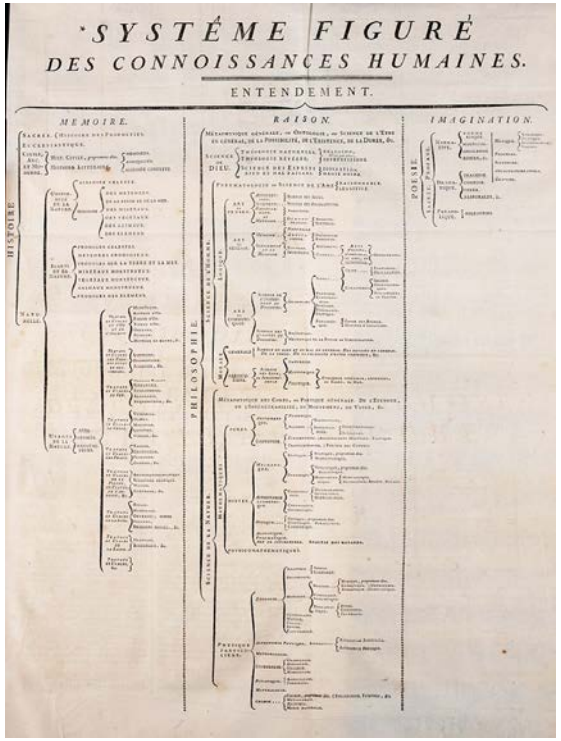
트리
(Trees)

소프트웨어학부
이 의 종

개요
(Introduction)

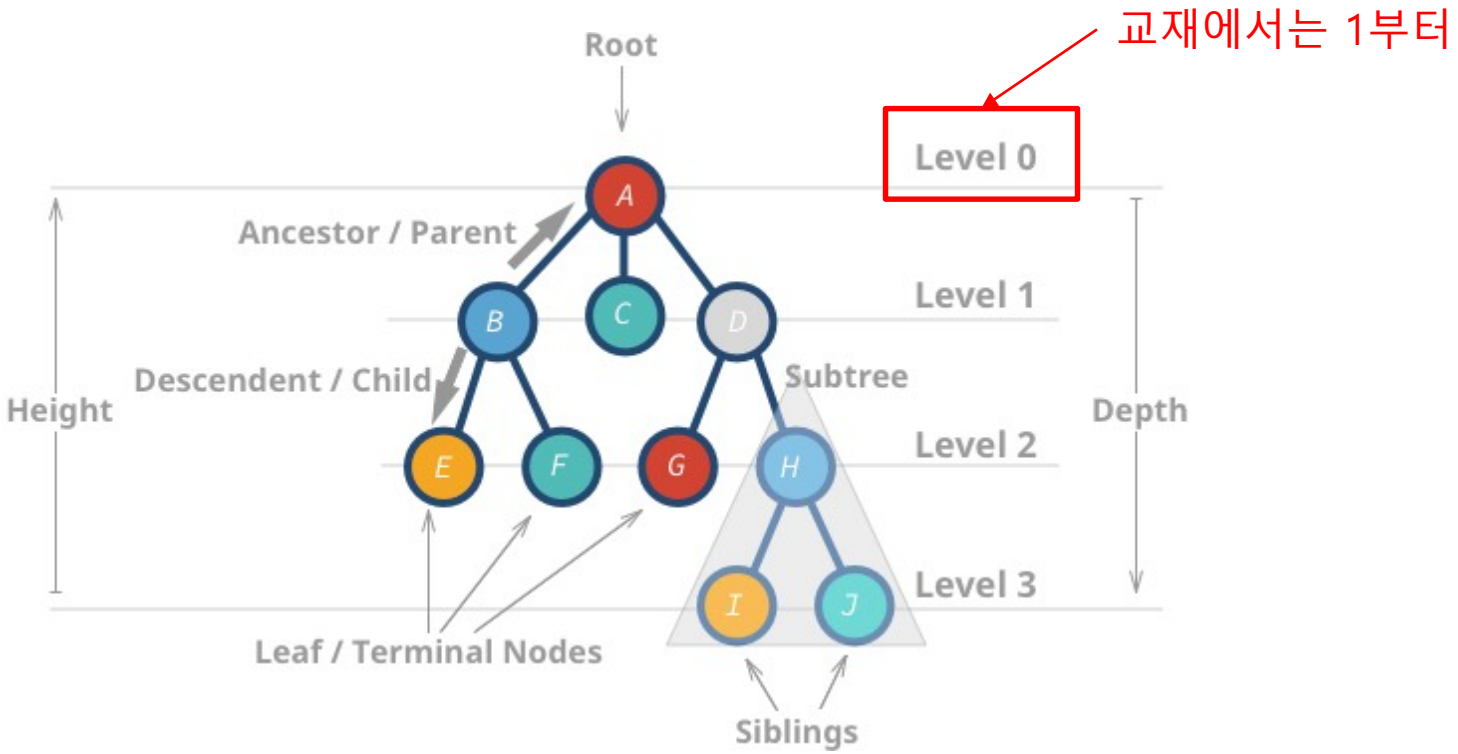


The original Encyclopédie used a tree diagram to show the way in which its subjects were ordered.

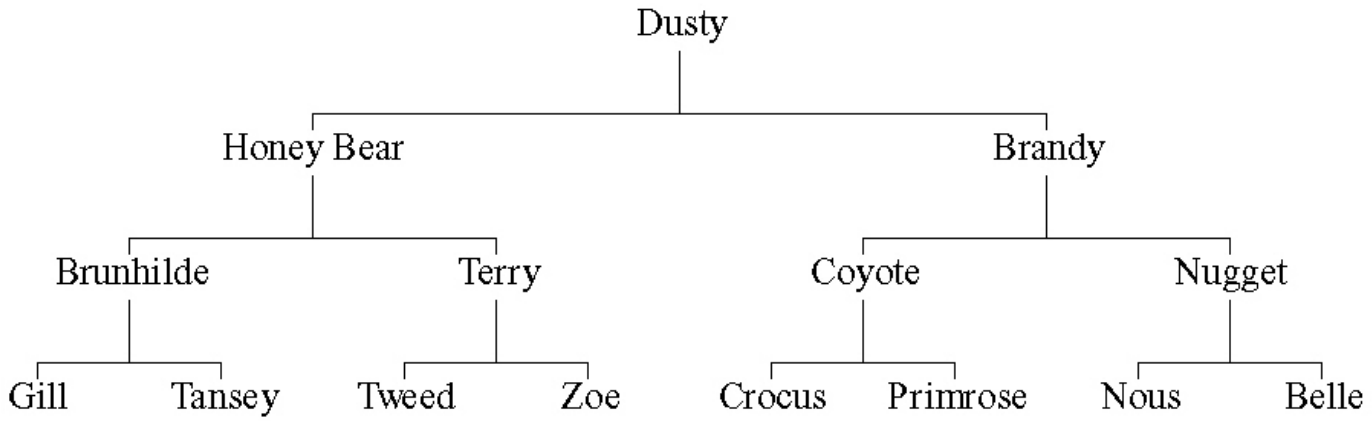
-- wikipedia

기본 용어(Terminology)

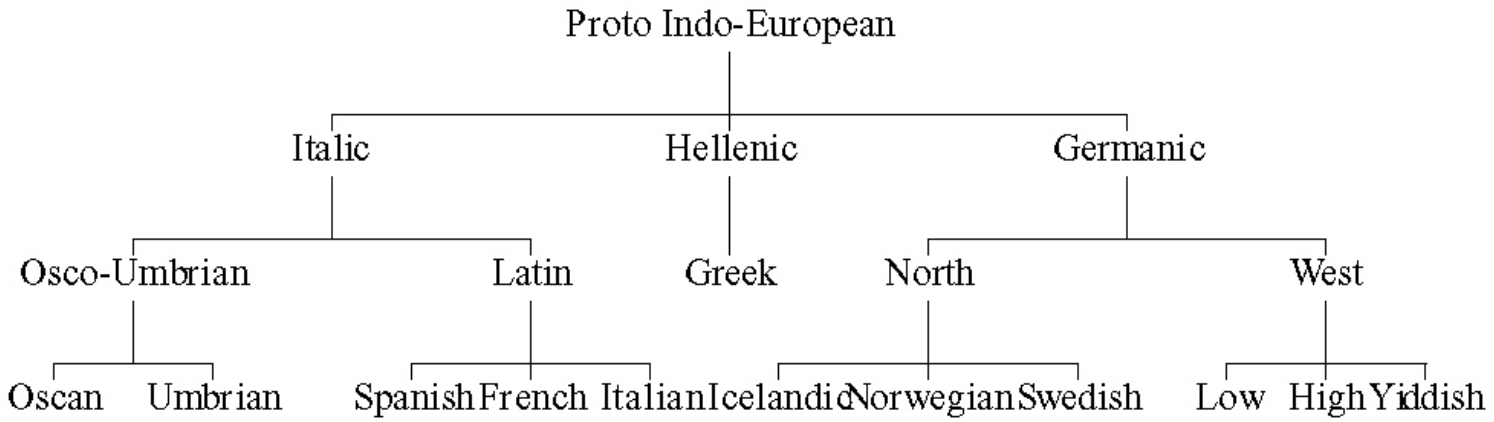
- 중요한 자료구조 중 하나
- 정보의 항목들이 가지(branch)로 연결되는 데이터 구조



기본 용어(Terminology)



(a) Pedigree

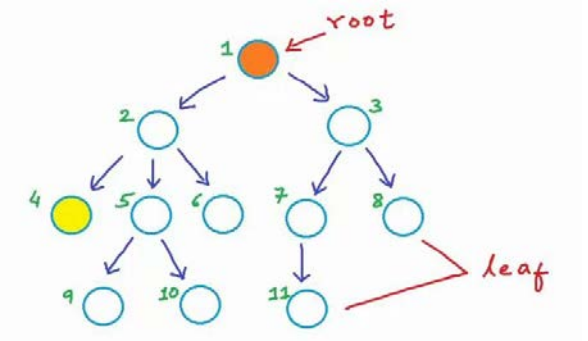


(b) Lineal

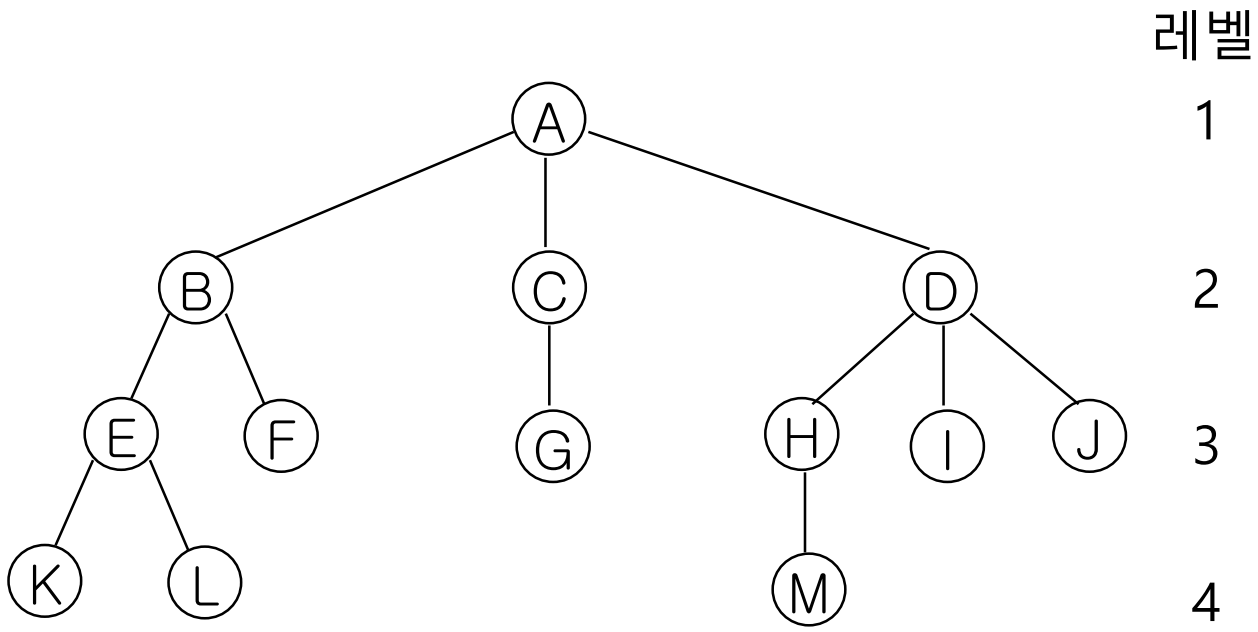
기본 용어(Terminology)

- **(정의)트리 : 하나 이상의 노드(node)로 이루어진 유한집합**
  1) 루트(root) 노드 1개
  2) 나머지 노드들은 n(≥0)개의 분리 집합T_1, T_2, … , T_n으로 분할 (T_i : 루트의 서브트리)

  - 노드 : 데이터(정보) 아이템 + 다른 노드로 뻗어진 가지
  - 노드의 차수(degree) : 노드의 서브트리 수
  - 단말(leaf) 노드 : 차수(degree) = 0
  - 비단말 노드 : 차수 ≠ 0
  - 자식 : 노드 X의 서브트리의 루트 (↔ 부모)
  - 형제(sibling) : 부모가 같은 자식들
  - 트리의 차수 = max{노드의 차수}
  - 조상 : 루트까지의 경로상에 있는 모든 노드
  - 노드 레벨 : 루트=레벨1, 자식 레벨=부모 레벨+1
  - 트리의 높이(깊이) = max{노드 레벨}



기본 용어(Terminology)

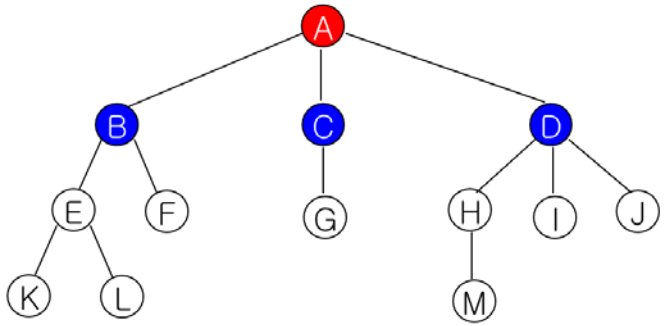


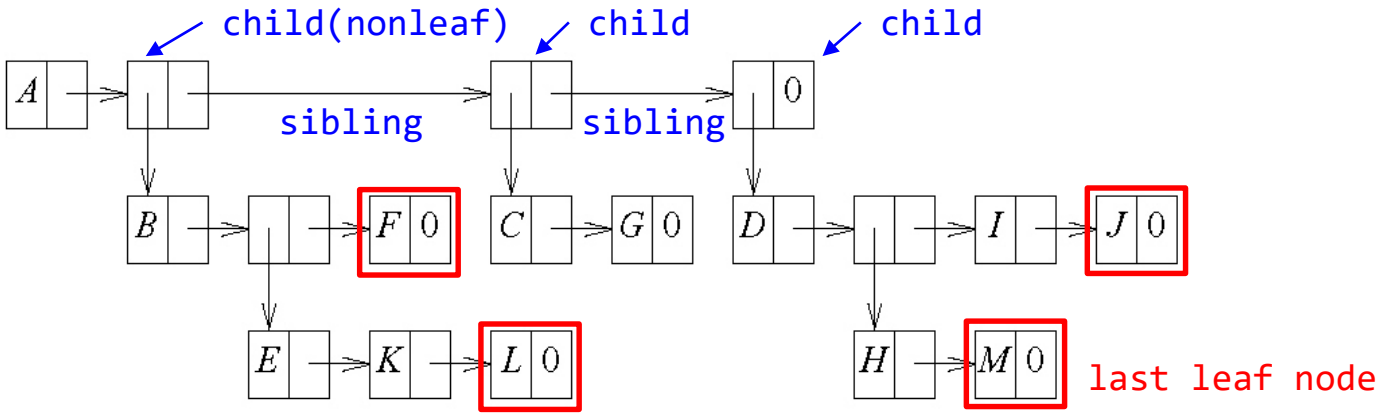
- **root node = A**
- **node 차수(degree)**
  - A = 3, C = 1, G = 0
- **leaf node = K, L, F, G, M, I, J**
- **non leaf node = A, B, C, D, E, H**
- **child nodes**
  - B, C, D(A의), E, F(B의) ..
- **sibling nodes**
  - (E, F), (B, C, D), (H, I, J) 등
- **ancestors = K's → E, B, A**
- **node level = A = 1, B = 2**
- **height = 4 (max node level)**
- **degree of tree**
  - 3 (max node degree)
- **subtrees**
  - subtree B, subtree C, subtree D
  - subtree E 등

트리의 표현(Representation of Trees)

- 리스트표현 (List Representation)

(A(B(E(K,L),F),C(G),D(H(M),I,J)))

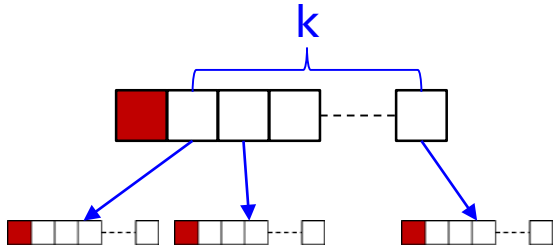




트리의 표현(Representation of Trees)

- 차수가 k인 트리에 대한 노드 구조

DATA	CHILD 1	CHILD 2	···	CHILD k



공간 낭비 많음

k원 트리에서 노드 수가 n이면 nk의 자식 필드 중 n(k-1)+1개 필드가 0

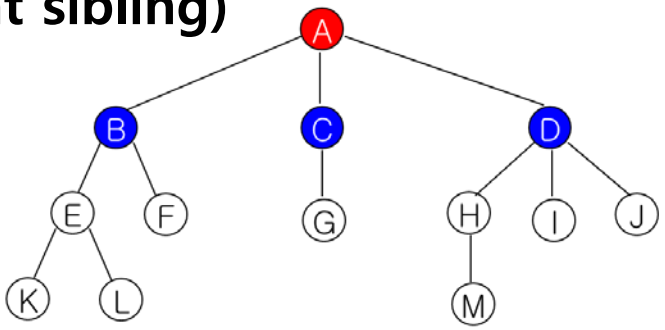
nk – (n-1)
=n(k-1)+1

worst case에서 k=n-1

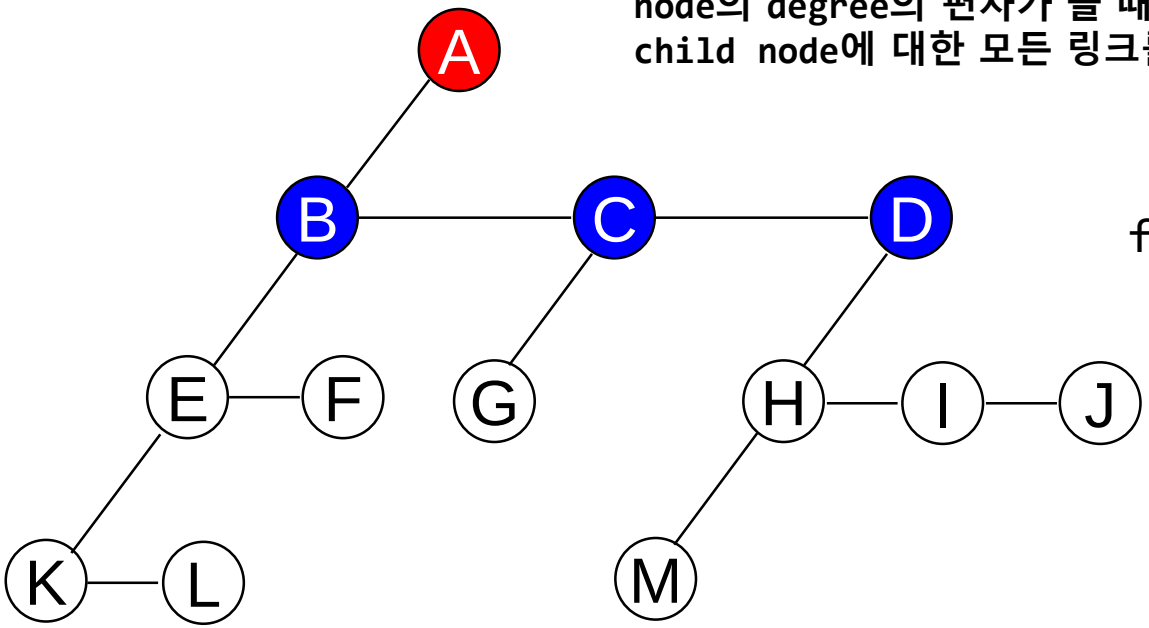
트리의 표현(Representation of Trees)

