 레벨 출력(검색) : 20-10-30-8-17-25-50
  - 순회(traversal) 출력(검색)

    전위 순회(pre-order traversal) : PLR

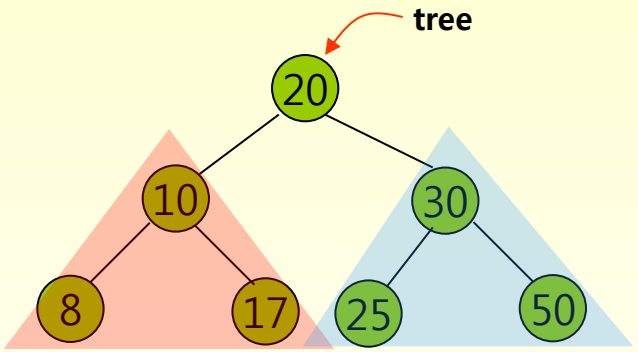
    20-10-8-17-30-25-50

    중위 순회(in-order traversal) : LPR

    8-10-17-20-25-30-50

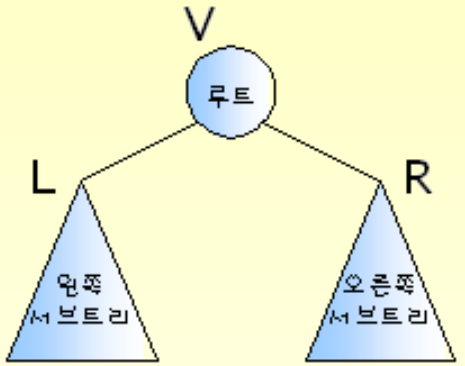
    후위 순회(post-order traversal) : LRP

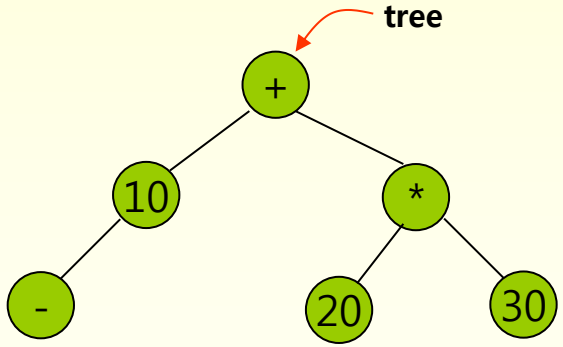
    8-17-10-25-50-30-20



이진트리의 순회

- 순회(traversal) : 트리의 모든 노드들을 순서적으로 방문하는 동작
- 3가지의 기본적인 순회방법
  - 전위 순회(pre-order traversal) : VLR
    - 자식 노드보다 부모 노드를 먼저 방문
  - 중위 순회(in-order traversal) : LVR
    - 왼쪽 자식, 부모, 오른쪽 자식 순으로 방문
  - 후위 순회(post-order traversal) : LRV
    - 부모 노드보다 자식 노드를 먼저 방문

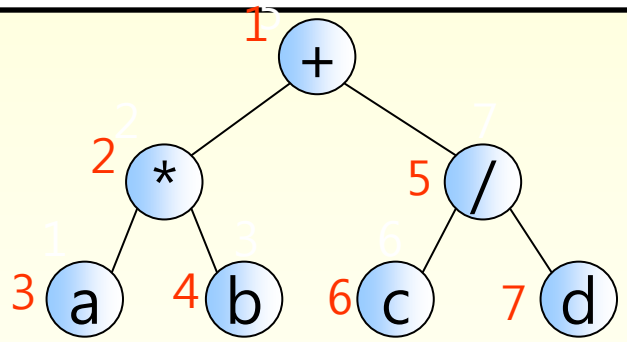




전위순회(순환법 적용)

- 노드 방문 순서 : 루트 -> 왼쪽 서브트리 -> 오른쪽 서브트리

```
preorder( TreeNode  *root )
{
    if ( root ){
      printf("%d", root->data );     // 부모 노드 방문
      preorder( root->left );        // 왼쪽 서브트리 순회를 순환적으로
      preorder( root->right );       // 오른쪽 서브트리 순회를 순환적으로
    }
}
```

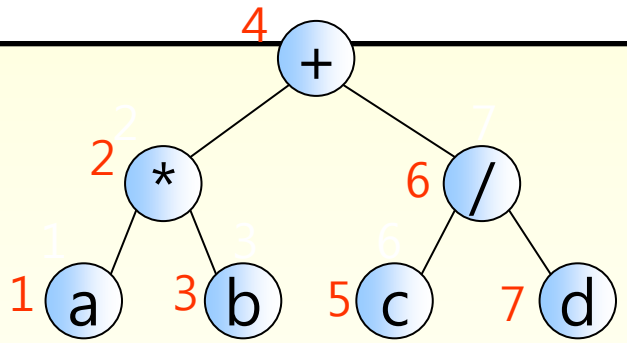


입력 수식 : a*b+c/d
전위순회 결과 : +*ab/cd

중위순회(순환법 적용)

- 노드 방문 순서 : 왼쪽 서브트리 -> 루트 -> 오른쪽 서브트리

```
inorder( TreeNode  *root ) //입력 순서 또는 크기 순서대로 저장/검색할 때 사용
{
    if ( root ){
      inorder( root->left );        // 왼쪽 서브트리를 순환적으로
      printf("%d", root->data );  // 부모 노드 방문
      inorder( root->right );      // 오른쪽 서브트리를 순환적으로
    }
}
```



입력 수식 : a*b+c/d
중위순회 결과 : a*b+c/d

중위순회(반복법 사용하면?)

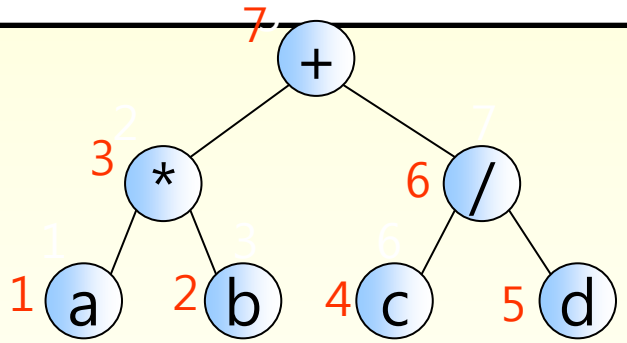
```
int top = -1;  TreeNode *stack[100];  //스택 사용
void push(TreeNode *p) {
    if (top < SIZE - 1) stack[++top] = p;
}
TreeNode *pop() {
    TreeNode *p = NULL;
    if (top >= 0) p = stack[top--];
    return p;
}
void inorder_iter(TreeNode *root) {
  while (1) {
      for (; root; root = root->left)  push(root);
      root = pop();
      if (!root) break;
      printf("[%d] ", root->data);
      root = root->right;
  }
}
```

```
inorder( TreeNode  *root )
{
    if ( root != NULL ) {
        inorder( root->left );
        printf("%d", root->data );
        inorder( root->right );
    }
}
```