- 왼쪽 자식-오른쪽 형제 표현(left child-right sibling)

data	
left child	right sibling



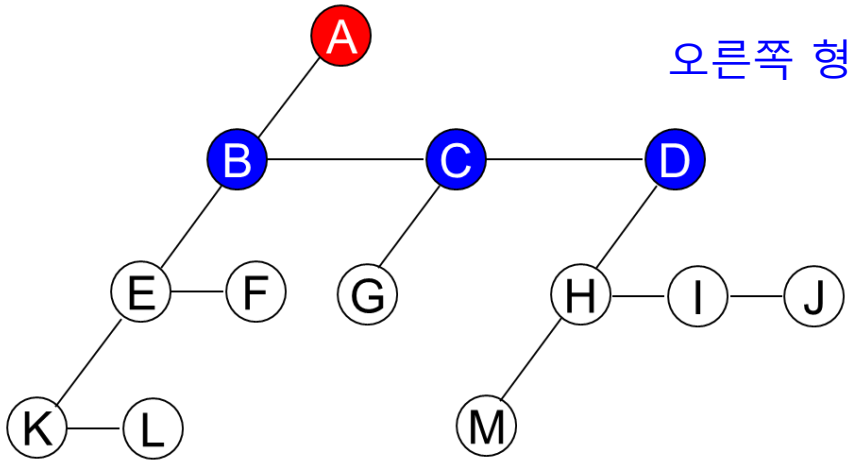
Note:
node의 degree의 편차가 클 때, degree가 고정되지 않을 때 child node에 대한 모든 링크를 유지하는 것은 낭비.



firstChild->sibling

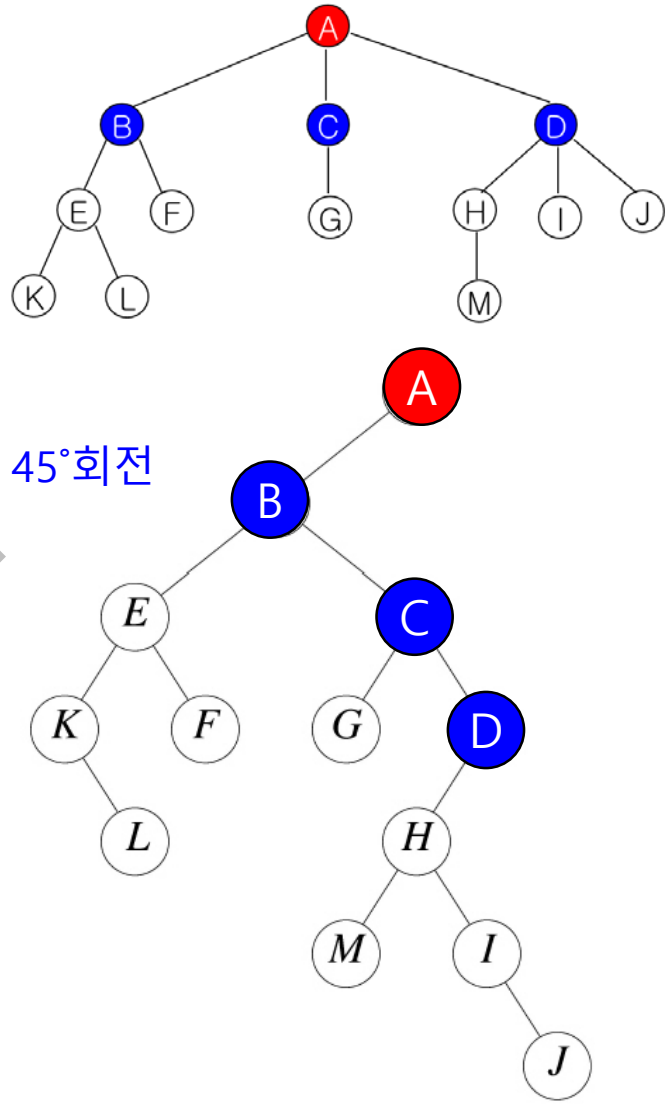
트리의 표현(Representation of Trees)

- **차수가 2인 트리 표현**
  - 왼쪽 자식-오른쪽 형제 트리의 오른쪽 형제 포인터를 45°회전
  - 루트 노드의 오른쪽 자식은 공백



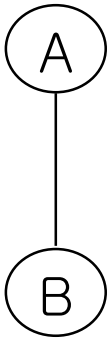
왼쪽 자식-오른쪽 형제 표현

오른쪽 형제 포인터를 45°회전

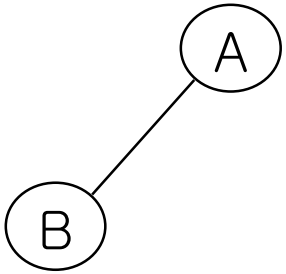


- **이진 트리(binary tree)**
  - 왼쪽 자식-오른쪽 자식 트리

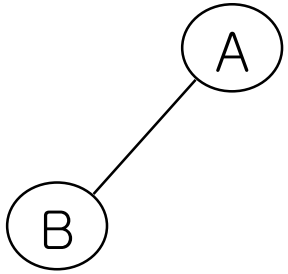
트리의 표현(Representation of Trees)



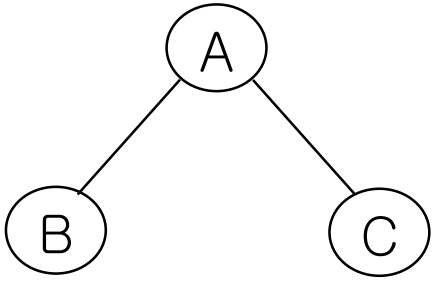
트리



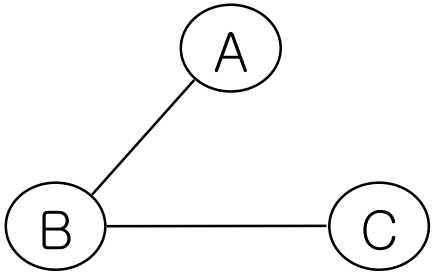
왼쪽 자식-오른쪽 형제 트리



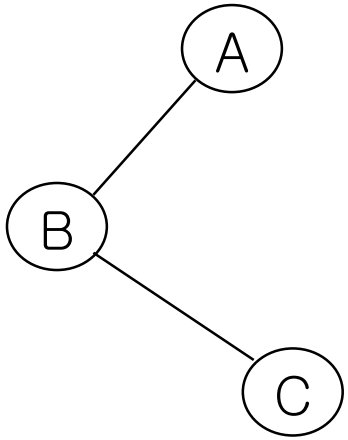
이진트리



트리



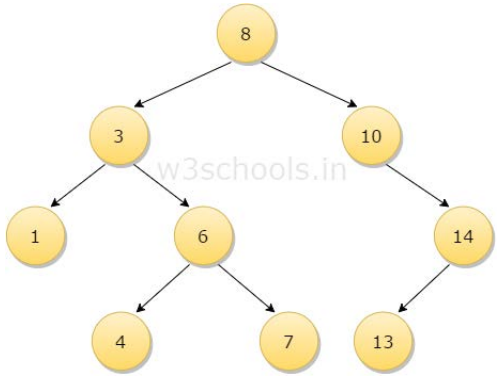
왼쪽 자식-오른쪽 형제 트리



이진트리

이진 트리
(Binary Trees)

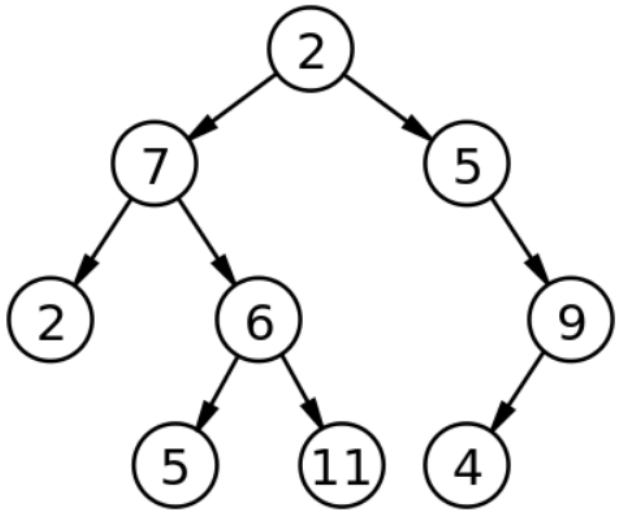
A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes. A binary tree is an important class of a tree data structure in which a node can have at most two children. Child node in a binary tree on the left is termed as 'left child node' and node in the right is termed as the 'right child node.'



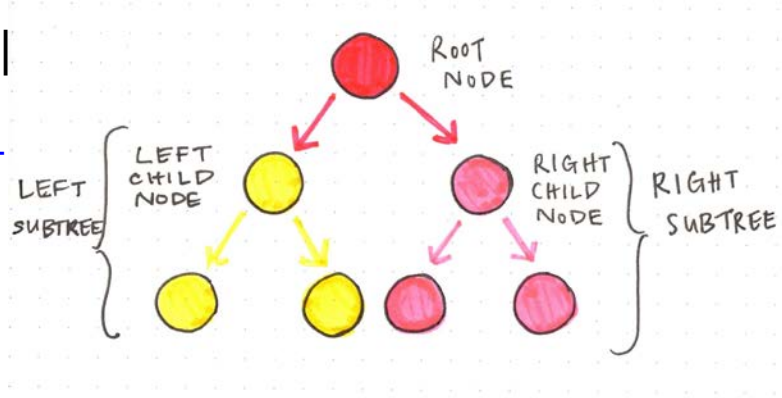
https://www.w3schools.in/data-structures-tutorial/binary-trees/

이진트리(Binary Trees)

- **이진 트리의 특성**
  - 한 노드는 최대 두 개의 자식노드를 가짐
  - 왼쪽 서브트리와 오른쪽 서브트리 구별
  - 0개의 노드를 가질 수 있음

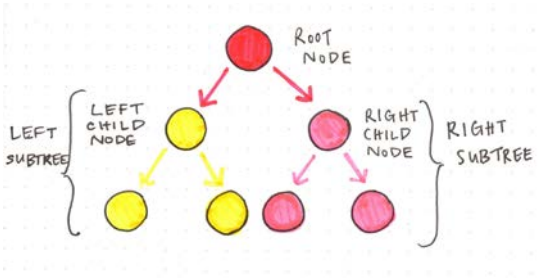


- **이진트리의 정의**
  - 공백이거나
  - 루트와 왼쪽 서브트리, 오른쪽 서브트리 두 개의 분리된 이진 트리로 구성 된 노드의 유한 집합



추상 데이터 타입(Abstract Data Type)

주의: B Tree 아님

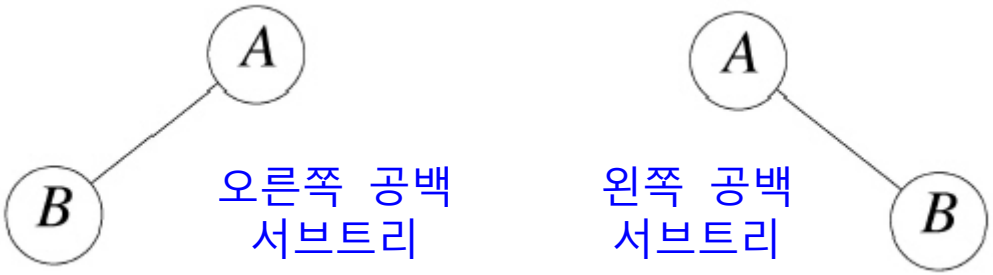



```
ADT Binary_Tree(줄여서 BinTree)
    object: 공백이거나 루트 노드, 왼쪽 Binary_Tree,
            오른쪽 Binary_Tree로 구되는 노드들의 유한집합

    functions:
       모든 bt, bt1, bt2 ∈ BinTree, item ∈ element
       BinTree Create()                ::= 공백 이진 트리를 생성
       Boolean IsEmpty(bt)             ::= if (bt == 공백 이진트리)
                                             return TRUE else return FALSE

       BinTree MakeBT(bt1, item, bt2)::= 왼쪽 서브트리가 bt1, 오른쪽 서브트리가
                                          bt2, 루트는 데이터를 갖는 이진 트리를 반환

       BinTree Lchild(bt)              ::= if(IsEmpty(bt)) return 에러
                                           else bt의 왼쪽 서브트리를 반환

       element Data(bt)                ::= if(IsEmpty(bt)) return 에러
                                           else bt의 루트에 있는 데이터를 반환

       BinTree Rchild(bt)              ::= if(IsEmpty(bt)) return 에러
                                           else bt의 오른쪽 서브트리를 반환
```

상이한 이진트리(Different Binary Trees)

- **이진 트리와 일반 트리의 차이점**
  - 공백 이진 트리 존재
  - 자식의 순서 구별

서로 다른 두 이진 트리



left child

오른쪽 공백 서브트리

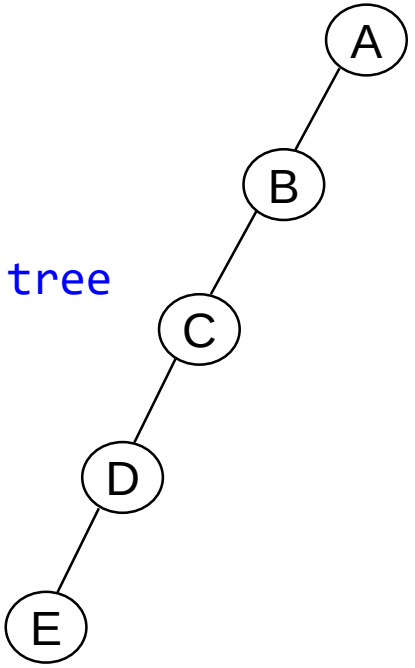
왼쪽 공백 서브트리

right child

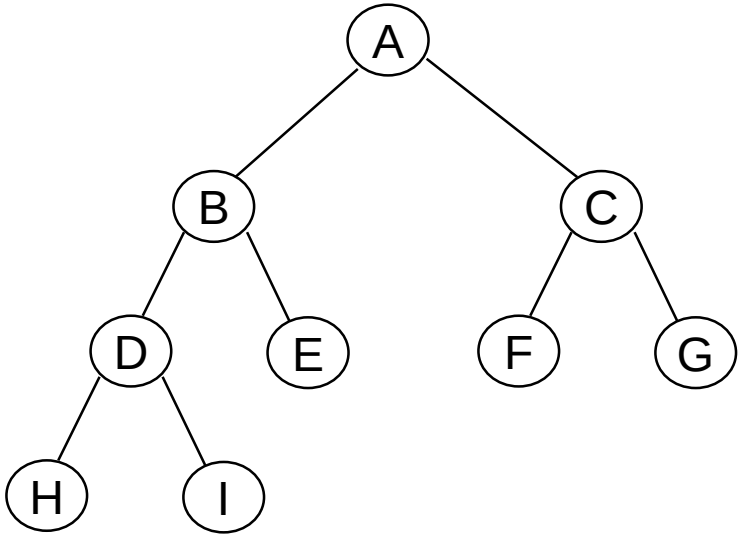
일반 트리로 취급할 경우 두 트리는 동일한 트리

편향 이진 트리와 완전 이진 트리
(Skewed and Complete Binary Trees)

skewed tree



(a) 편향 이진 트리

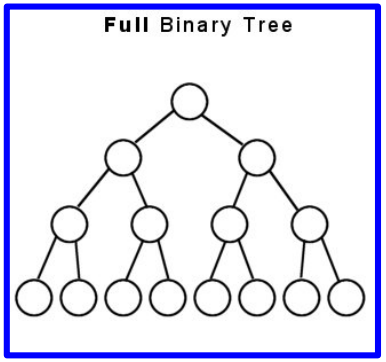


level
1
2
3
4
5

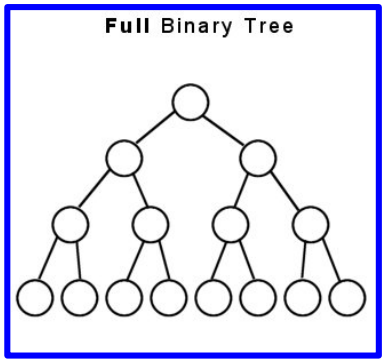
순차적으로 노드를 구성

complete binary tree

(b) 완전 이진 트리



이진트리의 성질(Properties of Binary Trees)



- **레벨 i에서의 최대 노드 수 : $2^{i-1}(i \geq 1)$**
  - 귀납 기초 : 레벨 i=1일 때 루트만이 유일한 노드
    - 레벨 i=1에서의 최대 노드 수 : $2^{1-1} = 2^0 = 1$
  - 귀납 가설 : i를 1보다 큰 임의의 양수라고 가정.
    - 레벨 i-1에서의 최대 노드 수는 2^{i-2}라고 가정
  - 귀납 과정 : 가설에 의해 레벨 i-1의 최대 노드 수는 2^{i-2}
    - 각 노드의 최대 차수는 2
    - 레벨 i의 최대 노드 수는 레벨 i-1에서의 최대 노드 수의 2배
    - 레벨 i에서 최대 노드의 수 = 2^{i-1}

- **깊이가 k인 이진 트리의 최대 노드 수 : $2^k - 1(k \geq 1)$**

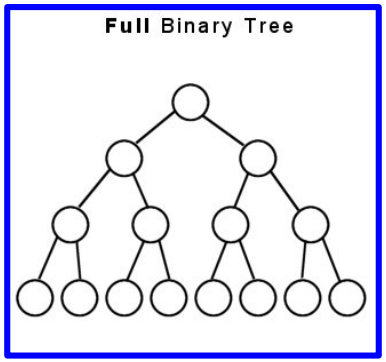
$$\sum_{i=1}^{k}(\text{레벨 } i\text{의 최대 노드 수}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

$$S_n = \frac{a(1-r^n)}{1-r} = \frac{a(r^n-1)}{r-1}$$

등비수열의 합

이진트리의 성질(Properties of Binary Trees)

- **리프 노드 수와 차수가 2인 노드 수와의 관계**
  - $n_0 = n_2 + 1$
    - n_0 : 리프 노드 수
    - n_2 : 차수가 2인 노드 수