후위순회(순환법 적용)

- 노드 방문 순서 : 왼쪽 서브트리 -> 오른쪽 서브트리 -> 루트

```
postorder( TreeNode  *root )  //수식을 후위표기식으로 변환할 때 응용
{
    if ( root ){
            postorder( root->left );     // 왼쪽 서브트리 순회를 순환적으로
            postorder( root->right );    // 오른쪽 서브트리순회를 순환적으로
            printf("%d", root->data );   // 부모 노드 방문
    }
}
```

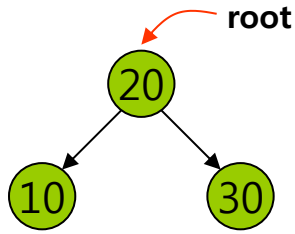


입력 수식 : a*b+c/d
후위순회 결과 : ab*cd/+

이진트리 순회방법

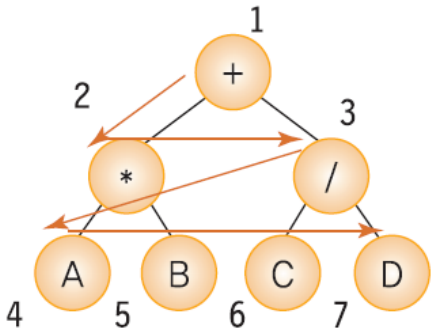
```
typedef struct tnode {
   int data;
   struct tree_node  *left_child, *right_child;
} tnode;
tnode *root;
inorder( tnode *ptr ){
    if ( ptr ){
      inorder( ptr->left_child );   // 좌측 검사
      printf("%d ", ptr->data );    // 노드 방문
      inorder( ptr->right_child );  // 우측 검사
    }
}
preorder( tnode *ptr ){
    if ( ptr ){
       printf("%d ", ptr->data );     // 노드 방문
       preorder( ptr->left_child );   // 좌측 검사
       preorder( ptr->right_child );  // 우측 검사
    }
}
postorder( tnode *ptr ){
    if ( ptr ){
      postorder( ptr->left_child );   // 좌측 검사
      postorder( ptr->right_child );  // 우측 검사
      printf("%d ", ptr->data );      // 노드 방문
    }
}
```

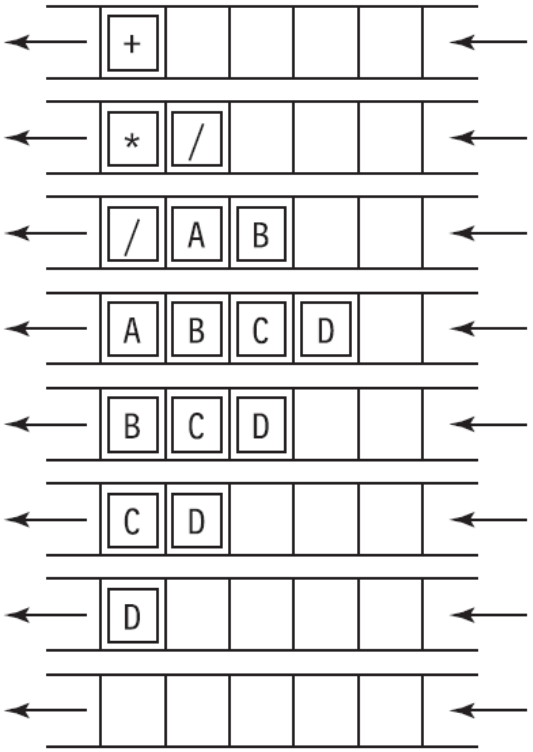
```
void main() //이진(검색X)트리 구성 후, 순회
{
   tnode *n1, *n2, *n3;
   n1 = (tnode *)malloc(sizeof(tnode));
   n2 = (tnode *)malloc(sizeof(tnode));
   n3 = (tnode *)malloc(sizeof(tnode));

   n1->data = 20;
   n1->left_child = n2;
   n1->right_child = n3;

   n2->data = 10;
   n2->left_child = NULL;
   n2->right_child = NULL;

   n3->data = 30;
   n3->left_child = NULL;
   n3->right_child = NULL;

   root = n1; //루트 지정
   printf("₩n 중위순회 결과 : ");
   inorder(root);
   printf("₩n 전위순회 결과 : ");
   preorder(root);
   printf("₩n 후위순회 결과 : ");
   postorder(root);
}
```



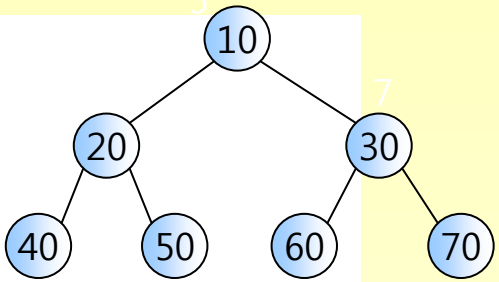
또다른 순회 : 레벨 순회

- 각 노드를 레벨 순으로 검사하는 순회 방법
- 앞의 순회 방법들이 순환법으로 스택 사용했던 것에 비해, 레벨 순회는 큐를 사용





또다른 순회 : 레벨 순회

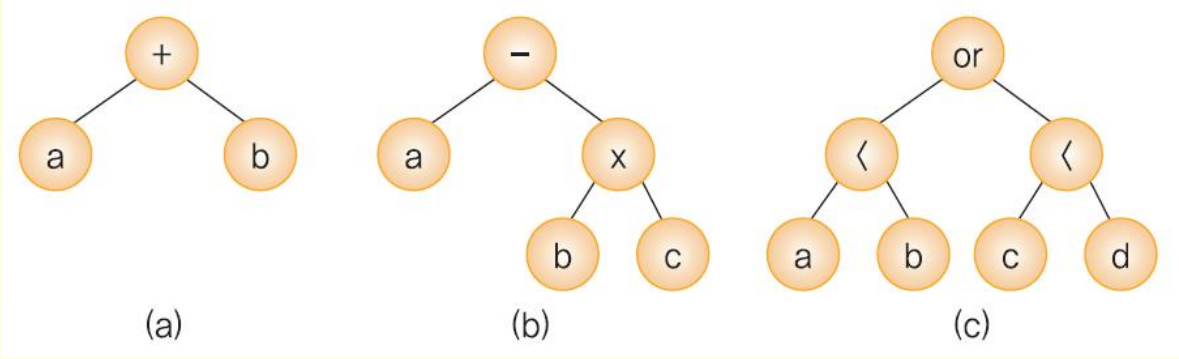


queue

```
// Queue 선언이 되어있다고 가정
… …
void  level_order(tnode *root)
{
    queue 초기화;
    tnode *p = root;
    if (p == NULL) return;
    enqueue(queue, root); //queue에 root 노드부터 삽입
    printf("₩n 레벨 순회 결과 : ");
    while (! isempty(queue)) //queue에 노드가 있을동안 반복
    {
        p← dequeue(queue);
        printf(" %d ", p->data);
        if (p->llink != NULL)  enqueue(queue, p->llink);
        if (p->rlink != NULL)  enqueue(queue, p->rlink);
    }
}
```