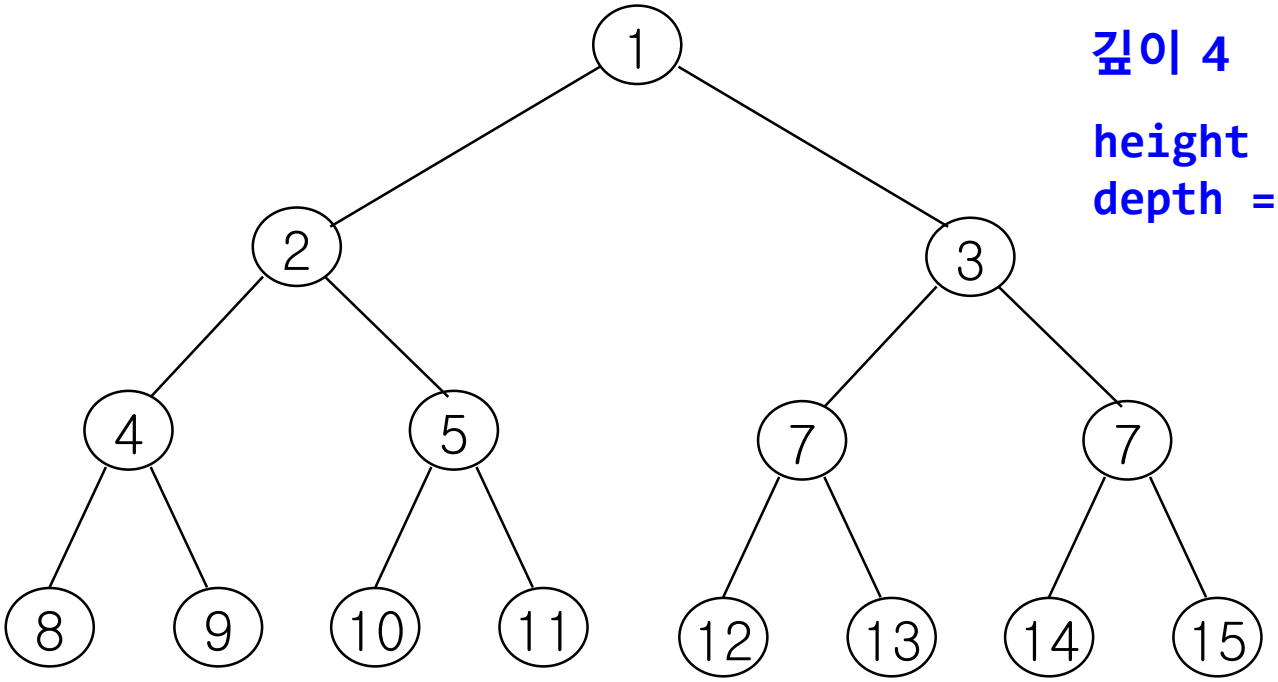
  - 증명
    - n_1 : 차수 1인 노드 수, n : 총 노드 수, B : 총 가지 수
    - $n = n_0 + n_1 + n_2$
    - 루트를 제외한 모든 노드들은 들어오는 가지가 하나씩 있으므로
      $n = B + 1$
    - 모든 가지들은 차수가 2 또는 1인 노드에서 뻗어 나오므로
      $B = n_1 + 2n_2$

$\therefore n = B + 1 = n_0 + n_1 + n_2 = n_0 + B - 2n_2 + n_2$

$n_0 = n_2 + 1$

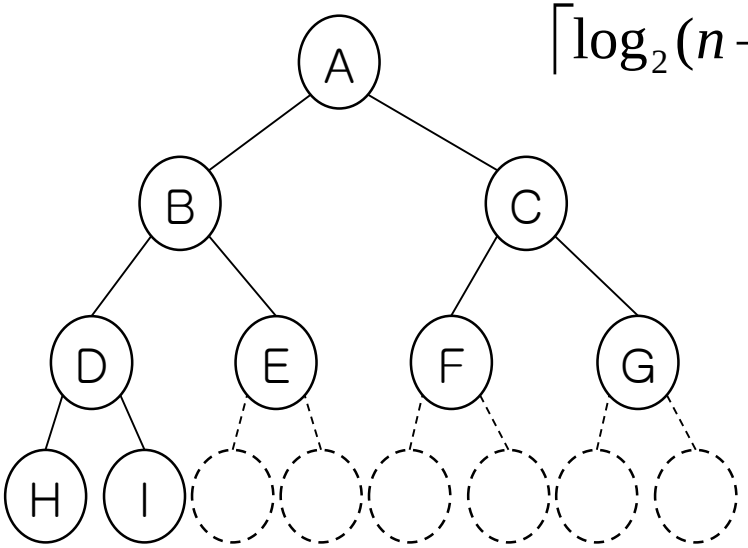
이진트리의 성질(Properties of Binary Trees)

- **포화 이진 트리(full binary tree)**
  - 깊이가 k이고, 노드수가 2^k-1 $(k \geq 0)$인 이진 트리
  - 노드 번호 $1,\ldots,2^k-1$ 까지 순차적 부여 가능



이진트리의 성질(Properties of Binary Trees)

- **완전 이진 트리(complete binary tree)**
  - 깊이가 k이고 노드 수가 n인 이진 트리
  - 각 노드들이 깊이가 k인 포화 이진 트리에서 1부터 n까지 번호를 붙인 노드와 1대 1로 일치
  - n 노드 완전 이진 트리의 높이 : $\lceil \log_2(n+1) \rceil$



깊이가 k인 이진 트리의 최대 노드 수 n : $2^k - 1 (k \geq 1)$

$n = 2^k - 1$

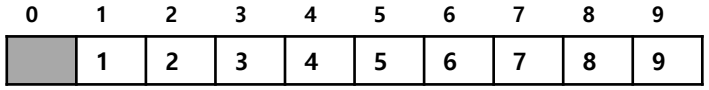
$2^k = n + 1$

$k = \log_2(n+1)$

Full binary tree인 경우

이진트리의 표현(Binary Tree Representations)

- **배열 표현(Array Representation)**



  - 1차원 배열에 노드를 저장
  - 노드번호 i는 배열의 i번째 위치에 저장(사상, mapping)
  - 보조 정리 5.4
    - n 개의 노드를 가진 완전 이진트리

      (1) parent(i) : $\lfloor i/2 \rfloor$ **if** i ≠ 1

      (2) leftChild(i) : 2i **if** 2i ≤ n

      왼쪽 자식 없음 **if** 2i > n

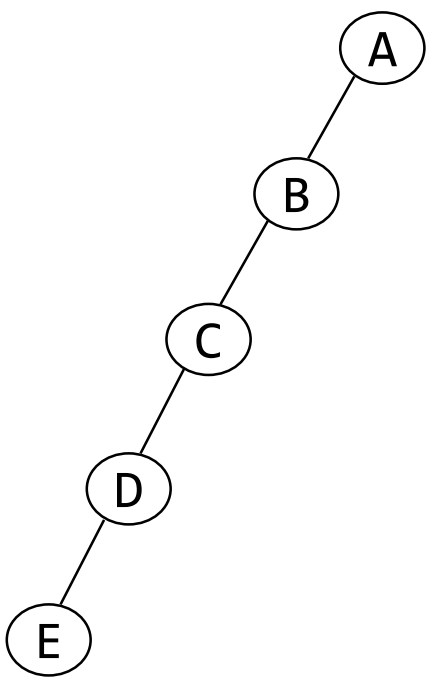
      (3) rightChild(i) : 2i+1 **if** 2i+1≤ n

      오른쪽 자식 없음 **if** 2i + 1 > n

  - 완전 이진 트리 : 낭비 되는 공간 없음
  - 편향 트리 : 공간 낭비
    - 최악의 경우, 깊이 k 편향 트리는 2^k-1중 k개만 사용

이진트리의 표현(Binary Tree Representations)

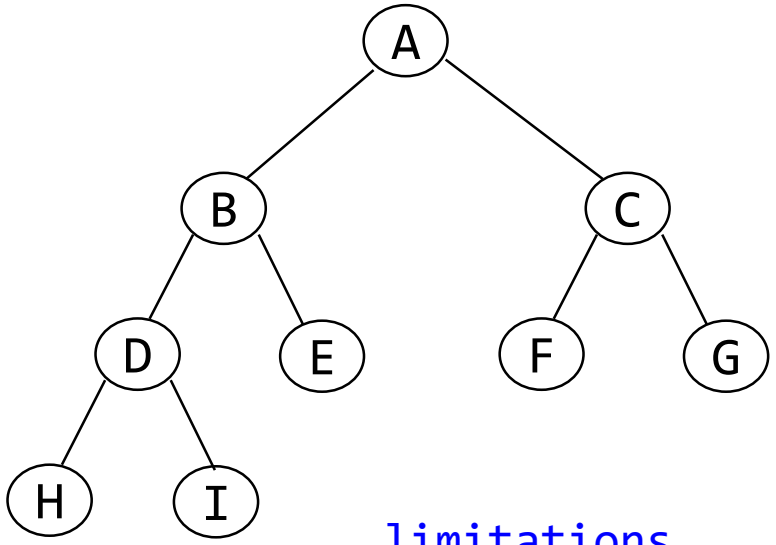
편향트리
(skewed tree)



[0]	-
[1]	A
[2]	B
[3]	-
[4]	C
[5]	-
[6]	-
[7]	-
[8]	D
[9]	-
.	.
.	.
.	.
[16]	E

깊이 k 편향 트리는
2^k-1중 k개만 사용

완전 이진 트리
(complete binary tree)



limitations

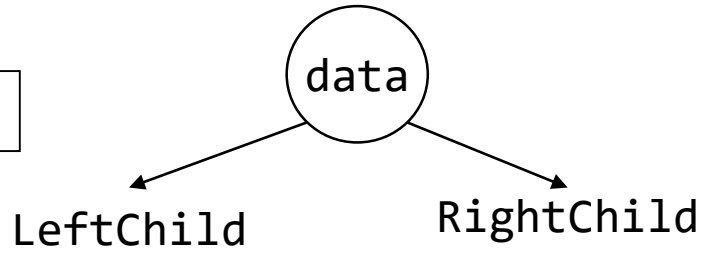
(1) parent(i) :	$\lfloor i/2 \rfloor$	**if** i ≠ 1
(2) leftChild(i) :	2i	**if** 2i ≤ n
	왼쪽 자식 없음	**if** 2i > n
(3) rightChild(i) :	2i+1	**if** 2i+1≤ n
	오른쪽 자식 없음	**if** 2i + 1 > n

[0]	-
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

이진트리의 표현(Binary Tree Representations)

- **연결 표현(Linked Representation)**

노드표현(node)

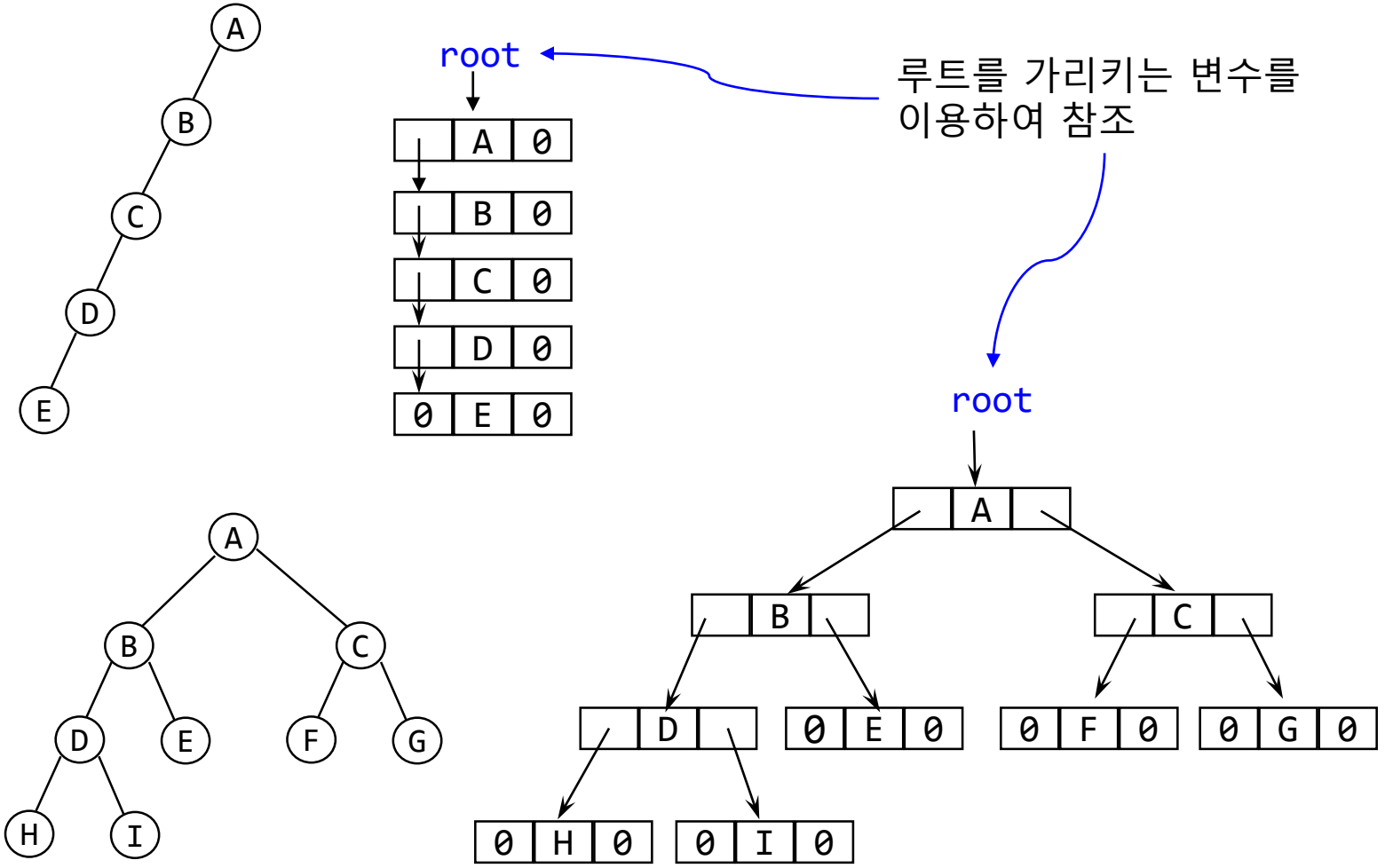



```
typedef struct node *treePointer;
typedef struct node{
        int data;
        treePointer leftChild, rightChild;
        };
```

부모 알기 어려움 → parent 필드 추가하여 해결

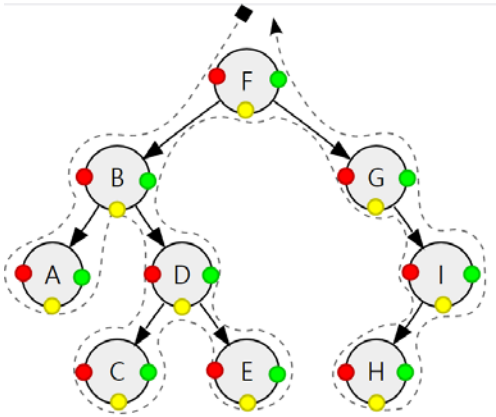
이진트리의 표현(Binary Tree Representations)

- 이진트리의 연결표현(linked representation)의 예



이진트리 순회
(Binary Tree Traversals)

In computer science, tree traversal (also known as tree search and walking the tree) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.



Depth-first traversal of an example tree:
pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

https://en.wikipedia.org/wiki/Tree_traversal

트리 순회(Tree Traversal)

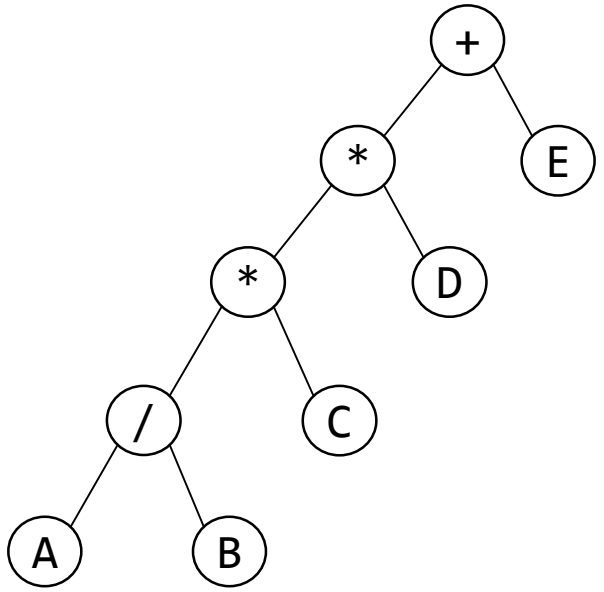
- 트리에 있는 모든 노드를 한 번씩만 방문
- 순회 방법 : LVR, LRV, VLR, VRL, RVL, RLV
  - L : 왼쪽 이동, V : 노드방문, R : 오른쪽 이동
- 왼쪽을 오른쪽보다 먼저 방문(LR)
  - LVR : 중위(inorder) 순회
  - VLR : 전위(preorder) 순회
  - LRV : 후위(postorder) 순회

Graph:

Depth First Traversal

Breath First Traversal

산술식의 이진트리 표현



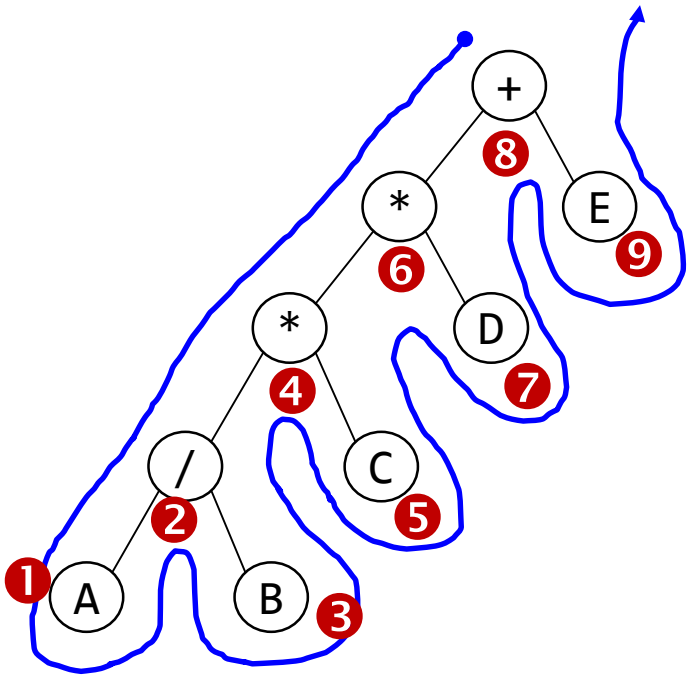
A / B * C * D + E

중위순회(Inorder Traversal)

infix, postfix, prefix와 연계해서 생각

LVR : A / B * C * D + E

```
void inorder(treePointer ptr)
/* 중위 트리 순회 */
{
   if (ptr) {
        inorder(ptr->leftChild);
        printf("%d", ptr->data);
        inorder(ptr->rightChild);
   }
}
```

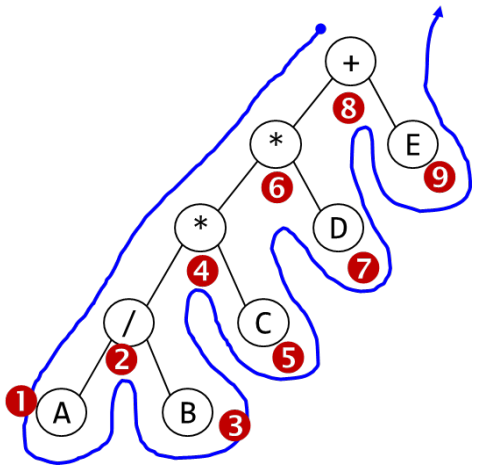


중위순회(Inorder Traversal)