수식 트리

- 수식 트리 : 산술식을 이진트리로 표현한 구조
  - 비 단말노드 : 연산자(operator)
  - 단말노드 : 피연산자(operand)
- 예

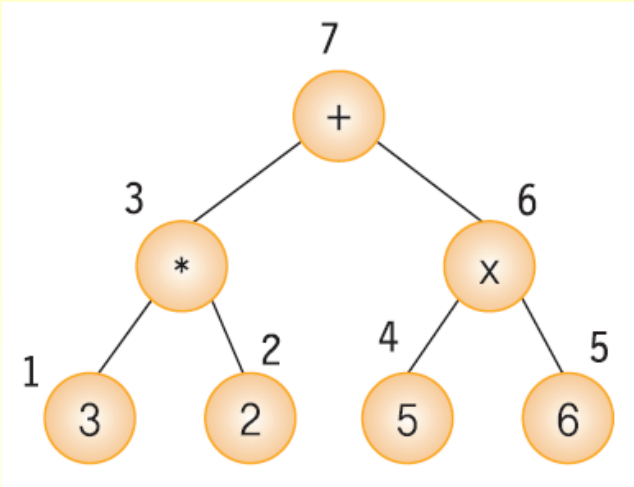


수식	a + b	a - (b × c)	(a < b) || (c < d)
전위순회	+ a b	- a × b c	|| < a b < c d
중위순회	a + b	a - b × c	a < b || c < d
후위순회	a b +	a b c × -	a b < c d < ||

수식 트리 계산

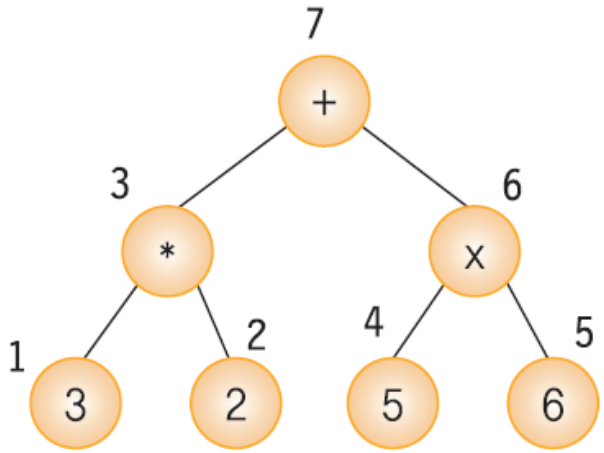
- 후위순회를 사용
  - 서브트리의 값을 순환호출로 계산
  - 비단말노드를 만나면 양쪽 서브트리의 값은 부모노드의 연산자를 이용하여 계산

```
evaluate(exp)
{
  if (exp = NULL)
     then  return 0;
     else  { x = evaluate(exp->left);   // 왼쪽
             y = evaluate(exp->right); // 오른쪽
             op = exp->data;           //중간
             return (x op y);
           }
}
```



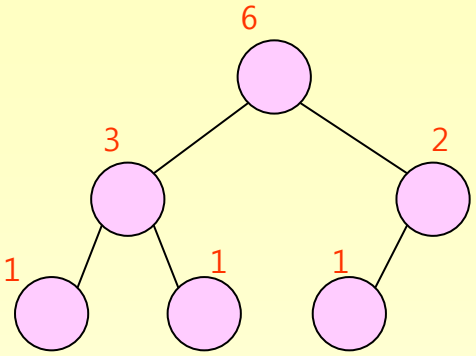
수식트리 계산의 간단 코드

```
int  evaluate(TreeNode *root) {
    if (root == NULL) return 0;
    if (root->llink == NULL && root->rlink == NULL)  return root->data;
    else {
          int op1 = evaluate(root->llink);    int op2 = evaluate(root->rlink);
          printf("₩n %d %c %d 계산 ", op1, root->data, op2); //중간 계산과정 출력
          switch (root->data) {
             case '+': return op1 + op2;
             case '-':  return op1 - op2;
             case '*':  return op1 * op2;
             case '/':  return op1 / op2;
          }
    }
}
void main( )
{
    //이진트리 구성문장들
    … …
    printf("수식 계산 결과 = %d  ₩n", evaluate(exp));
}
```



이진트리 연산 : 노드 개수 계산

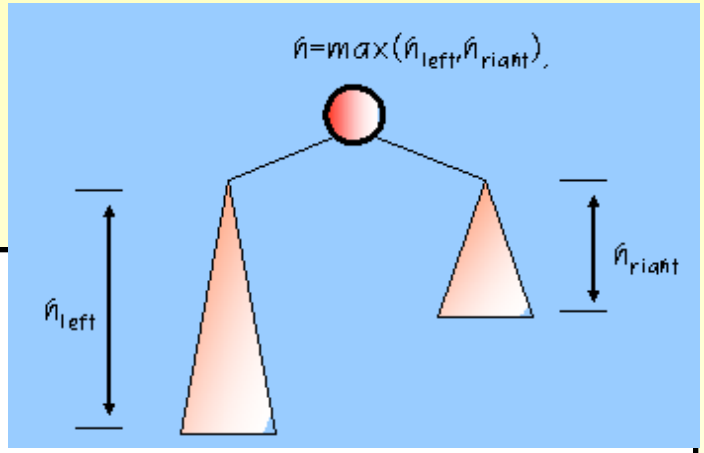
- 이진 트리에 소속된 노드 개수를 모두 계산
- 각각의 서브트리에 대하여 순환 호출하여 각 개수를 계산한 후, 반환되는 값에 1을 더하여 최종적으로 전체 개수를 반환




```
int get_node_count(TreeNode *p)
{
      int  count = 0;
      if ( p != NULL )
          count = 1 + get_node_count(p->llink) + get_node_count(p->rlink);
      return count; //빈 트리이면 0, 아니면 트리의 노드개수 리턴
}
```