- 중위순회(inorder traversal) 추적

LVR : A / B * C * D + E

```
void inorder(treePointer ptr)
/* 중위 트리 순회 */
{
   if (ptr) {
        inorder(ptr->leftChild);
        printf("%d", ptr->data);
        inorder(ptr->rightChild);
   }
}
```



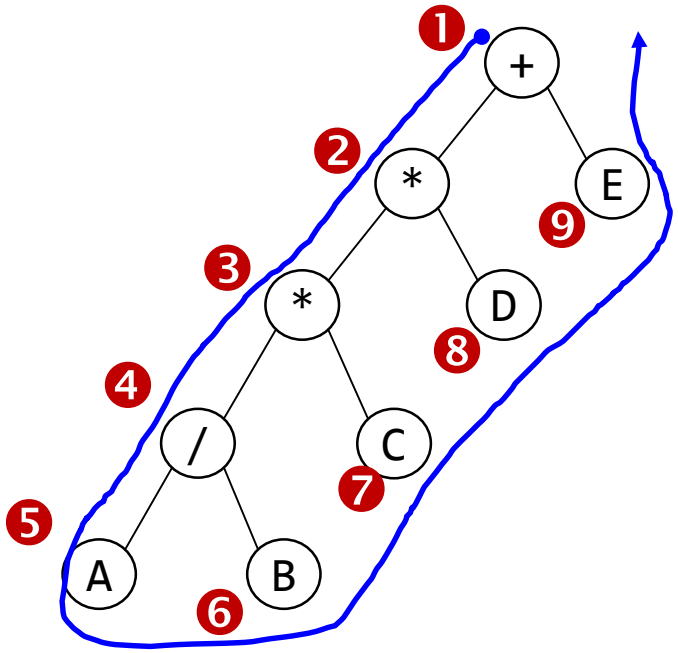
Call of inorder	Value in root	Action	inorder	in root	Value Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	**printf**
4	/		13	NULL	
5	A		2	*	**printf**
6	NULL		14	D	
5	A	**printf**	15	NULL	
7	NULL		14	D	**printf**
4	/	**printf**	16	NULL	
8	B		1	+	**printf**
9	NULL		17	E	
8	B	**printf**	18	NULL	
10	NULL		17	E	**printf**
3	*	**printf**	19	NULL	

출력 : A / B * C * D + E

전위순회(Preorder Traversal)

VRL : + * * / A B C D E

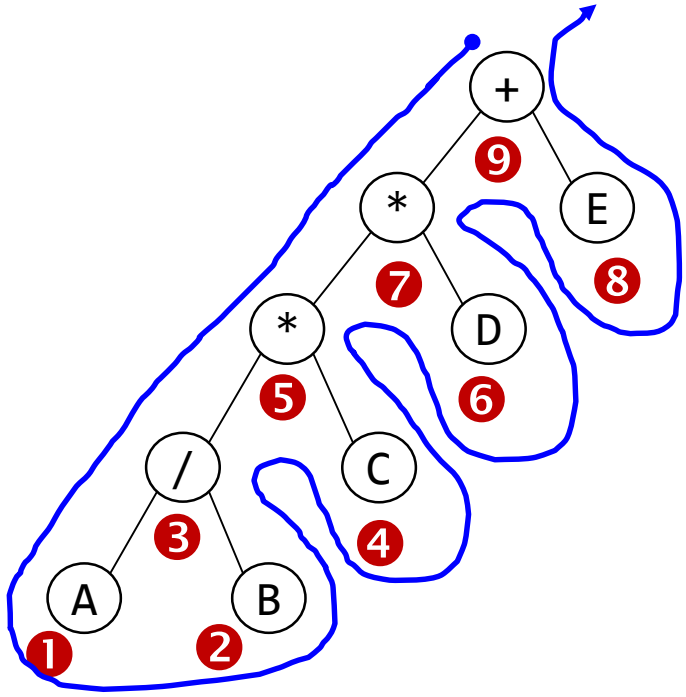
```
void preorder(treePointer ptr)
/* 전위 트리 순회 */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->leftChild);
        preorder(ptr->rightChild);
    }
}
```



후위순회(Postorder Traversal)

LRV : A B / C * D * E +

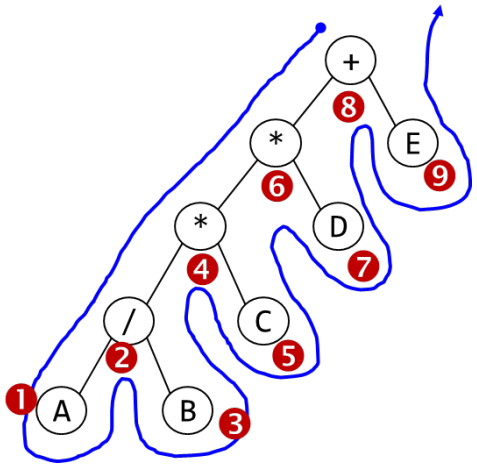
```
void postorder(treePointer ptr)
/* 후위 트리 순회 */
{
    if (ptr) {
        postorder(ptr->leftChild);
        postorder(ptr->rightChild);
        printf("%d", ptr->data);
    }
}
```



반복적 중위 순회(Iterative Inorder Traversal)

- 순환법(recursive) 방식이 아닌 반복법(iterative, ex: for loop)으로 동등한 함수 구현

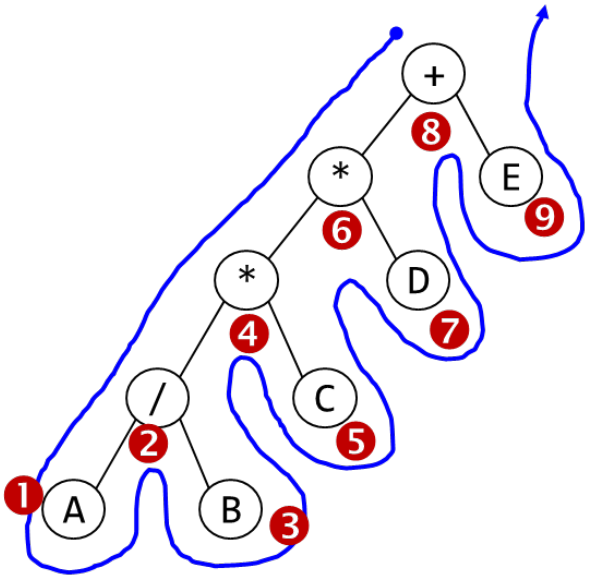
	Call of inorder	Value in root	Action	inorder	in root	Value Action
push() →	1	+		11	C	
push() →	2	*		12	NULL	
push() →	3	*		11	C	printf
push() →	4	/		13	NULL	
push() →	5	A		2	*	printf
	6	NULL		14	D	
pop() →	5	A	printf	15	NULL	
	7	NULL		14	D	printf
pop() →	4	/	printf	16	NULL	
push() →	8	B		1	+	printf
	9	NULL		17	E	
pop() →	8	B	printf	18	NULL	
	10	NULL		17	E	printf
pop() →	3	*	printf	19	NULL	



stack을 이용하여 구현

반복적 중위 순회(Iterative Inorder Traversal)

- 중위 순회를 위한 **비순환 프로그램**



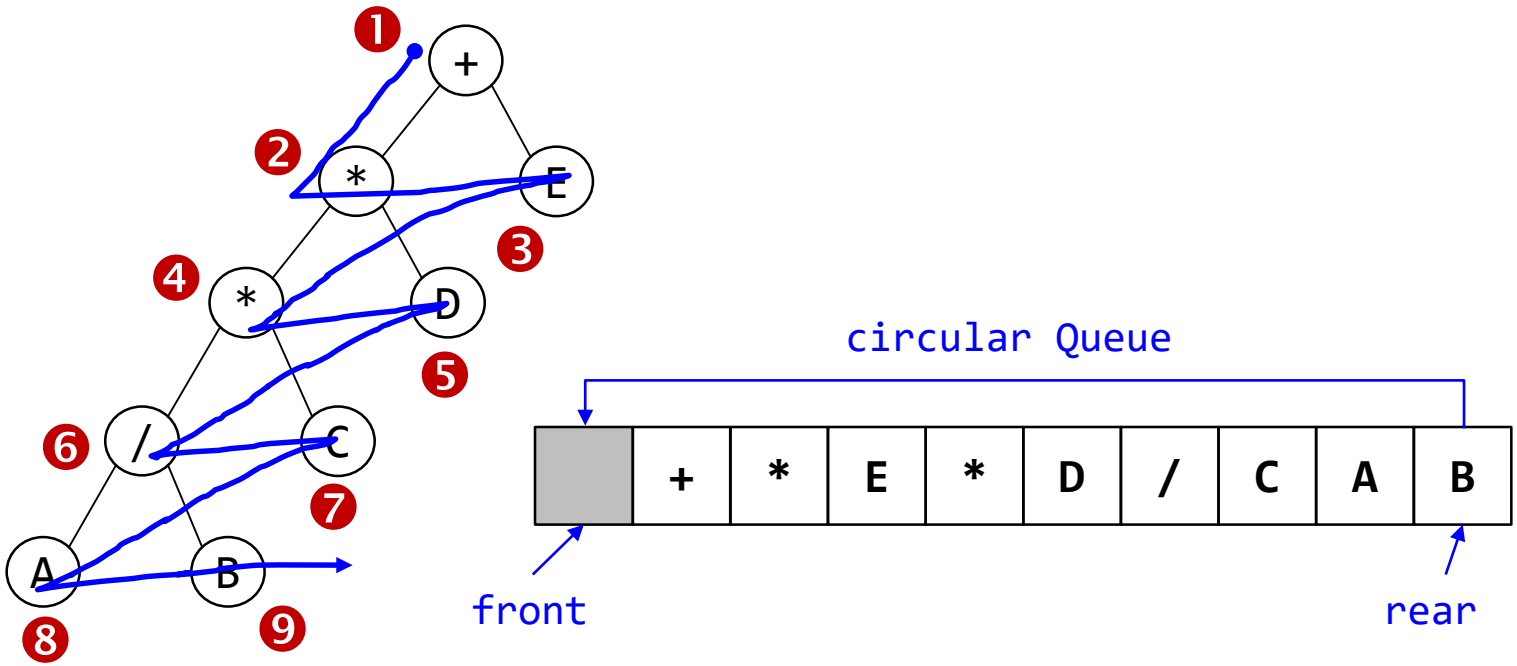

```
void iterInorder(treePointer node)
{
   int top = -1; /* 스택 초기화 */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
        for (; node; node = node->leftChild)
                 push(node); /*스택에 삽입 */
        node = pop();       /*스택에서 삭제 */

        if (!node) break;  /* 공백 스택 */

        printf("%d", node->data);
        node = node->rightChild;
   }
}
```

Time Complexity: O(n)

Space Complexity: O(n)

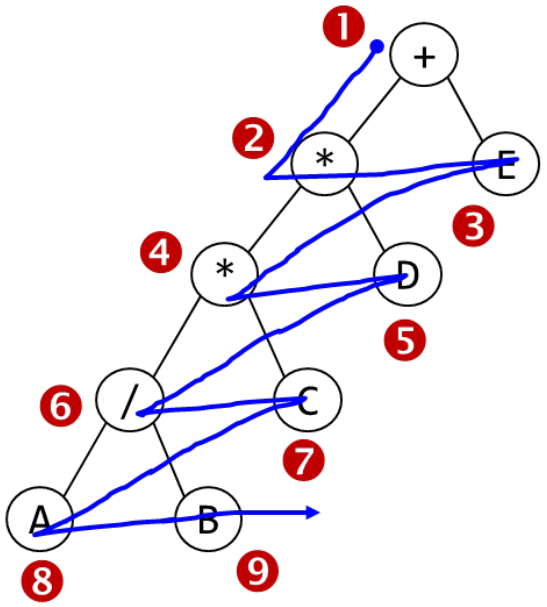
레벨순서 순회(Level-order Traversal)

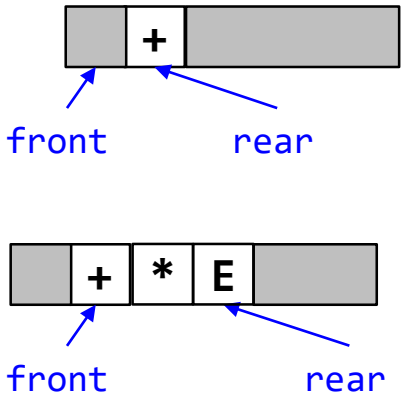
- **큐 사용** ← Inorder traversal의 iterative 방식은 stack 이용
- **루트 방문 → 왼쪽 자식 방문 → 오른쪽 자식 방문**
- **+ * E * D / C A B**



레벨순서 순회(Level-order Traversal)

```
void levelOrder(treePointer ptr)
/* 레벨 순서 트리 순회 */
{
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];

    if (!ptr) return; /* 공백 트리 */
    addq(front, &rear, ptr);

    for ( ; ; ) {
        ptr = deleteq(&front, rear);
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                addq(front, &rear, ptr->leftChild);
            if (ptr->rightChild)
                addq(front, &rear, ptr->rightChild);
        }
        else break;
    }
}
```





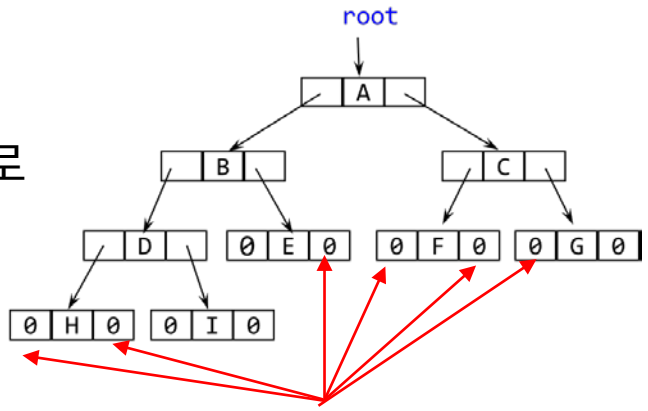
front위치 확인

스택 없는 순회(Traversal without a Stack)

(Question) 스택을 위한 추가적인 공간을 사용하지 않고
이진트리 순회가 가능한가?

- **방법 1) 각 노드에 parent(부모)필드 추가**
  - 스택을 사용하지 않아도 루트 노드로 올라갈 수 있음

- **방법 2) 스레드(thread)이진 트리로 표현**
  - 각 노드마다 두 비트 필요 (정보)
  - leftChild필드와 rightChild 필드를 루트로 돌아갈 수 있는 경로를 유지하도록 사용
  - 경로 상의 주소 스택은 리프 노드에 저장



NULL pointer 재활용

이진트리의 추가연산
(Additional Binary Operations)

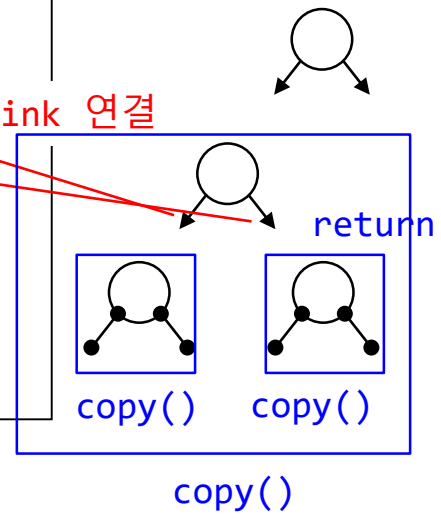
이진트리의 복사(Copying Binary Trees)

```
treePointer copy(treePointer original)
/* 주어진 트리를 복사하고 복사된 트리의 treePointer를 반환한다. */
{
    treePointer temp;
    if (orginal) {
        temp = (treePointer) malloc(sizeof(node));
        if ( IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->leftChild = copy(original->leftChild);
        temp->rightChild = copy(original->rightChild);
        temp->data = original->data;
        return temp;
    }
    return NULL;
}
```

← tree node 할당

link copy→

data copy→



동일성 검사(Testing Equality)

- 두 이진 트리가 동일한지 검사

```
int equal(treePointer first, treePointer second)

{/* 두 이진 트리가 동일하면 TRUE, 그렇지 않으면 FALSE를 반환한다. */

        return ((!first && !second) ||
                (first && second && (first->data == second->data) &&
                 equal(first->leftChild, second->leftChild) &&
                 equal(first->rightChild, second->rightChild));

}
```

OR

둘다 NULL이면 → TRUE

first != NULL
second != NULL

first, second의 값은?

마지막 노드에 다달아서 first, second == NULL일 때 TRUE 리턴

만족성 문제(The Satisfiability Problem)

- **변수 x_1, x_2, …, x_n 과 연산자 ∧, ∨, ¬으로 이루어진 식**
  - 변수는 2개의 값만을 가질 수 있음 → {참, 거짓}

- **규칙**

  (1) 변수도 하나의 식

  (2) x, y가 식일때, ¬x, x∧y, x∨y도 식

  (3) 연산순서는 ¬, ∧, ∨ 순, 괄호 사용

- **명제 해석식(formula of propositional calculus)**
  - 위 규칙을 이용해서 구성한 식

$$x_1 \vee (x_2 \wedge \neg x_3)$$

만족성 문제(The Satisfiability Problem)

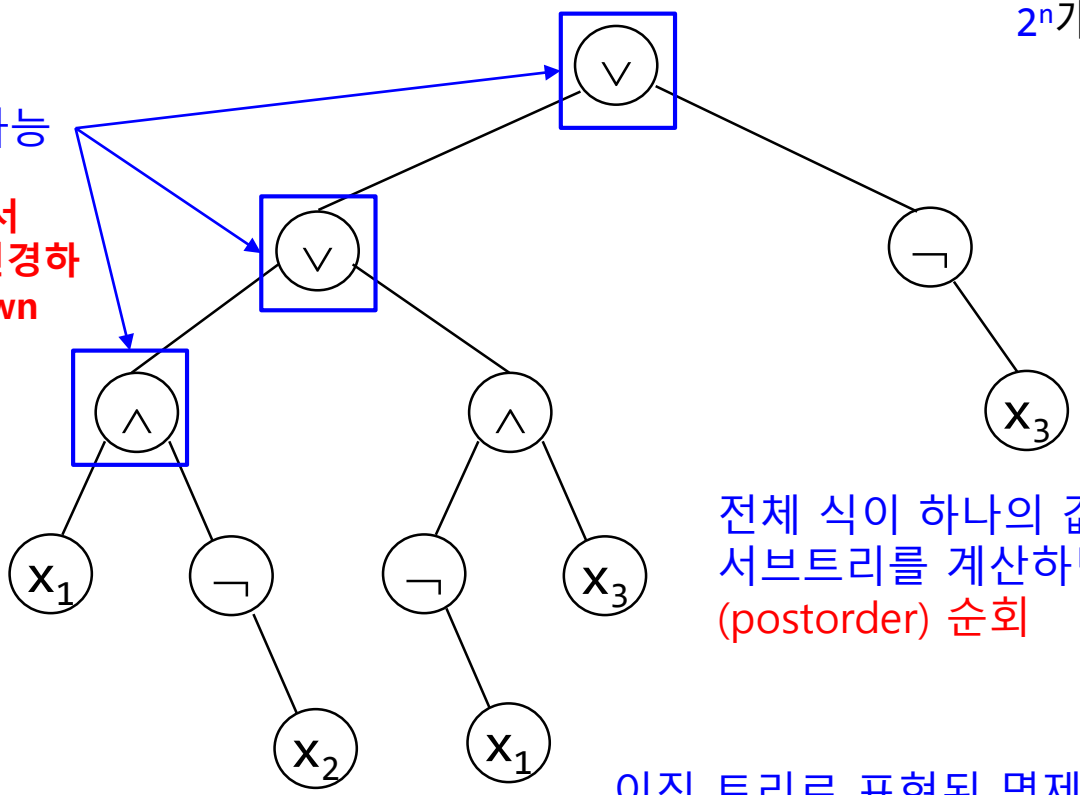
- **명제식의 만족성(satisfiability) 문제**
  - 식의 값이 true가 되도록, 변수에 값을 지정할 수 있는 방법이 있는가?