이진트리 연산 : 높이(height) 계산

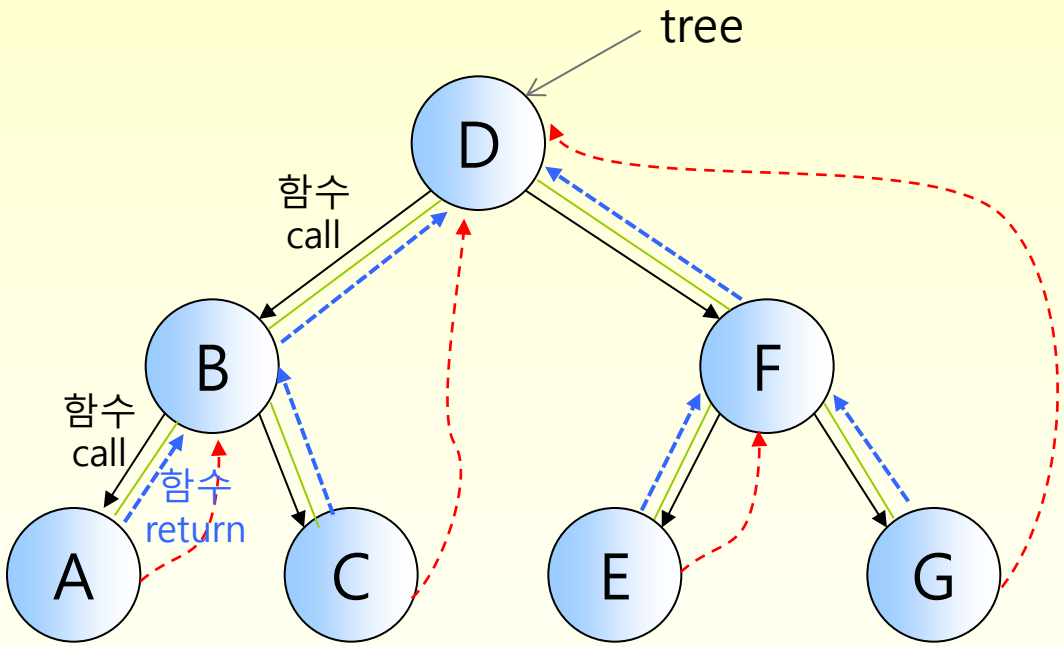
- 서브트리에 대하여 차례대로 순환 호출하고 서브 트리들의 반환값 중에서 최대값을 구하여 반환




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

스레드(threaded) 이진트리

- 트리에서 순회할 때, 부모 노드에서 자식 노드로 이동하려면 llink, rlink 이용한 함수 call & 부모 노드로 되돌아오려면 함수 return 방식을 사용
- 단말 노드는 2개의 링크가 모두 NULL이므로, 낭비되는 것으로 판단됨 ➔ 부모 노드로 되돌아가는 링크로 활용하면 어떨까?

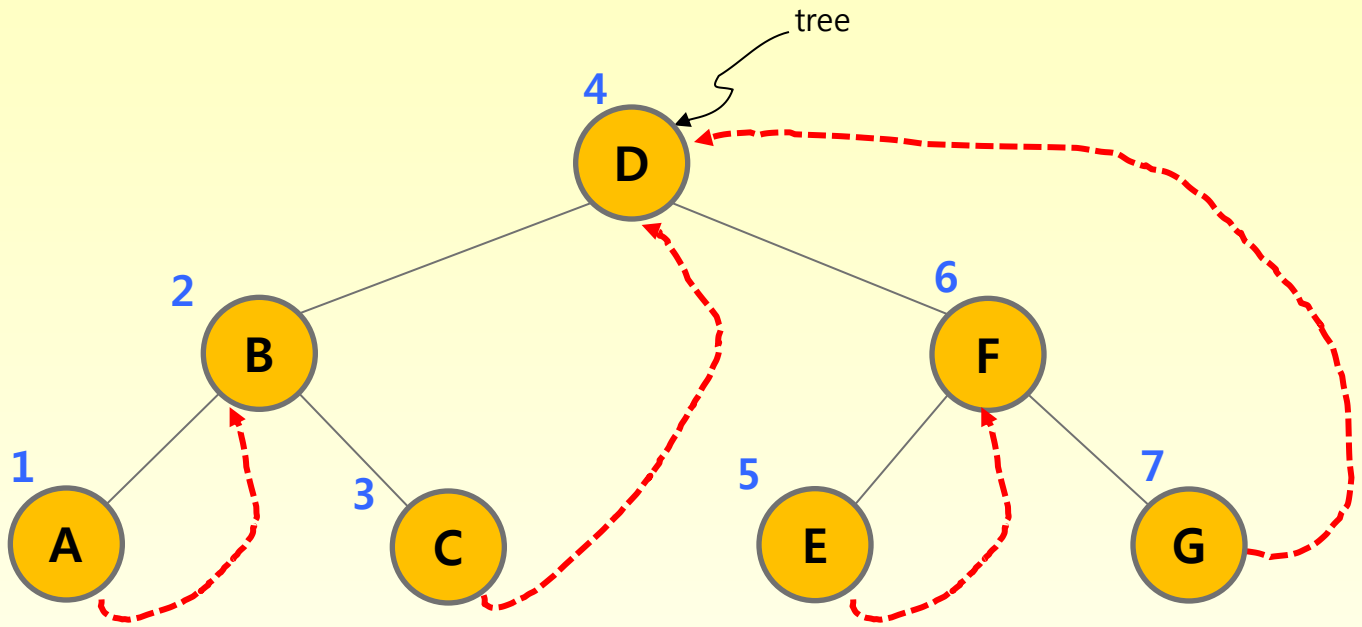


스레드(threaded) 이진트리

- 트리는 단방향 구조 ➔ 순회 과정에서 함수호출/리턴의 반복 동작
- 단말노드는 모두 NULL 링크 ➔ 비사용 NULL 링크를 활용하여 함수호출에 의한 트리 순회를 하지 않고도 노드들을 쉽게 순회할 수 있는 방법 제공
  - NULL이 저장된 링크에 중위순회의 다음 순서 노드에 대한 포인터 저장
  - 즉, 다음에 방문할 노드 주소를 빈 링크에 저장해두면 함수호출 불필요
    ➔ 자식노드로 이동은 llink/rlink 이용, 부모노드로 이동은 변형 link 이용
- 단말/비단말 노드 구별하기 위해 is_thread 필드를 노드에 포함시켜 구성

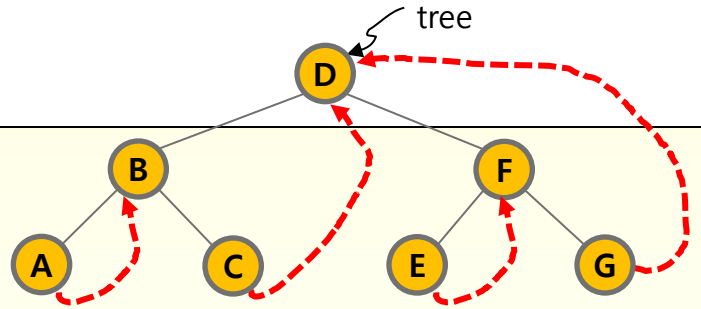
```
typedef  struct  TreeNode {
     int  data;
     struct  TreeNode  *llink;
     struct  TreeNode  *rlink;
     int  is_thread;  // 스레드 여부(0/1)
} TreeNode;
```