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

n개의 변수에 대해 2^n개의 조합이 가능

교환법칙 적용 가능

Compiler, Database에서 Optimal Tree 구조로 변경하여 Process → Cost down



전체 식이 하나의 값이 될 때까지 서브트리를 계산하면서 후위순위(postorder) 순회

이진 트리로 표현된 명제식

만족성 문제(The Satisfiability Problem)

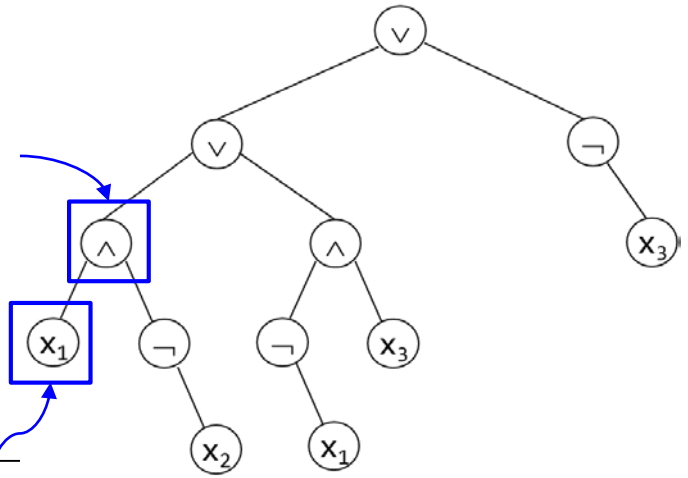
- 만족성 문제를 위한 노드 구조

leftChild	data	value	rightChild

```
typedef enum {not, and, or, true, false} logical;
typedef struct node *treePointer;
typedef struct node {
        treePointer leftChild;
        logical data;
        short int value;
        treePointer rightChild;
        };
```

← 변수의 현재 값이나 명제식의 연산자

true, false 값→



data == value
(true or false)

만족성 문제(The Satisfiability Problem)

- **만족성 알고리즘의 첫 번째 버전**
  - 리프의 data는 이 노드가 나타내는 변수의 현재 값을 가짐
  - root가 n개의 변수를 갖는 명제식의 트리를 가리킴

```
for (all 2^n possible combinations)
{
        generate the next combination;
        replace the variables by their values;
        evaluate root by traversing it in postorder;
        if (root->value) {
                printf(<combination>);
                return;
        }
}
printf("No satisfiable combination\n");
```

Time Complexity?

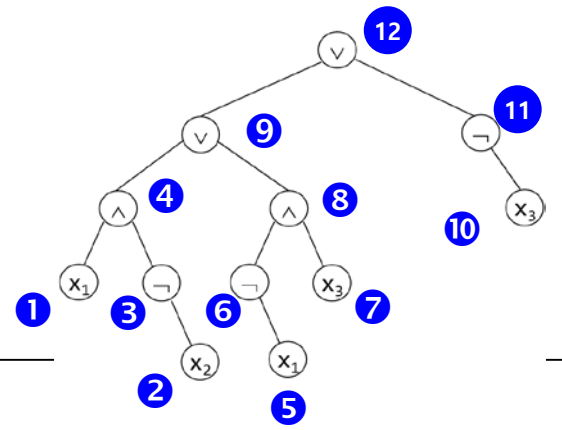
Reasonable?

만족성 문제(The Satisfiability Problem)

- 후위 순회 연산 함수

```
void postOrderEval(treePointer node)
{/* 명제 해석 트리를 계산하기 위해 수정된 후위 순회 */
    if (node) {
        postOrderEval (node->leftChild);
        postOrderEval (node->rightChild);
        switch(node->data) {
            case not:
                node->value = !node->rightChild->value; break;
            case and:
                node->value = node->rightChild->value &&
                              node->leftChild->value; break;
            case or:
                node->value = node->rightChild->value ||
                              node->leftChild->value; break;
            case true:
                node->value = TRUE; break;
            case false:
                node->value = FALSE;
        }
    }
}
```

순환적 후위순회
(재귀호출,
recursive call)

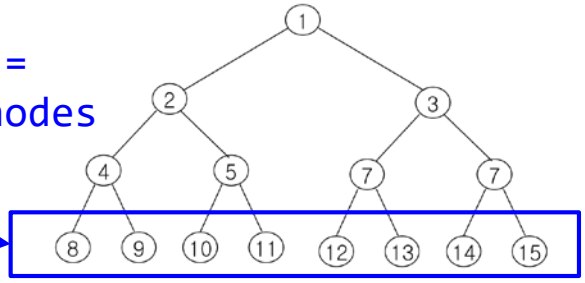


스레드 이진 트리
(Threaded Binary Trees)

스레드(Threads)

- n 노드 이진 트리의 연결 표현
  - 총 링크의 수 : 2n
  - 널 링크의 수 : n+1

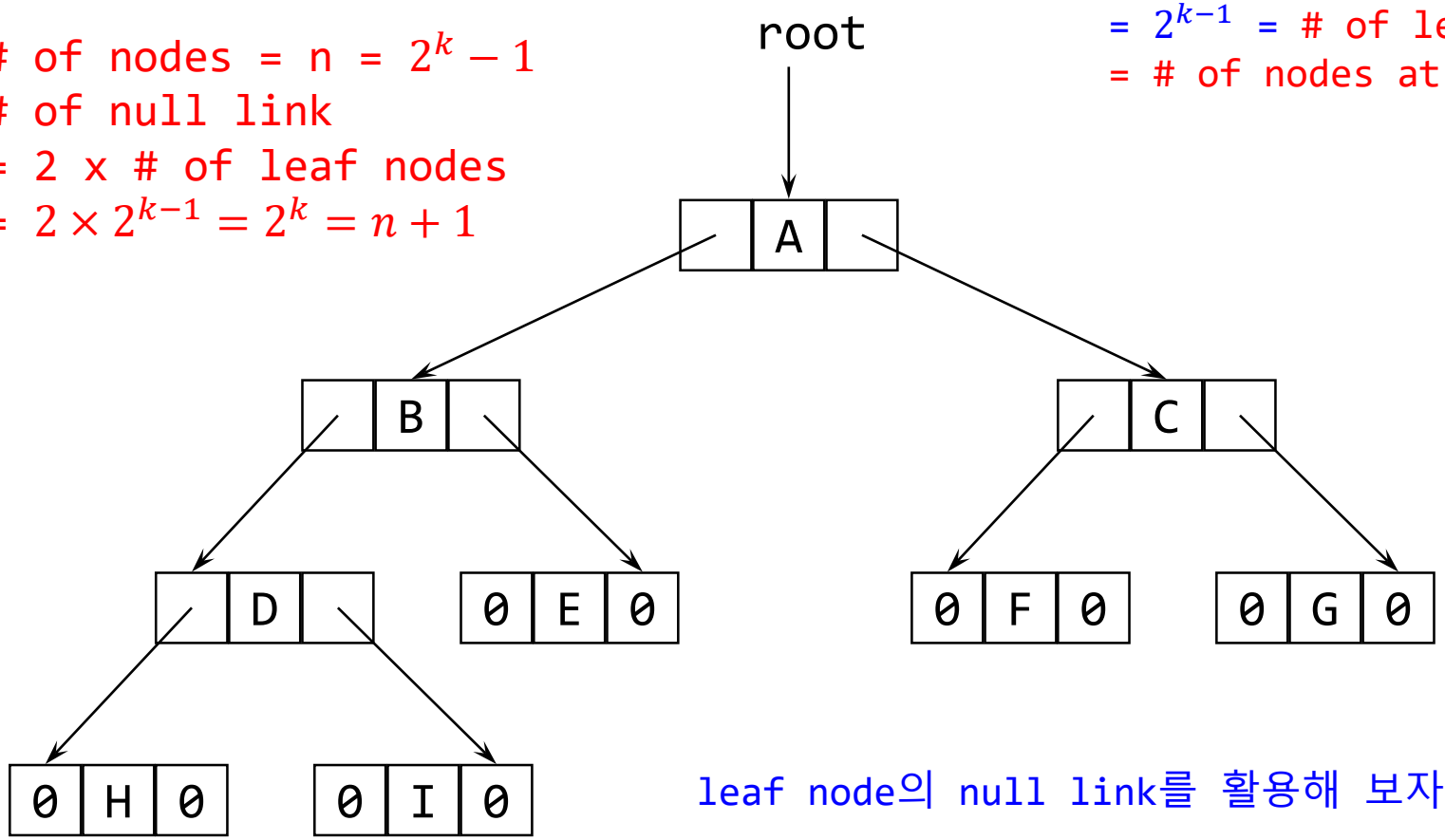
worst case =
# of leaf nodes



full binary tree

# of nodes = n = $2^k - 1$
# of null link
= 2 x # of leaf nodes
= $2 \times 2^{k-1} = 2^k = n + 1$

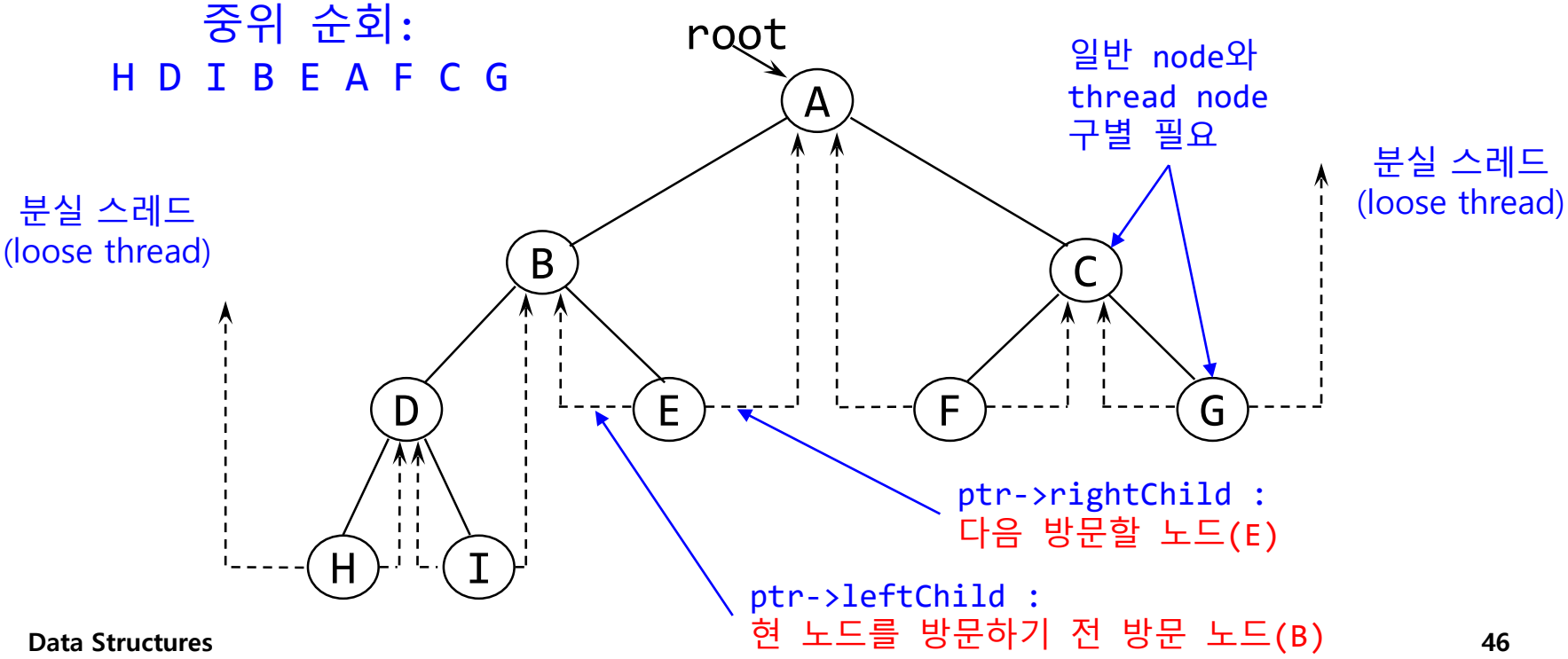
= 2^{k-1} = # of leaf nodes
= # of nodes at level k



leaf node의 null link를 활용해 보자는 개념

스레드(Threads)

- **스레드(Thread) 구성 방법**
  - 널 링크 필드를 다른 노드를 가리키는 포인터로 대치
  - if ptr->rightChild == NULL,
    ptr->rightChild = ptr의 중위 후속자에 대한 포인터
  - if ptr->leftChild == NULL,
    ptr->leftChild = ptr의 중위 선행자에 대한 포인터



스레드(Threads)

```
typedef struct threadedTree *threadedPointer;
typedef struct threadedTree {
    short int leftThread;
    threadedPointer leftChild;
    char data;
    threadedPointer rifhtChild;
    short int rightThread; }
```

2개 필드 추가

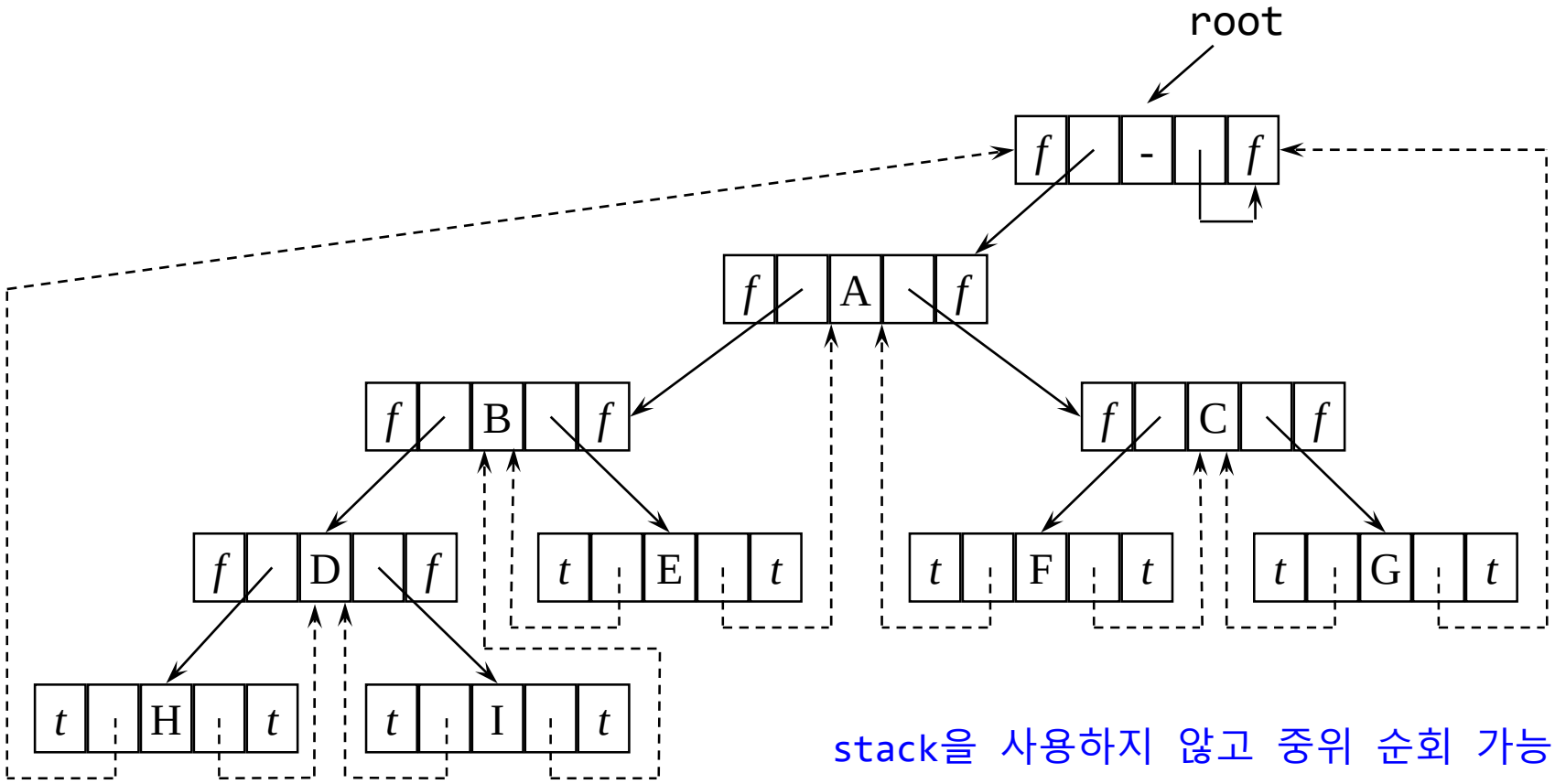
- **노드 구조**

  - leftThread == false if leftChild가 포인터
    == true if leftChild가 스레드
  - rightThread == false if rightChild가 포인터
    == true if rightChild가 스레드

<공백 다항식의 헤드노드와 유사>

- **헤드 노드(header node)**
  - 분실 스레드(loose thread) 문제 해결 → 헤드노드를 가리키도록
  - root가 헤드노드(header node)를 가리키도록 → root->leftChild가 첫 번째 노드

스레드 이진 트리의 메모리 표현
(Memory Representation of Threaded Tree)



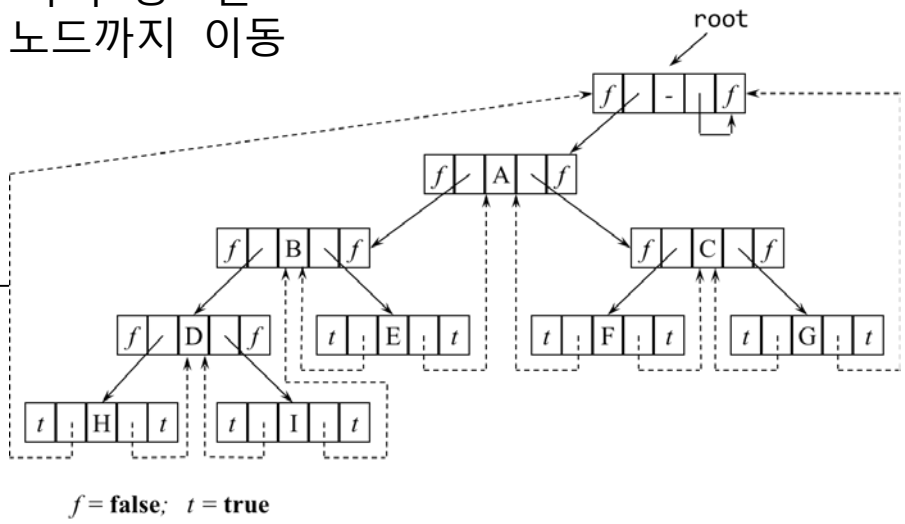
f = **false**; t = **true**

stack을 사용하지 않고 중위 순회 가능

ptr->rightThread = true 이면,
다음 방문할 노드 = ptr->rightChild

스레드 이진 트리의 중위 순회
(Inorder Traversal of a Threaded Binary Tree)

- x->rightThread
  - == true : x->rightChild
  - == false : 오른쪽 자식의 왼쪽 자식 링크를 따라 가서 leftThread==true인 노드까지 이동



스레드 이진 트리에서 중위 후속자의 탐색

```
threadedPointer insucc(threadedPointer tree)
{
        threadedPointer temp;
        temp = tree->rightChild;
        if (!tree->rightThread)      ← rightThread = false
                while (!temp->leftThread)     ← 마지막 leftChild 까지 이동
                        temp = temp->leftChild;
        return temp;   ← 다음 insucc()이 호출될 때는
                         temp->rightChild로 이동
}
```

스레드 이진 트리의 중위 순회
(Inorder Traversal of a Threaded Binary Tree)

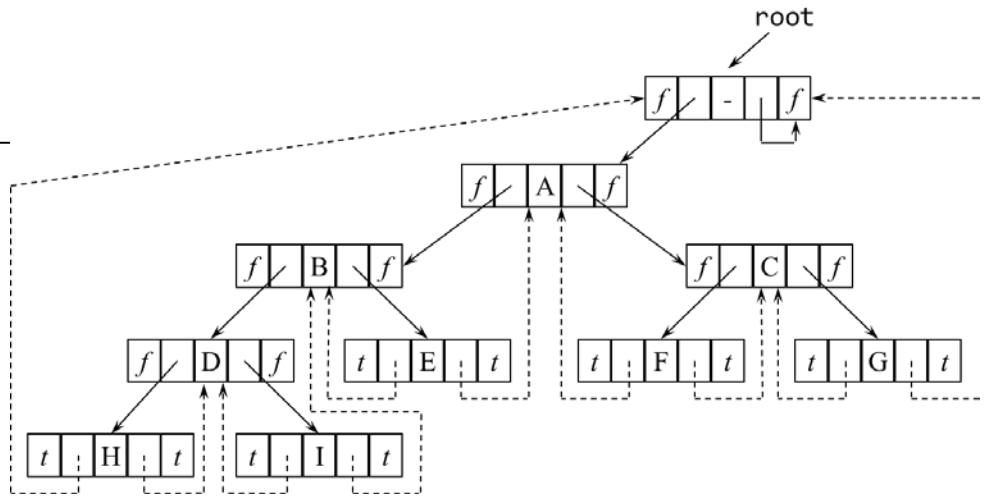
- 스레드 이진 트리에서 중위 순회

```
void tinorder(threadedPointer tree)
{/* 스레드 이진 트리의 중위 순회 */
        threadedPointer temp = tree;
        for ( ; ; ) {
                temp = insucc(temp);
                if (temp == tree) break;
                printf("%3c", temp->data);
        }
}
```

root인가?

```
insucc() ...
temp = tree->rightChild;
if (!tree->rightThread)
```

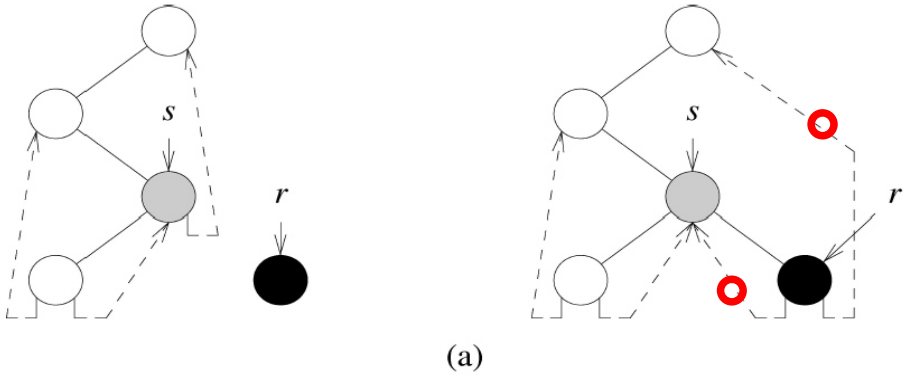
2번째 insucc() call 때
temp = D
tree = H



스레드 이진 트리에서의 노드 삽입
(Inserting a Node into a Threaded Binary Tree)

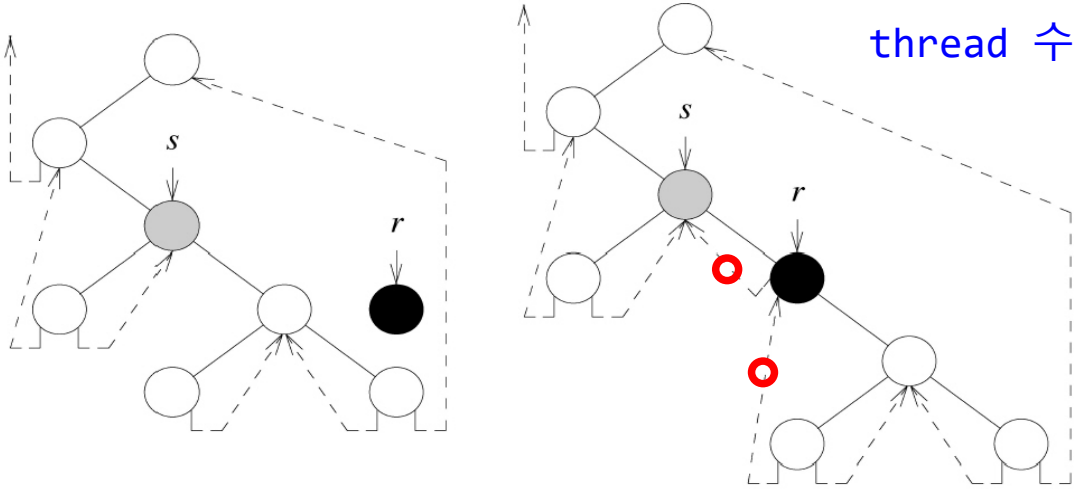
s의 오른쪽 자식으로 r을 삽입

s의 오른쪽 자식이 null 일 때



(a)

s의 오른쪽 자식이 null이 아닐 때



thread 수정 필요

before

(b)

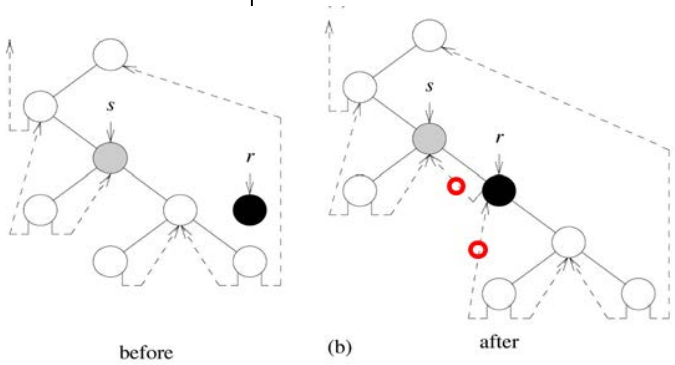
after

스레드 이진 트리에서의 노드 삽입(2)

- s의 오른쪽 자식으로 r을 삽입

```
void insertRight(threadedPointer s, threadedPointer r)
{/* 스레드 이진 트리에서 r을 s의 오른쪽 자식으로 삽입 */
    threadedPointer temp;
    r->rightChild = parent->rightChild;
    r->rightThread = parent->rightThread;
    r->leftChild = parent;
    r->leftThread = TRUE;
    s->rightChild = child;
    s->rightThread = FALSE;
    if (!r->rightThread) {
        temp = insucc(r);
        temp->leftChild = r;
    }
}
```

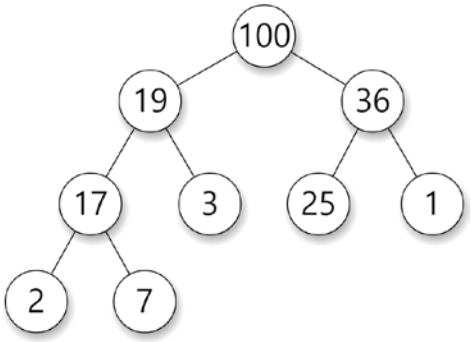
← s->rightChild = r;



‘쌓아 올린다’

힙
(Heaps)

In computer science, a heap is a specialized tree-based data structure which is essentially an almost complete tree that satisfies the heap property: if P is a parent node of C, then the key (the value) of P is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the key of C. --- wikipedia



https://en.wikipedia.org/wiki/Heap_(data_structure)

우선순위 큐(Priority Queue)

heap은 우선순위 큐 구현에 사용

- 우선순위가 가장 높은(낮은) 원소를 먼저 삭제
- 임의의 우선순위를 가진 원소 삽입 가능
- 추상 데이타 타입 MaxPriorityQueue

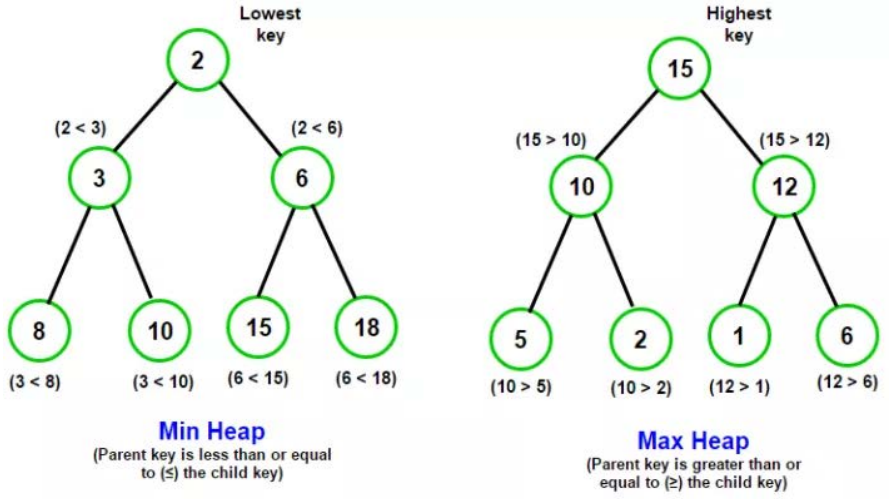
```
ADT MaxPriorityQueue is
    objects: a collection of n > 0 elements, each element has a key
    functions:    for all q ∈ MaxPriorityQueue, item ∈ Element, n ∈ integer

        MaxPriorityQueue create(max_size)::= create an empty priority queue.

        Boolean isEmpty(q, n)      ::= if(n > 0) return ~~TRUE~~ FALSE;
                                        else return ~~FALSE~~ TRUE;
        Element top(q, n)          ::= if(!isEmpty(q,n)) return an instance
                                         of the largest element in q
                                       else return error.
        Element pop(q, n)          ::= if(!isEmpty(q,n)) return an instance
                                       of the largest element in q and
                                       remove it from the heap else return error.
        MaxPriorityQueue push(q, item, n) ::= insert item into q and return the
                                              resulting priority queue.
```

우선순위 큐(Priority Queue)

- **무순서(unordered) 선형 리스트로 구현 (가장 간단한 방법)**
  - IsEmpty() : O(1) <-- linked list가 null 인지 검사
  - push() : O(1) <-- 임의의 위치에 노드 삽입
  - top() : O(n) <-- list 전체 검사 (n개 노드)
  - pop() : O(n) <-- top()을 검사 후 삭제 (n개 노드)

- **최대 힙(max heap)로 구현**
  - IsEmpty() : O(1)
  - top() : O(1)
  - push() : O(log n)
  - pop() : O(log n)

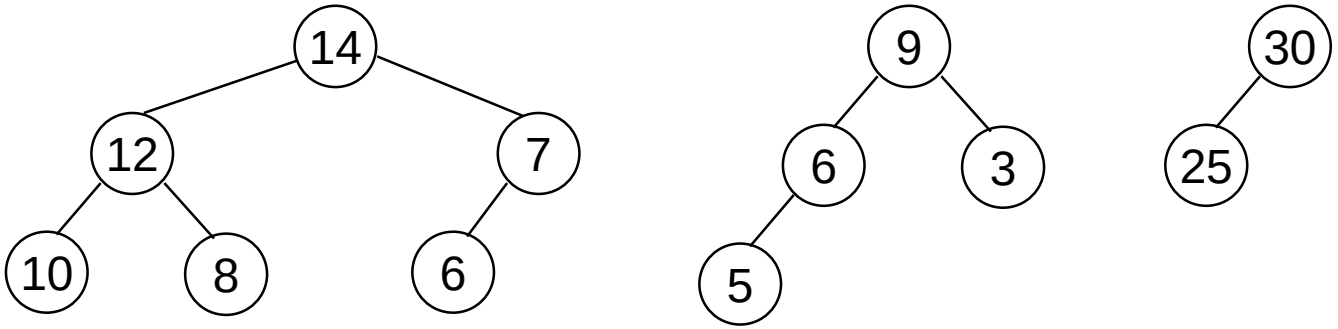
$n=2^k-1$



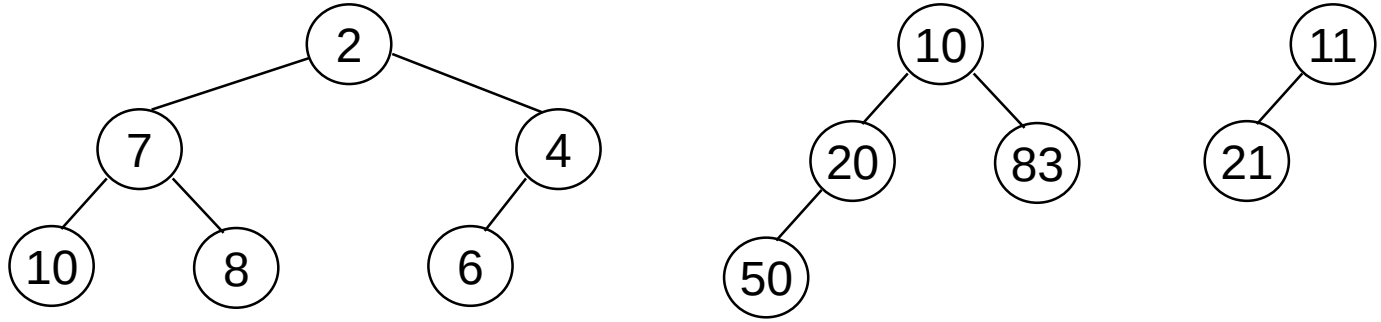
최대 힙의 정의(Definition of a Max Heap)

- 최대(최소)트리: 각 노드의 키 값 그 자식의 키 값보다 작지(크지) 않은 트리.
- 최대 힙: 최대 트리이면서 완전 이진 트리(complete binary tree).
- 최소 힙: 최소 트리이면서 완전 이진 트리.

최대 힙
(max heap)

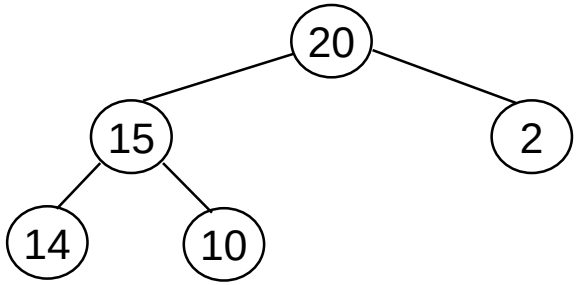


최소 힙
(min heap)

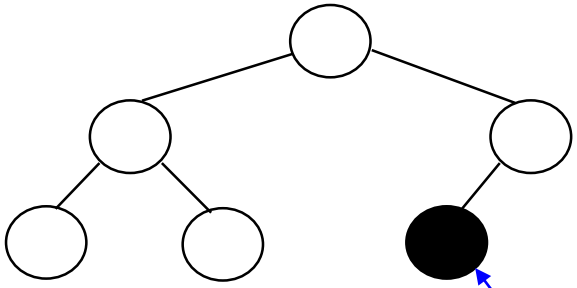


최대 힙에서의 삽입(Insertion into a Max Heap)

- **삽입 후에도 최대 힙 유지 : push()**
  - 새로 삽입된 원소는 부모 원소와 비교하면서 최대 힙이 되는 것이 확인될 때까지 위로 올라감



(a)

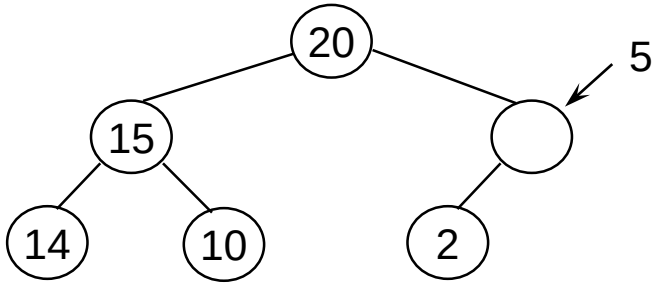


(b)

n=6 일 때
완전 이진 트리 형태

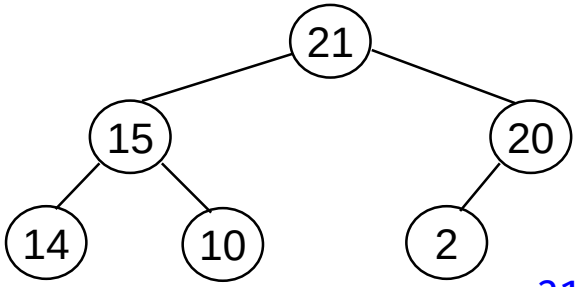
완전이진트리
마지막 노드부터
bubbling up

5 삽입



(c)

2<-->5
switch



(d)

Time Complexity

O(log n)

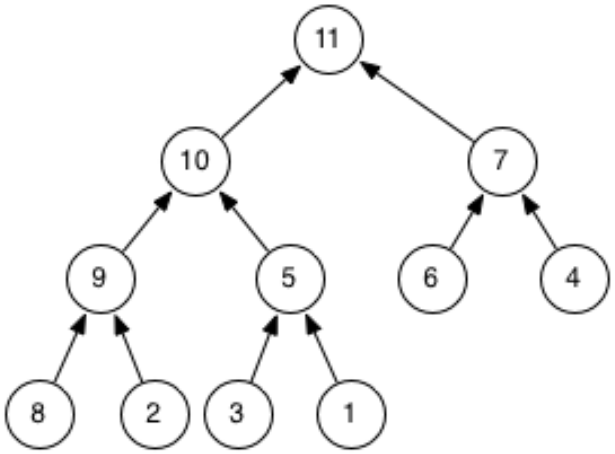
5 대신
21이 삽입된 경우

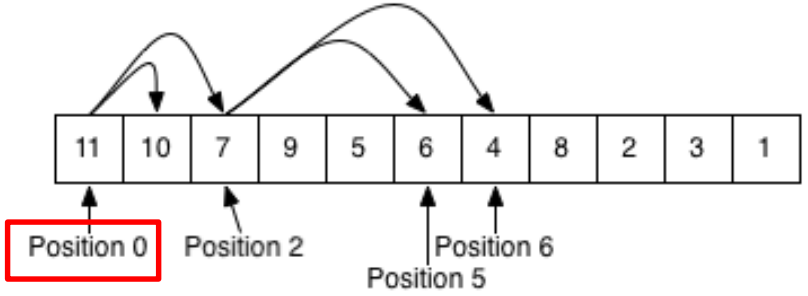
최대 힙에서의 삽입(Insertion into a Max Heap)

```
#define MAX_ELEMENTS 200 /* 최대 힙 크기 + 1 */
#define HEAP_FULL(n) (n == MAX_ELEMENT - 1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* 다른 필드들 */
        } element;

element heap[MAX_ELEMENTS]
int n = 0;
```