중위순회를 위한 스레드(threaded) 이진트리



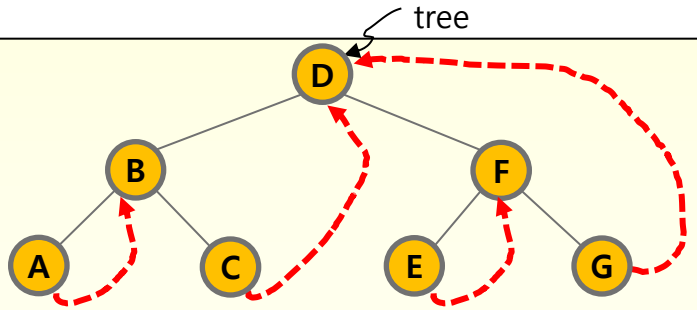
중위순회를 위한 스레드(threaded) 이진트리

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



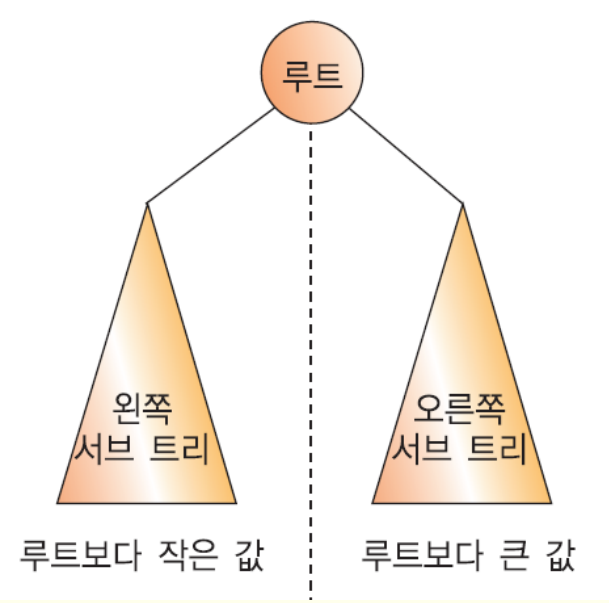
스레드를 이용한 중위순회

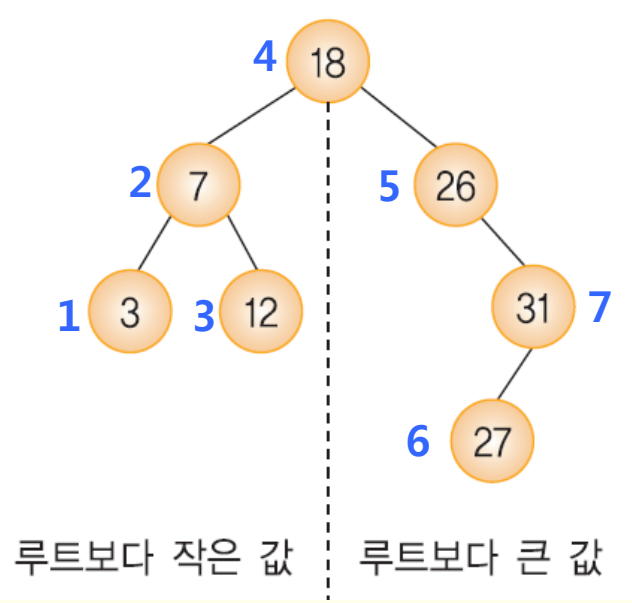
```
void   thread_inorder(TreeNode   *tree)
{
    TreeNode *q = tree;
    while (q->llink != NULL)
        q = q->llink;     // q를 트리의 가장 왼쪽 노드로 이동
    do {
        printf(" %d ", q->data);    // 데이터 출력
        q = find_successor(q);      // 후속자 함수 호출
    } while(q != NULL);             // NULL이 아닐동안 끝까지 추적
}
```



이진 탐색트리

- 이진 탐색트리는 빠르고 효율적인 검색을 위해 사용하는 자료구조
- 구조 특성 : key(왼쪽서브트리) ≤ key(루트노드) ≤ key(오른쪽서브트리)
- 이진 탐색트리를 중위 순회 ➔ 오름차순으로 정렬된 순서로 검색 가능





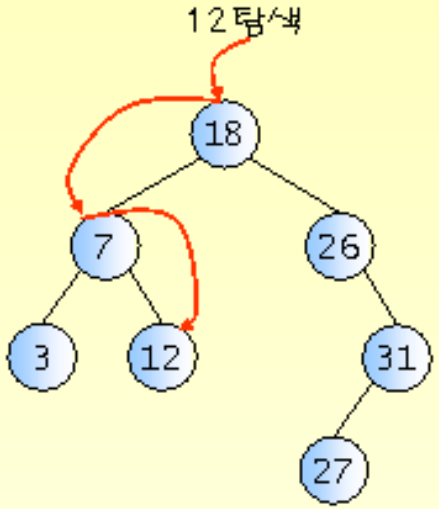
이진 탐색트리의 탐색 연산(순환적)

▣ 검색 키와 트리의 노드 데이터 비교

1. 비교한 결과가 검색 키 = 노드 데이터 이면,
   탐색 성공이므로 노드주소 리턴

2. 비교한 결과가 검색 키 < 노드 데이터 이면,
   루트의 왼쪽 자식으로 이동(왼쪽 링크를 따라
   탐색노드 변경)하여 찾을 때까지 되풀이

3. 비교한 결과가 검색 키 > 노드 데이터 이면,
   루트의 오른쪽 자식으로 이동(오른쪽 링크를
   따라 탐색노드 변경)하여 찾을 때까지 되풀이




```
int  search(tree, k) // 트리주소, 검색키
{
    p = tree;
    if (p == NULL) return NULL;
    if (k = p->data) return x;
    else  if  (k < p->data)
                return  search(p->llink, k);
          else return  search(p->rlink, k);
}
```

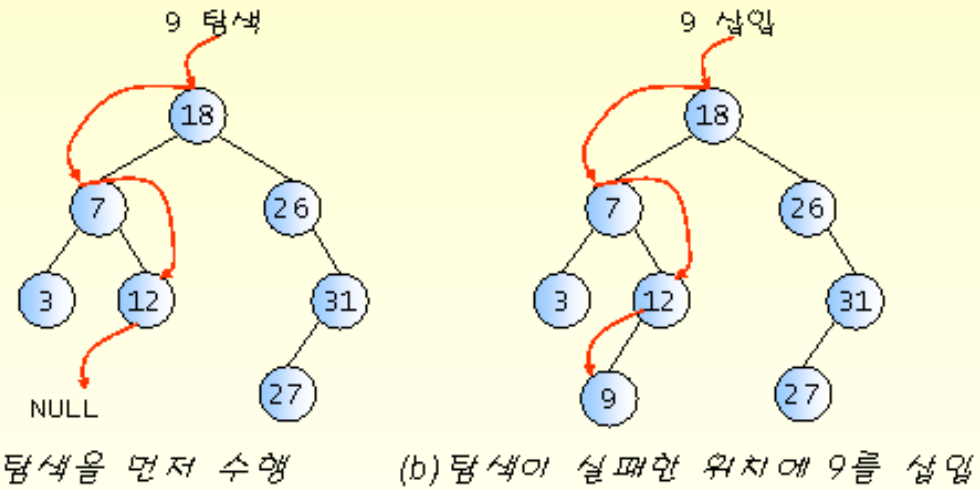
이진 탐색트리의 탐색 연산(반복적)

```
//순환법
int  search(tree, k) // 트리주소, 검색키
{
    p = tree;
    if (p == NULL) return NULL;
    if (k = p->data) return x;
    else  if  (k < p->data)
                return  search(p->llink, k);
          else  return  search(p->rlink, k);
}
```

```
//반복법
int  search(tree, k) // x : 포인터, k : 검색키
{
    p = tree;
    if (p == NULL) return NULL;
    while(p != NULL)
    {
        if (k == p->data) return p;
        else if (k < p->data)  p = p->llink;
             else              p = p->rlink;
    }
}
```