보조정리 5.4(p. 217)

node i의 부모의 위치
= parent(i) = $\lfloor i/2 \rfloor$





Index가 0부터 시작할 경우

parent(i) = $\lfloor (i-1)/2 \rfloor$

최대 힙에서의 삽입(Insertion into a Max Heap)

```
void push(element item, int *n)
{/* insert item into a max heap of current size *n */
        int i;
        if (HEAP_FULL(*n)){
                fprintf(stderr, "The heap is full. \n");
                exit(EXIT_FAILURE);
        }

        i = ++(*n);
        while ((i != 1) && (item.key > heap[i/2].key)){
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i] = item;
}
```

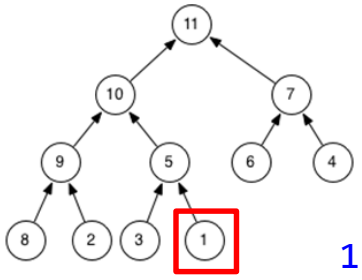
next slot →

부모의 key와 비교

부모보다 크다면 부모를 i위치로

자신의 위치를 부모의 위치 (i/2)로 변경

0	1	2	3	4	5	6	7	8	9	10	11
	11	10	7	9	5	6	4	8	2	3	1

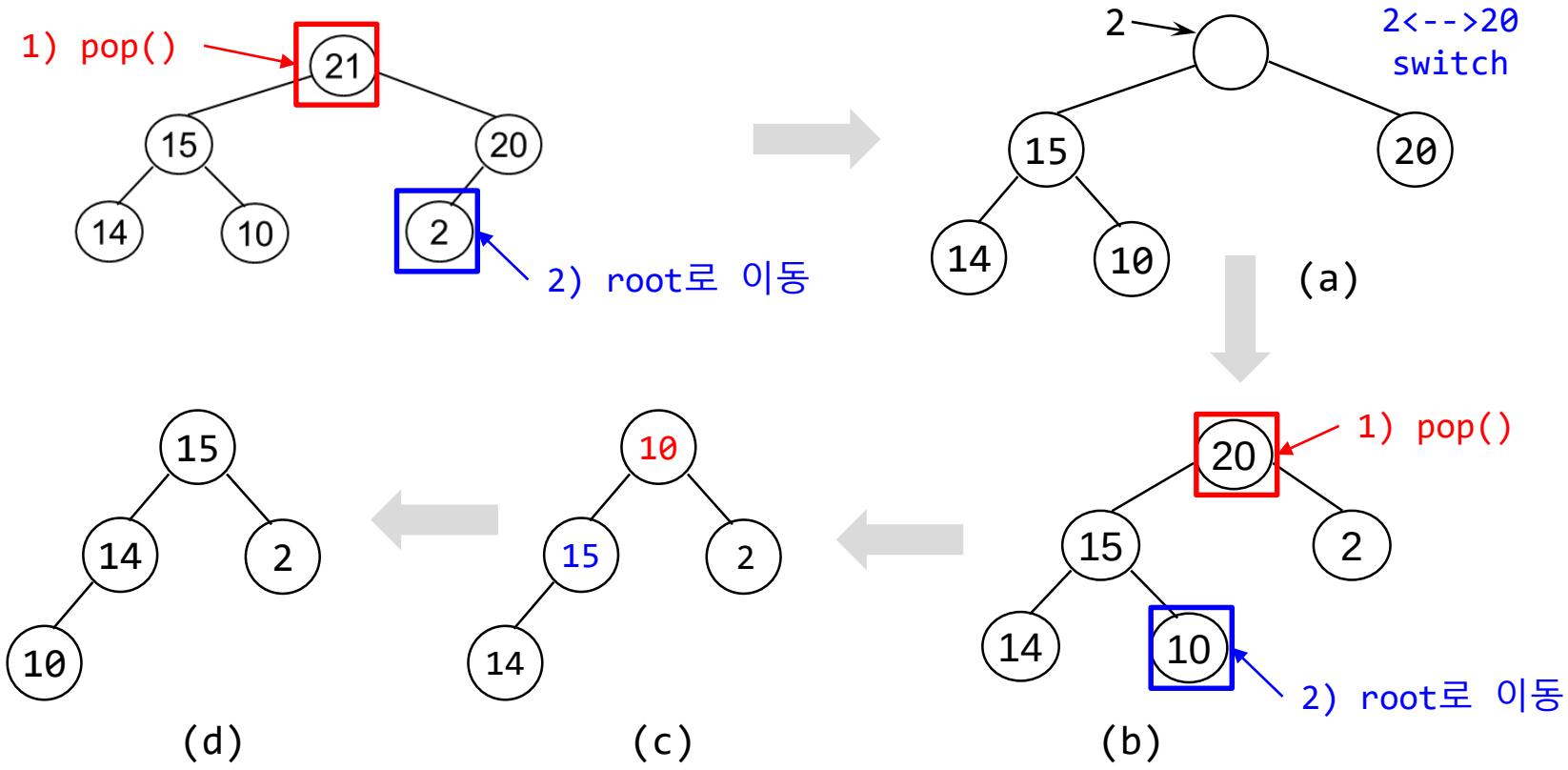


Time Complexity : $O(\log_2 n)$

1 대신 7이 입력될 경우는 어떻게 되는가?

최대 힙에서의 삭제(Deletion from a Max Heap)

- **루트 삭제 : pop()**
  - 루트와 마지막 위치의 노드 교환 후 완전 이진 트리로 재구성
  - 루트와 왼쪽 자식, 오른쪽 자식 비교 → 가장 큰 것을 루트로



20이 삭제된 후
완전 이진 트리 형태

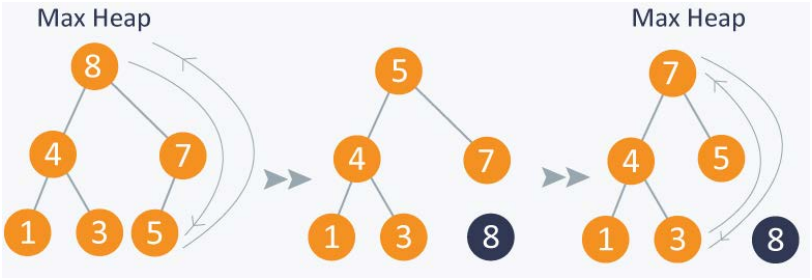
최대 힙에서의 삭제(Deletion from a Max Heap)

```
element pop(int *n)
{       int parent, child;
        element item, temp;
        if(HEAP_EMPTY(*n)) {
            fprintf(stderr, "The heap is empty\n");
            exit(EXIT_FAILURE);
        }
  root → item = heap[1];
마지막 노드 → temp = heap[(*n)--];
        parent = 1;
        child = 2;
        while(child <= *n){
                if (child < *n) && (heap[child].key < heap[child+1].key)
                        child++;  /* right child쪽에서 max를 찾음*/
                if(temp.key >= heap[child].key) break;
                heap[parent] = heap[child];  ← 현재 level에서 가장 큰 값을 부모로
                parent = child;              ← 현재 level을 parent로
                child *= 2;        /* 다음 level의 leftchild */
        }
        heap[parent] = temp;
        return item;
}
```

left child or
right child 결정 →

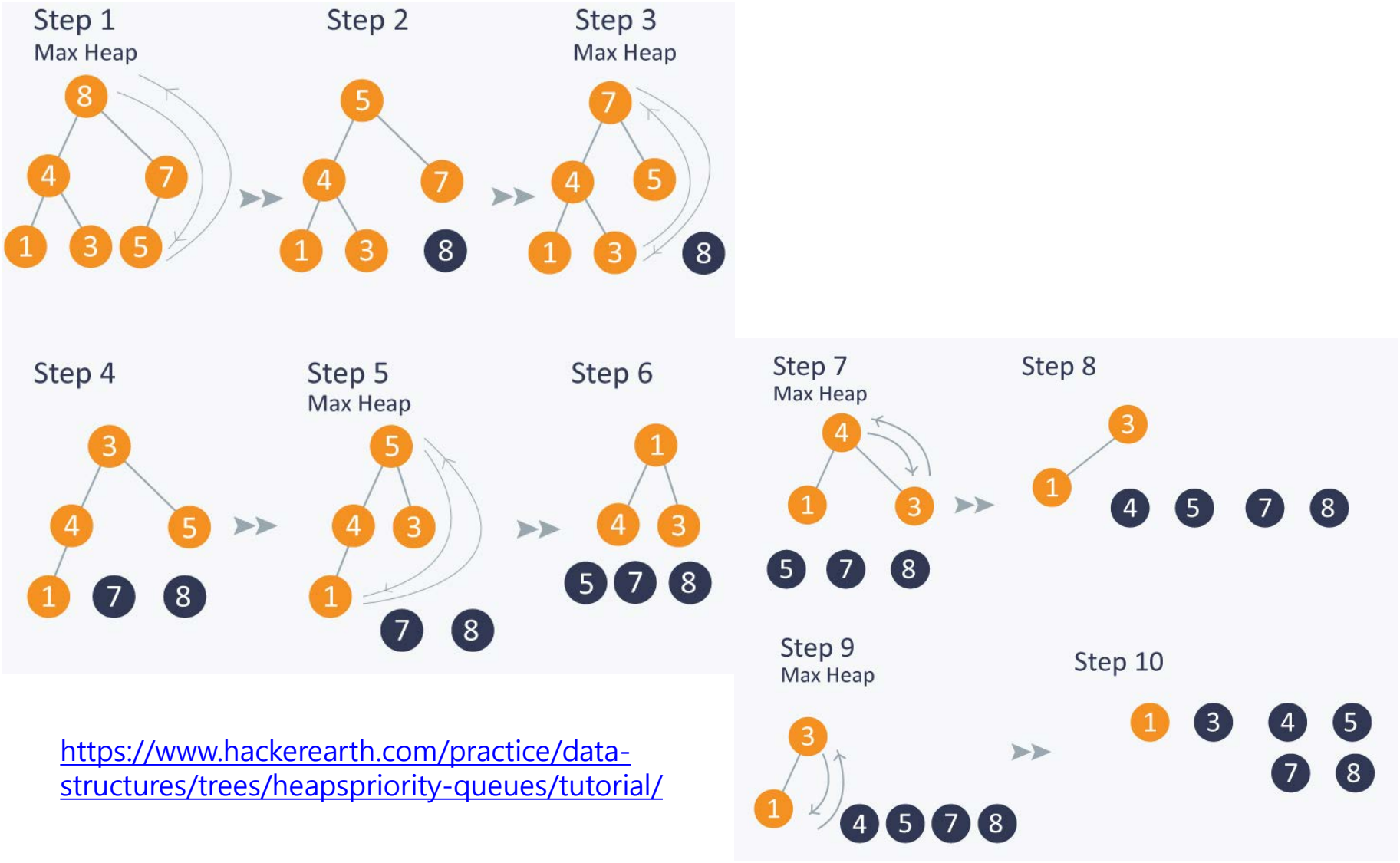
Heap[] n=6

0	1	2	3	4	5	6
	8	4	7	1	3	5



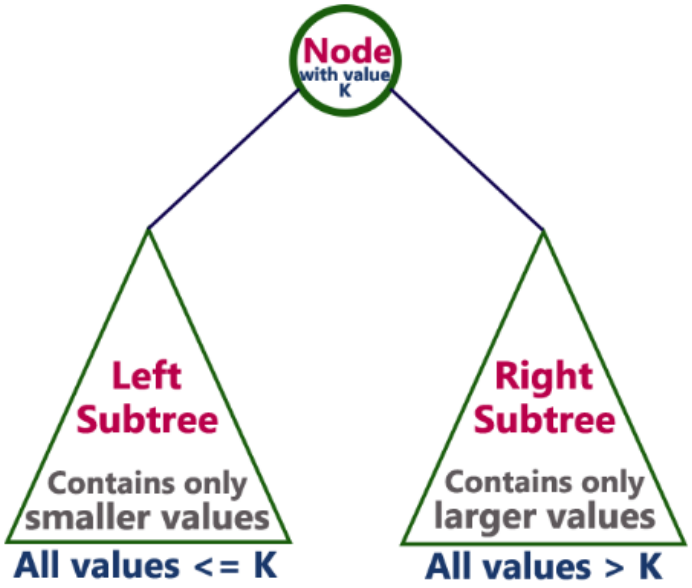
Time Complexity: O(log n)

최대 힙에서의 삭제(Deletion from a Max Heap)



https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/tutorial/

이원 탐색 트리
(Binary Search Trees)



정의(Definition)

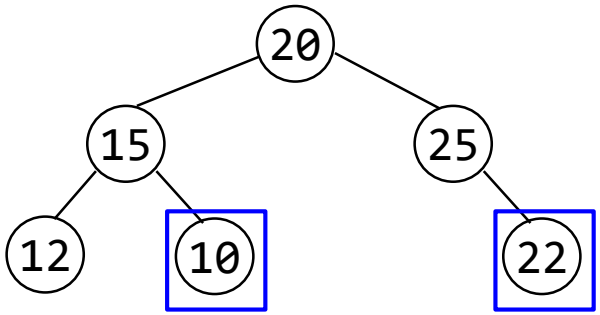
- **Example: 사전(dictionary)**
  - pair<키, 원소>의 집합

```
ADT Dictionary is
    objects: a collection of n > 0 pairs,
             each pair has a key and an associated item
    functions:
        for all d∈Dictionary, item ∈ Item, k ∈ Key, n∈ integer

        Dictionary Create(max_size)  ::= create an empty dictionary

        Boolean IsEmpty(d, n)    ::= if(n > 0) return ~~TRUE~~ FALSE
                                     else return ~~FALSE~~ TRUE

        Element Search(d, k)     ::= return item with key k,
                                     return NULL if no such element.

        Element Delete(d, k)     ::= delete and
                                     return item (if any) with key k;

        void Insert(d, item, k)  ::= insert item with key k into d.
```

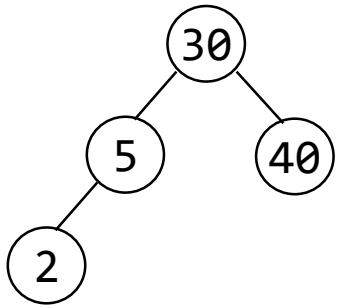
정의(Definition)

- **이원 탐색 트리는 공백가능하고, 만약 공백이 아니라면**
  (1) 모든 원소는 서로 상이한 키를 갖는다.
  (2) 왼쪽 서브트리의 키들은 그 루트의 키보다 작다.
  (3) 오른쪽 서브트리의 키들은 그 루트의 키보다 크다.
  (4) 왼쪽과 오른쪽 서브트리도 이원 탐색 트리이다.

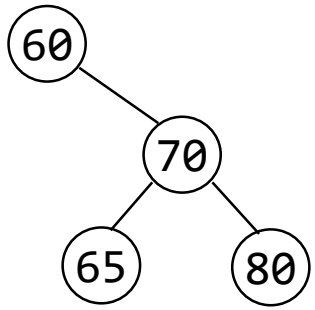
- **이원 탐색 트리 구별**



(a) X



(b) O



(c) O

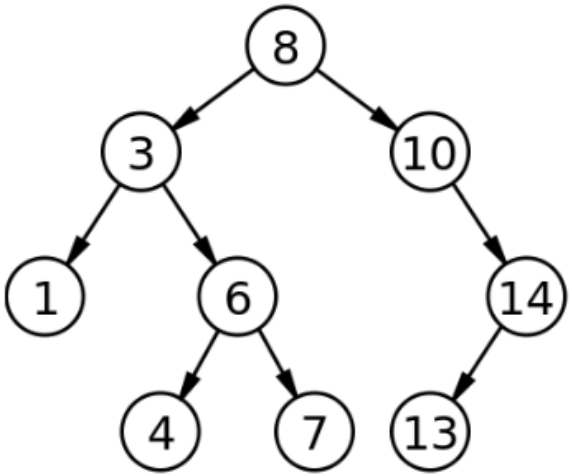
이원 탐색 트리의 탐색(Searching a Binary Search Tree)

```
typedef struct node *treePointer;
typedef struct node{
        int data;
        treePointer leftChild, rightChild;
        };
```

```
// struct node* search(struct node* tree, int key)

treePointer search(treePointer tree, int key)
{
    /* 키값이 key인 노드에 대한 포인터를 반환함.
       그런 노드가 없는 경우에는 NULL을 반환 */
        if (!tree) return NULL;
        if (key == tree->data) return tree;
        if (key < tree->data)
                return search(tree->left_child, key);
        return search(tree->right_child, key);
}
```

이원 탐색 트리의 탐색(Searching a Binary Search Tree)

- k = 루트의 키 : 성공적 종료
- k < 루트의 키 : 왼쪽 서브트리 탐색
- k > 루트의 키 : 오른쪽 서브트리 탐색



교재:
root → tree로 변경

```
element* search(treePointer tree, int key)
{
    /* 키값이 key인 노드에 대한 포인터를 반환함.
       그런 노드가 없는 경우에는 NULL을 반환 */
     if (!tree) return NULL;
     if (key == tree->data) return tree;
     if (key < tree->data)
             return search(tree->left_child, key);   ← recursive search
     return search(tree->right_child, key);          ← recursive search
}
```

Time Complexity = O(log n)

~~Time Complexity = O(h)~~

이원 탐색 트리의 반복적 탐색
(Iterative Search of a Binary Search Tree)

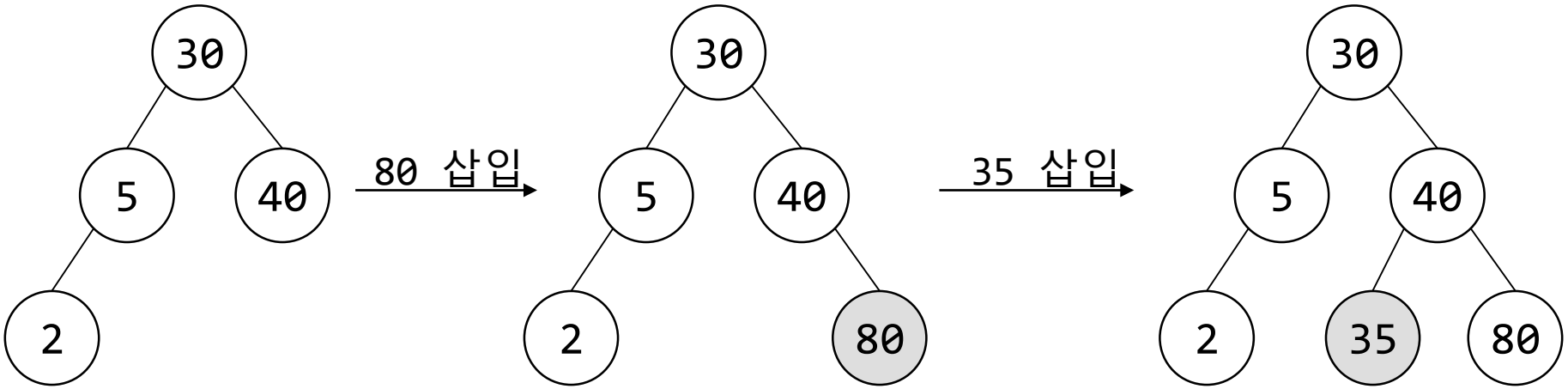
updated

```
treePointer ~~element*~~ iterSearch(treePointer tree, int key)
{
      /* 키값이 key인 노드에 대한 포인터를 반환함.
        그런 노드가 없는 경우는 NULL을 반환 */
        while (tree) {
                if (key == tree->data) return tree;
                if (key < tree->data)
                        tree = tree->left_child;
                else
                        tree = tree->right_child;
        }
        return NULL;
}
```