이진 탐색트리의 삽입 연산

1. 이진 탐색트리에 데이터를 삽입하려면, 삽입할 위치를 검색하는 것이 첫 번째 절차임
2. 적절한 위치(더 이상 탐색 불가) 노드의 왼쪽 또는 오른쪽 자식으로 새로운 노드를 삽입

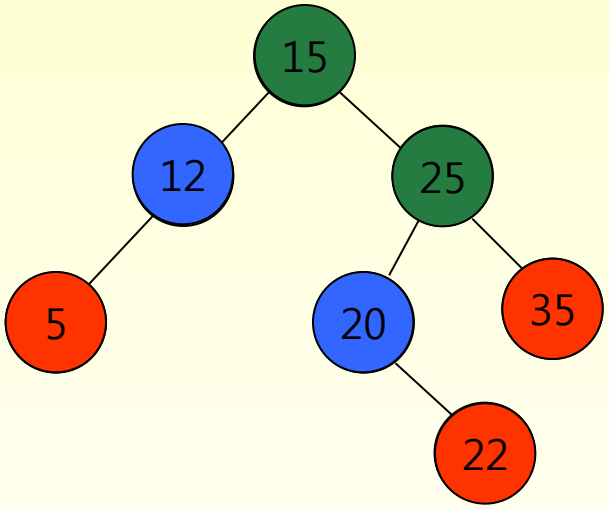


(a) 탐색을 먼저 수행 (b) 탐색이 실패한 위치에 9를 삽입

```
TreeNode * insert_node(tree, x)
{
  TreeNode *node, *p = tree;
  if (p==NULL) { //빈 트리일 경우
    node = 새 노드를 할당;
    node->data = x;
    node->llink = node->rlink = NULL;
    return  tree = node;
  }
  if (x < p->data)
        p->llink = insert(p->llink, x);
  else  p->rlink = insert(p->rlink, x);
  return  p;
}
```

이진 탐색트리의 삭제 연산

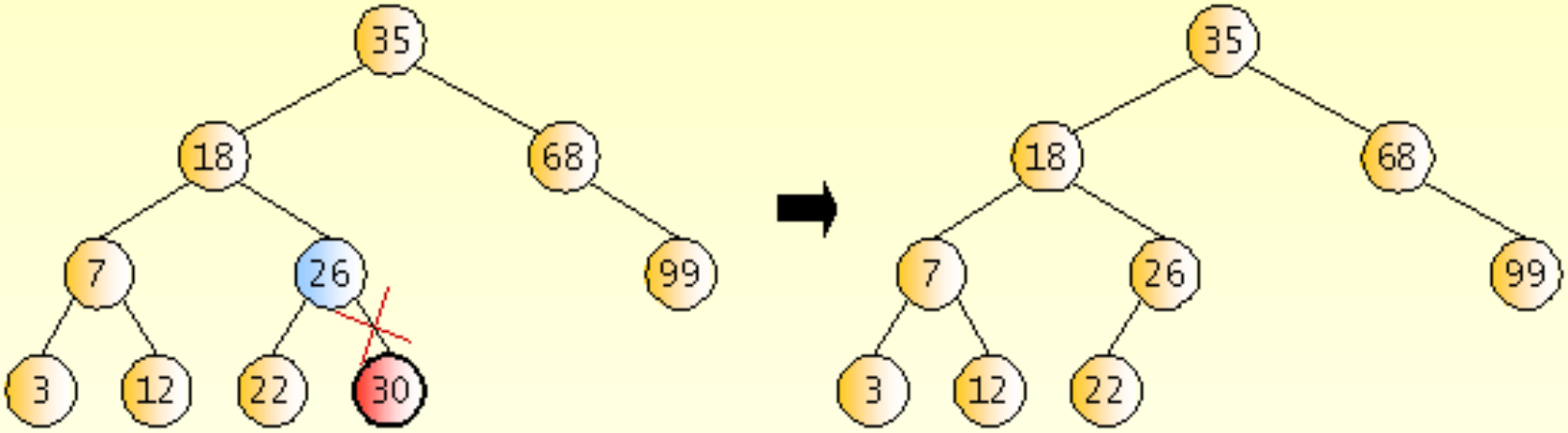
- 삭제 연산에서 고려할 노드 상태의 3가지 경우
  - [case 1] 삭제하려는 노드가 단말(terminal) 노드일 경우
  - [case 2] 삭제하려는 노드가 1개의 자식 노드를 가지고 있는 경우
  - [case 3] 삭제하려는 노드가 2개의 자식 노드를 모두 가지고 있는 경우



이진 탐색트리의 삭제 연산

[CASE 1] 삭제하려는 노드가 단말 노드일 경우 : 간단함

- 단말 노드의 父母 노드를 찾아서 링크 값을 NULL로 저장



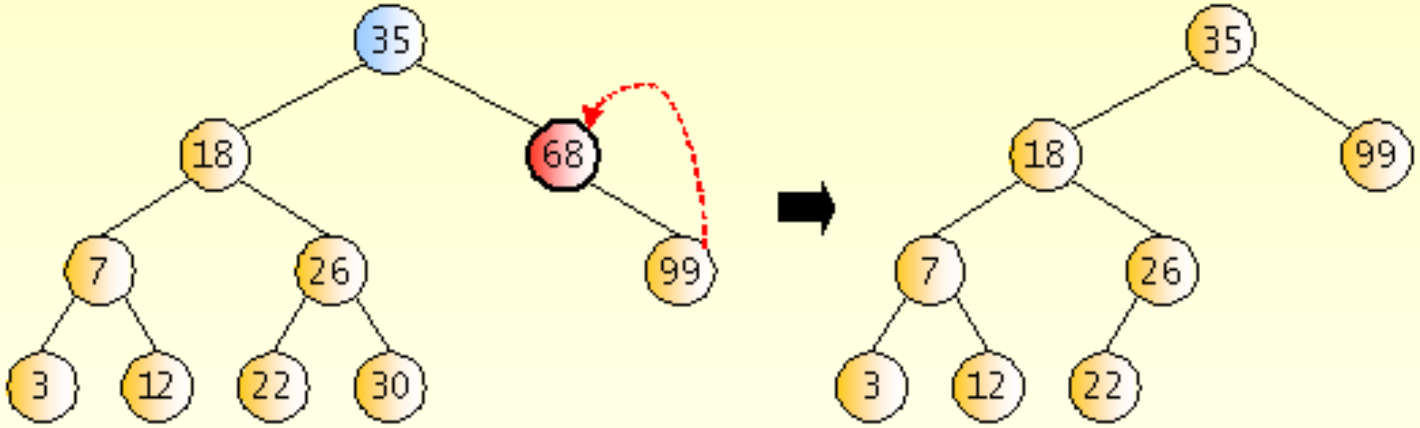
(a) 삭제전 (b) 삭제후

그림 7.42 이진탐색트리의 삭제연산: 삭제노드가 단말노드인 경우

이진탐색트리의 삭제 연산

[CASE 2] 삭제하려는 노드가 하나의 서브트리만 가지고 있는 경우

: 삭제할 노드가 왼쪽이나 오른쪽 서브 트리를 가지고 있는 경우, 삭제하려는 노드 자리에 서브 트리의 루트 노드를 대체



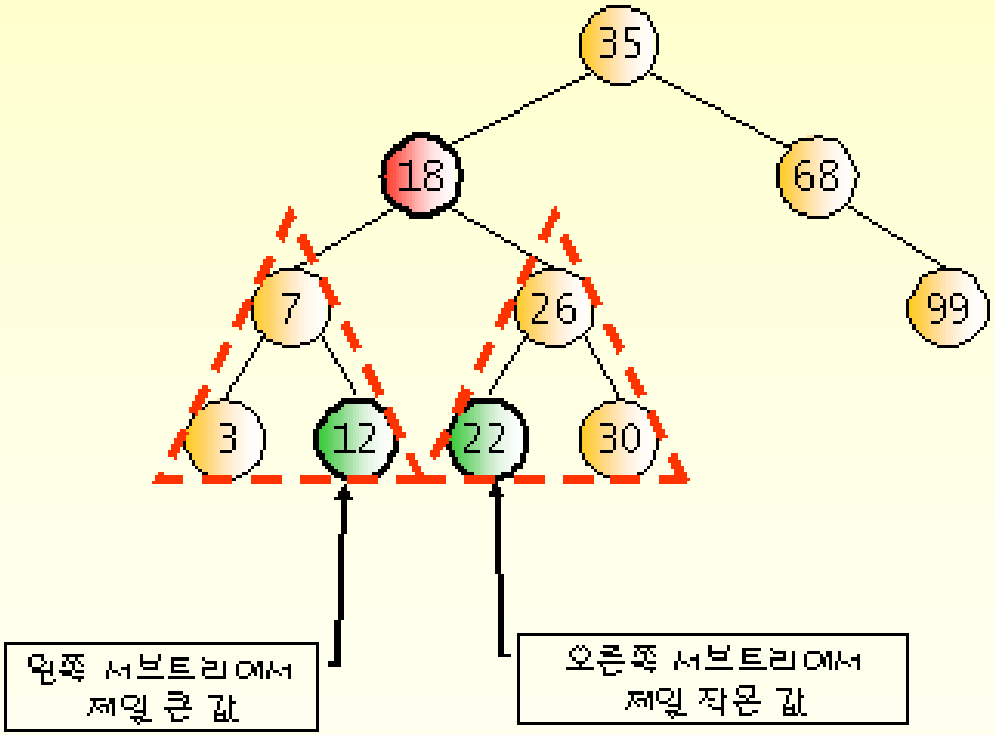
(a) 삭제전 (b) 삭제후

그림 7.43 이진탐색트리의 삭제연산: 삭제노드가 하나의 서브트리를 가지고 있는 경우

이진탐색트리의 삭제 연산

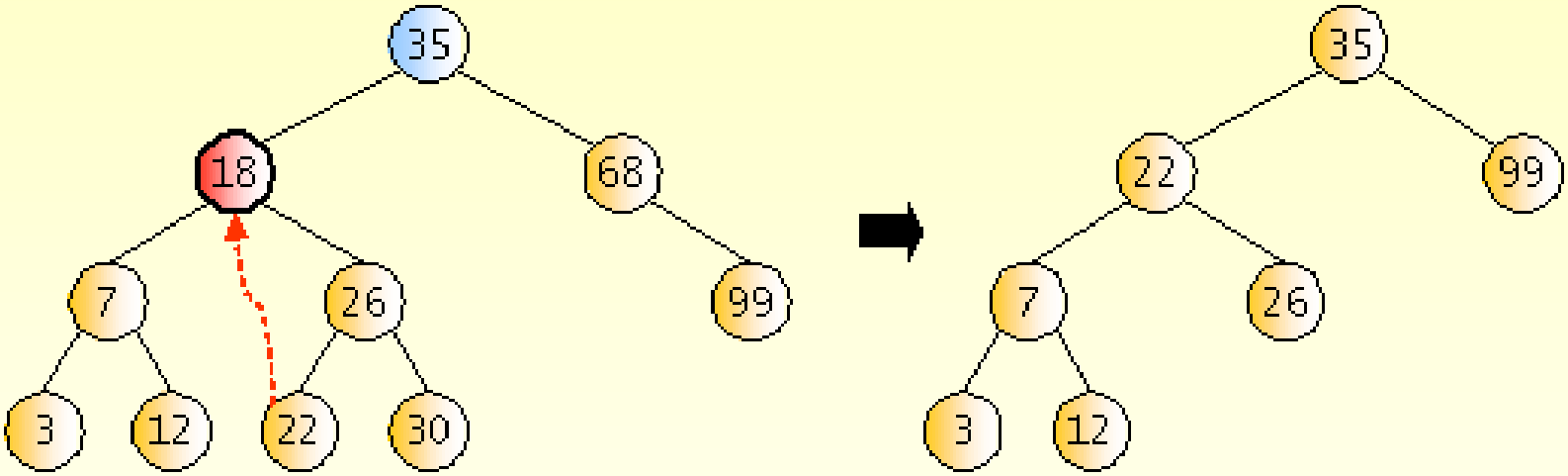
[CASE 3] 삭제하려는 노드가 2개의 서브트리를 가지고 있는 경우

: 삭제할 노드와 가장 가까운 크기의 값(큰 값 중에서 최소 값 or 작은 값 중에서 최대 값)을 가진 노드를 삭제할 노드와 대체



이진탐색트리의 삭제 연산

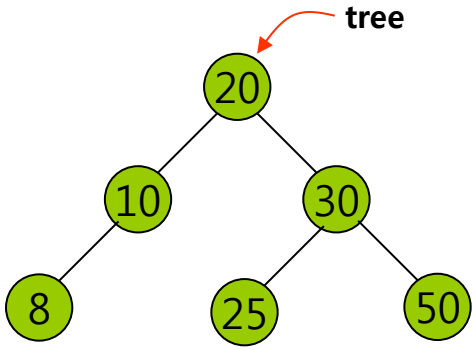
예 : 삭제하려는 노드가 2개의 서브트리를 가지고 있는 경우, 삭제할 노드를 큰 값 중에서 최소값을 가진 노드로 대체하는 예제



이진 탐색트리의 삭제 3가지 연산

```
TreeNode * delete_node(TreeNode * root, int key)  {
    TreeNode  *temp, *p = root;
    if (p == NULL) return root;
    if (key < p->data)  p->left = delete_node(p->llink, key); //왼쪽으로
    else if (key > p->data) p->rlink = delete_node(p->rlink, key); //오른쪽으로
        else { // key == p->data 일 경우, 3가지 경우를 고려해서 노드 삭제
            if (root->llink == NULL)  //왼쪽 자식이 없는 노드일 경우
              { temp = p->rlink; free(p);  return temp; }
            else if (root->rlink == NULL) //오른쪽 자식이 없는 노드일 경우
                  { temp = p->llink; free(p);  return temp; }
            // 자식이 둘다 존재하는 노드일 경우, 대체할 노드를 검색
            temp = p->rlink;
            while (temp->llink != NULL)
                  temp = temp->llink;
            p->data = temp->data;  //노드 데이터를 대체
            p->rlink = delete_node(p->rlink, temp->data); //노드 삭제
          }
    return  root;
}
```

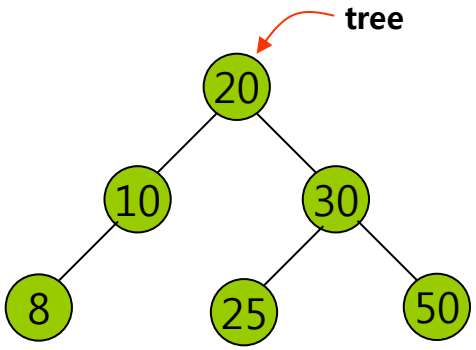
이진 검색트리 구현 by 연결리스트




```
… …
TreeNode  *tree = NULL;  //입력순서 : 20-30-25-10-8-50
void main()
{
   int  menu, item;
   while (1)
   {
      printf("이진트리 메뉴 1)삽입  2)삭제  3)출력  4)검색  5)순회  …  7)종료 : ");
      scanf("%d", &menu);
      if (menu == 7) break;
      switch (menu) {
         case 1: insert_node(); break;    // 입력한 값을 이진트리의 자기 자리에 삽입
         case 2: delete_node();  break;  // 입력한 값을 검색하여 삭제(자식노드 연결)
         case 3: out();          break;   // 루트부터 출력(출력순서 선택)
         case 4: search();    break;   // 값 입력하여 검색하여 위치 리턴
         case 5: break;  //case 계속됨
         default : printf("₩n 잘못된 메뉴… "); break;
      }
   }
}
```

이진 검색트리 표현 by 연결리스트

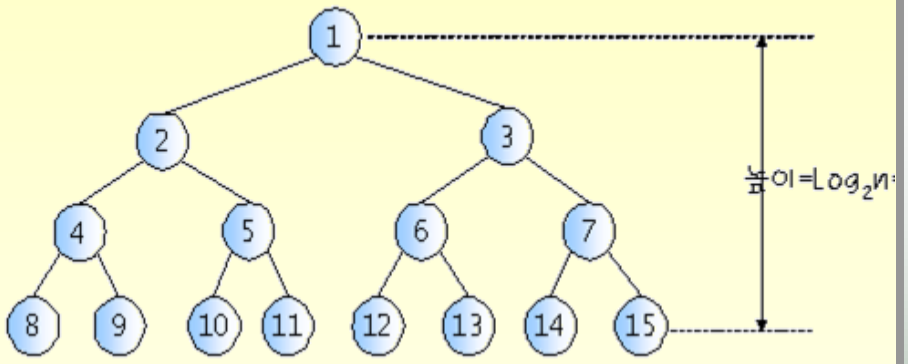
```
TreeNode * insert_node(      )
{// 값 입력 → 루트에서 시작하여 정확한 삽입위치를 찾기 → 노드 구성해서 연결



}
TreeNode * delete_node(      )
{// 값 입력 → search() 함수 호출 → 리턴된 노드를 삭제 → 자식노드 연결 마무리



}
void   out(     )
{//출력 유형에 따라 출력

}
TreeNode * search(     )
{//값 입력 → 루트부터 검색하여 정확한 위치 찾기 → 노드 주소 리턴

}
```



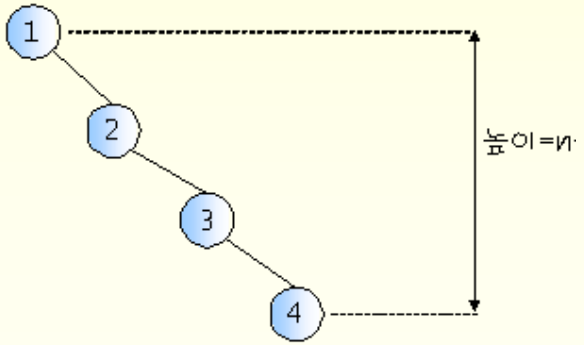
이진탐색트리의 성능

- 이진 탐색트리에서의 **탐색, 삽입, 삭제 연산의 시간 복잡도**는 노드 개수에는 상관없이 트리의 **높이 h**에 비례

- **최선의 경우**
  - 이진 탐색트리가 좌우 서브트리의 높이 또는 노드 개수가 균형적으로 생성된 경우
  - $h = \log_2 n$

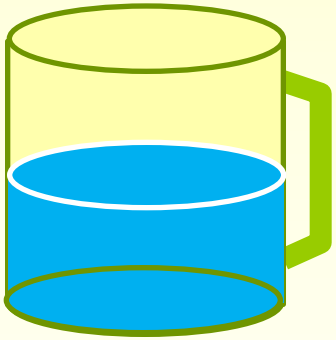


- **최악의 경우**
  - 한쪽으로 치우친 경사진 이진트리의 경우
  - $h = n$
  - 순차 탐색의 시간복잡도가 같아짐



잠깐 쉬어가면서 "Think"

- 퀴즈-1 : 빈 방에 컵이 있다. 컵은 생맥주 잔과 같이 곧은 원통 모양으로, 밑바닥과 입구의 지름이 같다. 컵에는 물이 절반 정도 채워져 있다.
  컵의 물이 절반이 넘는지, 넘지 않는지를 판단할 수 있는 방법은?
  이때, 아무런 도구나 기구는 방 안에 존재하지 않는다.
  온도, 화학반응, 물을 마셔 본다는 등의 억지스러운 상상은 스스로 촌스러워지는 지름길 !!



잠깐 쉬어가면서 "Think"

- 퀴즈-2 : 63 빌딩에서 계란을 떨어뜨려 깨어지지 않는 층수를 맞추시오. 몇번 만에 계란이 깨어지지 않는 층 수를 알아낼 수있는가?