Time Complexity = O(log n)

이원 탐색 트리에서의 삽입
(Inserting into a Binary Search Tree)

- x의 key값을 가진 노드를 탐색
- 탐색이 성공하면 이 키에 연관된 원소를 변경: pair(key, data)
- 탐색이 실패하면 탐색이 끝난 지점에 삽입



이원 탐색 트리에서의 삽입
(Inserting into a Binary Search Tree)

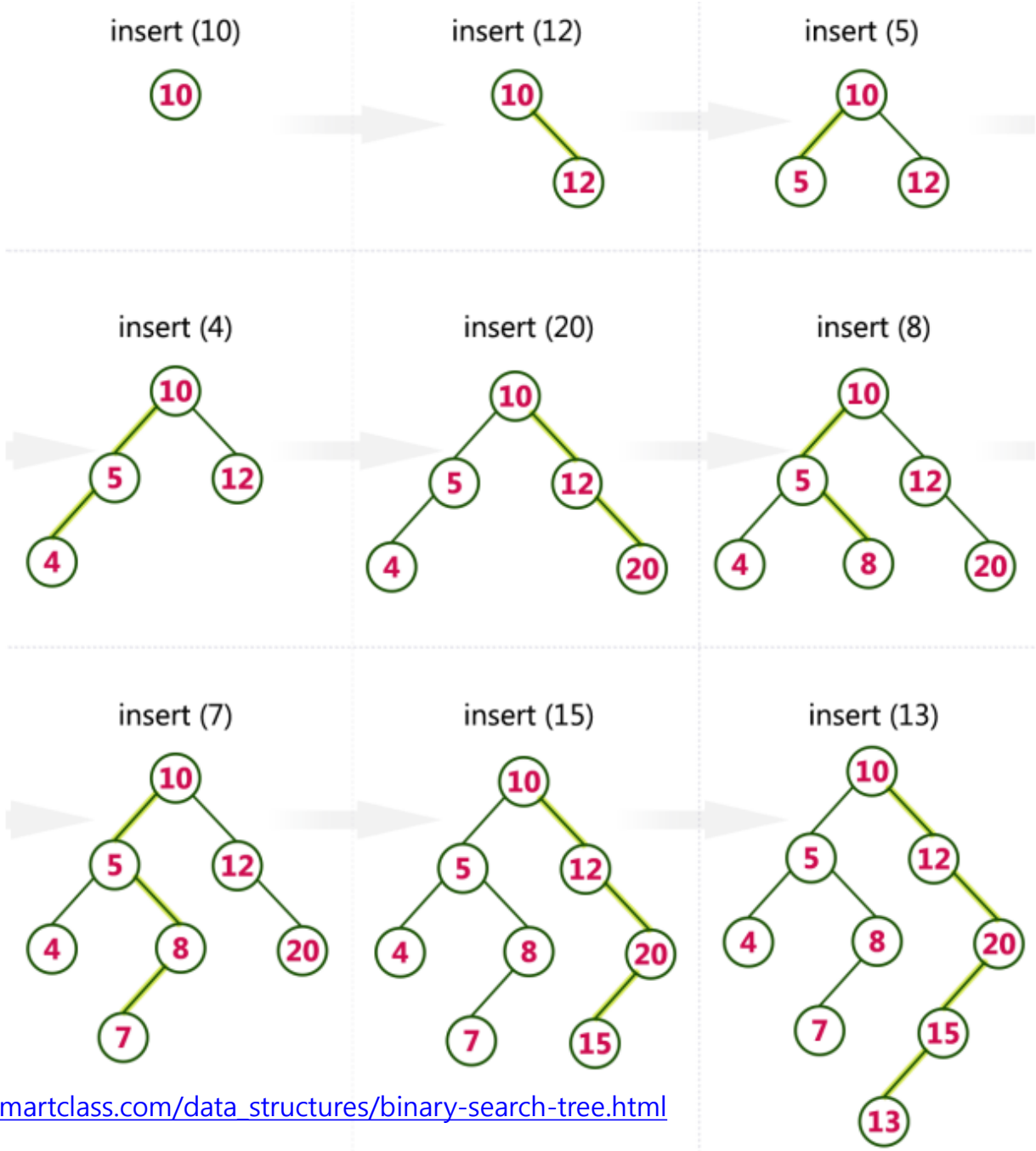
key에 대한 내용
(사전에서 단어에 대한 내용)

```
void insert (treePointer *node, int k, iType theItem)
{    /* 트리내의 노드가 k를 가리키고 있으면 아무 일도 하지 않음;
        그렇지 않은 경우는 data=(k, theItem)인 새 노드를 첨가 */

        treePointer ptr, temp = modifiedSearch(*node, k);
        if(temp || !(*node)) {
                /* k is not in the tree */
                MALLOC(ptr, sizeof(*ptr));
                ptr->data.key = k;
                ptr->data.item = theItem;
                ptr->leftChild = ptr->rightChild = NULL;
                if(*node) /* insert as child of temp */
                        if(k<temp->data.key) temp->leftChild = ptr;
                        else temp->rightChild = ptr;
                else *node = ptr;
        }
}
```

insertion의 위치

← null tree 인 경우

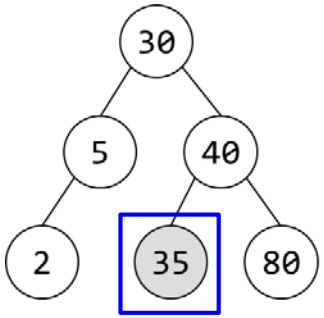
*node의 값이 바뀜 → 호출하는 쪽의 값을 변경
treePointer node 가 아니라 treePointer *node인 이유



http://btechsmartclass.com/data_structures/binary-search-tree.html

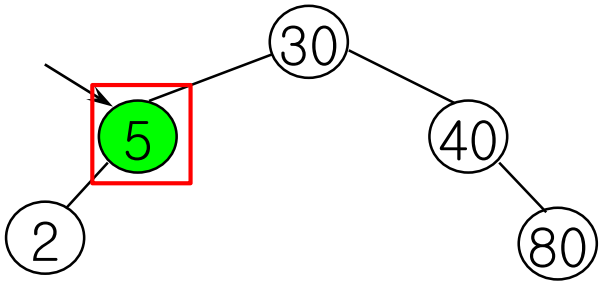
이원 탐색 트리에서의 삭제
(Deletion from a Binary Search Tree)

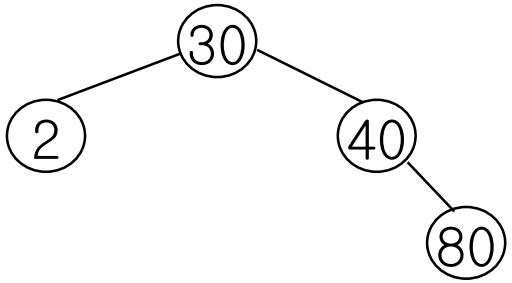
- **리프 원소의 삭제**
  - 부모의 자식 필드에 0을 삽입, 삭제된 노드 반환



leaf 삭제,
40의 leftchild = null

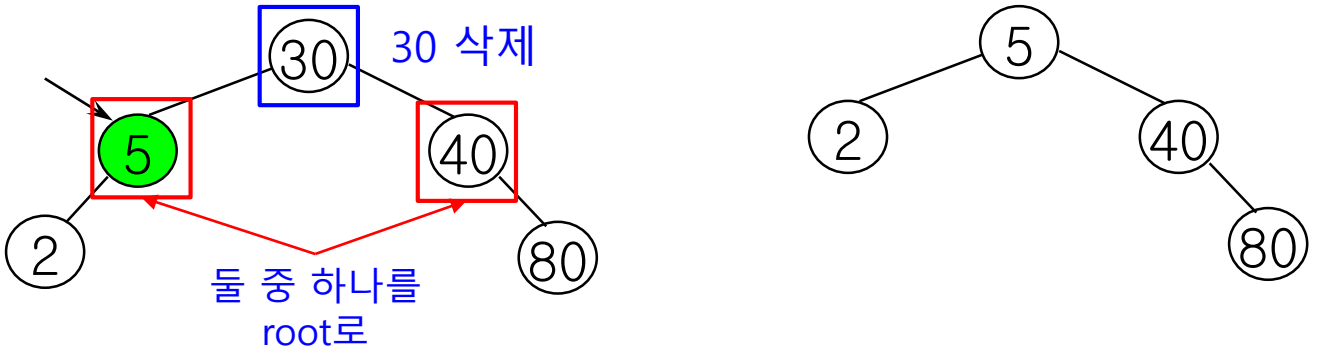
- **하나의 자식을 가진 비리프 노드의 삭제**
  - 삭제된 노드의 자식을 삭제된 노드의 자리에 위치



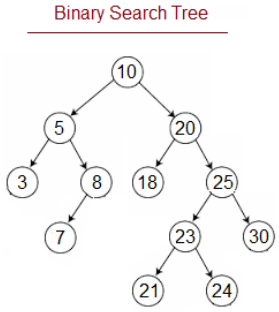


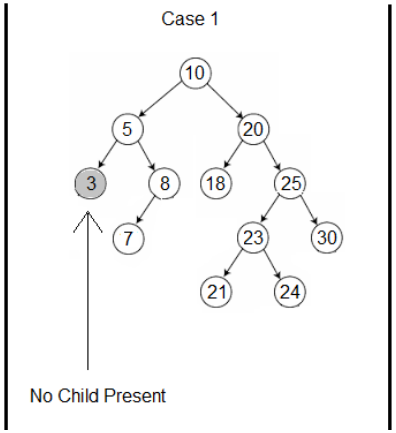
이원 탐색 트리에서의 삭제
(Deletion from a Binary Search Tree)

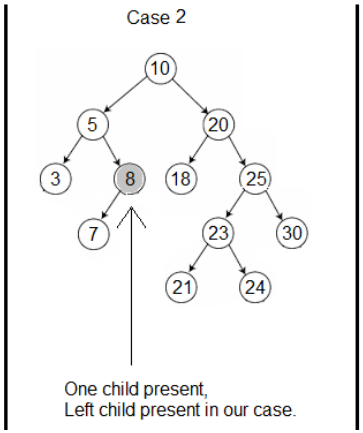
- **두 개의 자식을 가진 non-leaf 노드의 삭제**
  - 왼쪽 서브트리에서 가장 큰 원소나 오른쪽 서브트리에서 가장 작은 원소로 대체
  - 대체된 서브트리에서 대체한 원소의 삭제 과정 진행

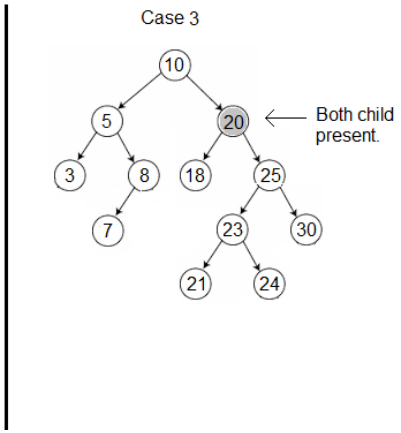


Time Complexity = O(log n), ~~Time Complexity = O(h)~~

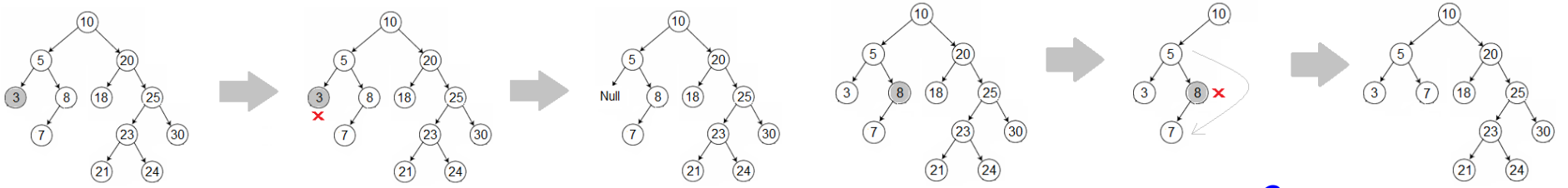




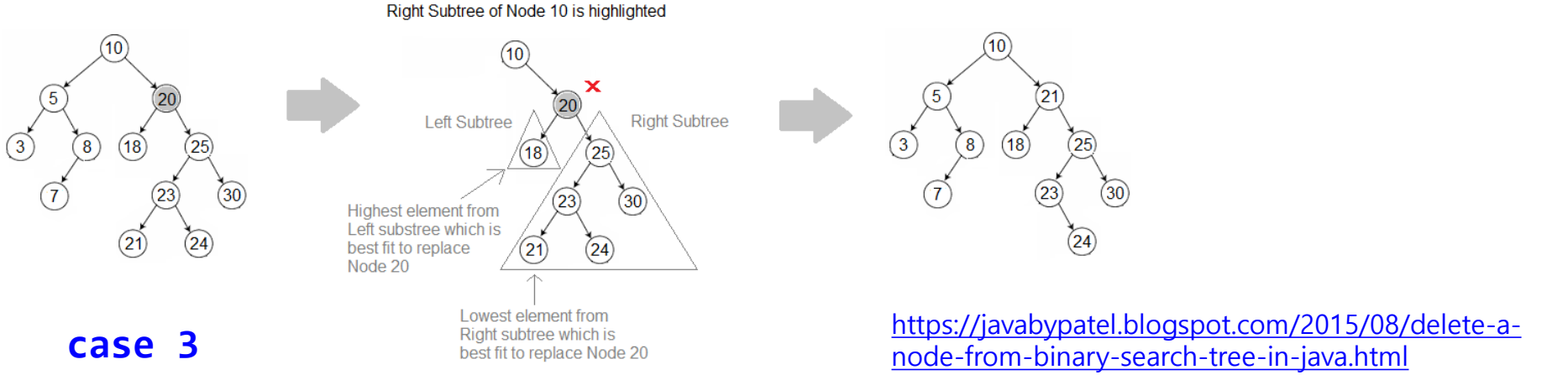




case 1



case 2

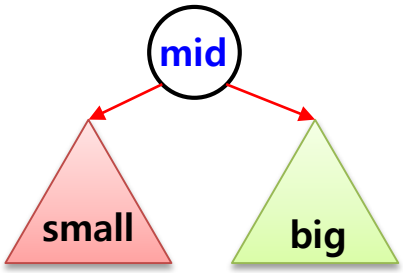


case 3

https://javabypatel.blogspot.com/2015/08/delete-a-node-from-binary-search-tree-in-java.html

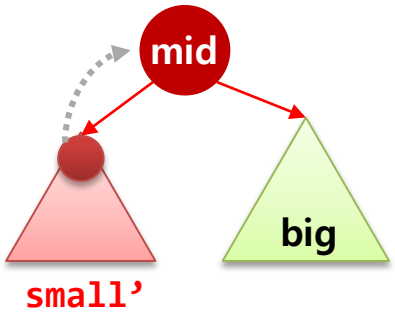
이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)

- **threeWayJoin(small, mid, big)**
  - 이원 탐색 트리 small과 big에 있는 쌍들과 쌍 mid로 구성되는 이원 탐색 트리를 생성
  - 새로운 노드를 하나 얻어서
    - 데이타 필드=mid
    - 왼쪽 자식 포인터=small.root
    - 오른쪽 자식 포인터는 big.root
  - O(1)



이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)

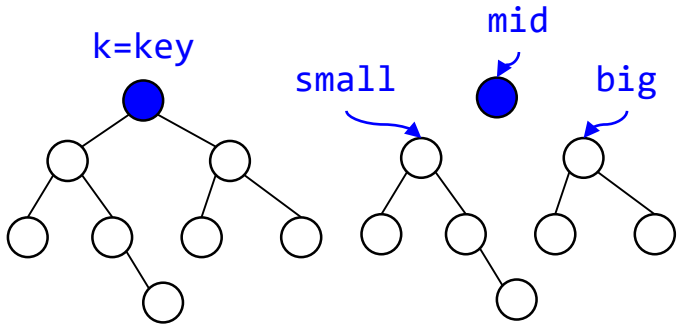
- **twoWayJoin(small, big)**
  - 두 이원 탐색트리 small과 big에 있는 모든 쌍들을 포함하는 하나의 이원 탐색 트리 생성
  - small 또는 big이 공백일 경우
    - 공백이 아닌 것이 이원 탐색 트리
  - 둘 다 공백이 아닌 경우
    - small에서 가장 큰 키 값을 가진 mid 쌍 삭제 : small'
    - ThreeWayJoin(small', mid, big) 수행
  - O(height(small)) ← small에서 노드 삭제가 발생
    ← O(log (# of nodes in small))



이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)

- **split(k, small, mid, big)**
  - 이원 탐색 트리 *theTree를 세 부분으로 분할
  - small은 *theTree의 원소 중 키 값이 k보다 작은 모든 쌍 포함
  - mid는 *theTree의 원소 중 키 값이 k와 같은 쌍 포함
  - big은 *theTree의 원소 중 키값이 k보다 큰 모든 쌍 포함

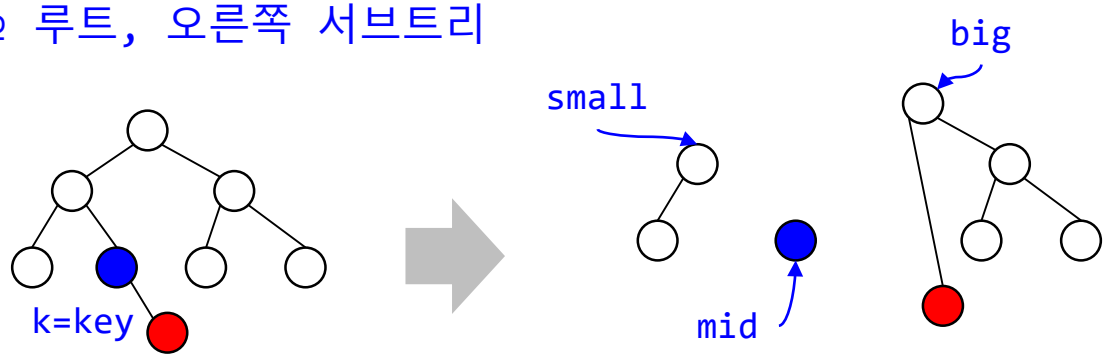
  - k = root->data.key인 경우 (=key)
    - small = *theTree의 왼쪽 서브트리
    - mid = 루트에 있는 쌍
    - big = *theTree의 오른쪽 서브 트리



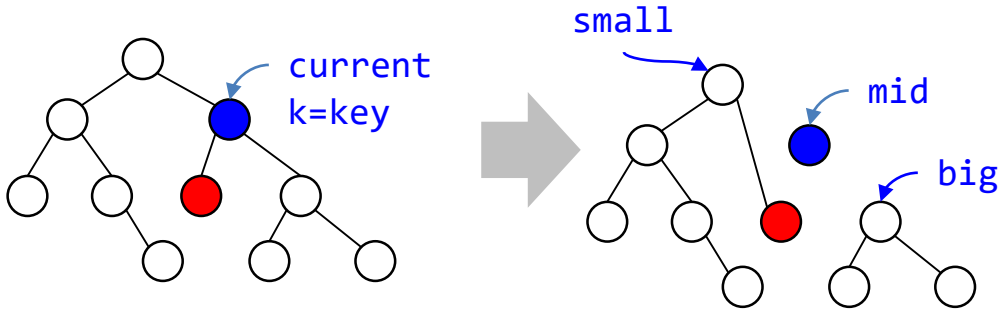
이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)

updated

- k < root->data.key인 경우
  - big ⊇ 루트, 오른쪽 서브트리

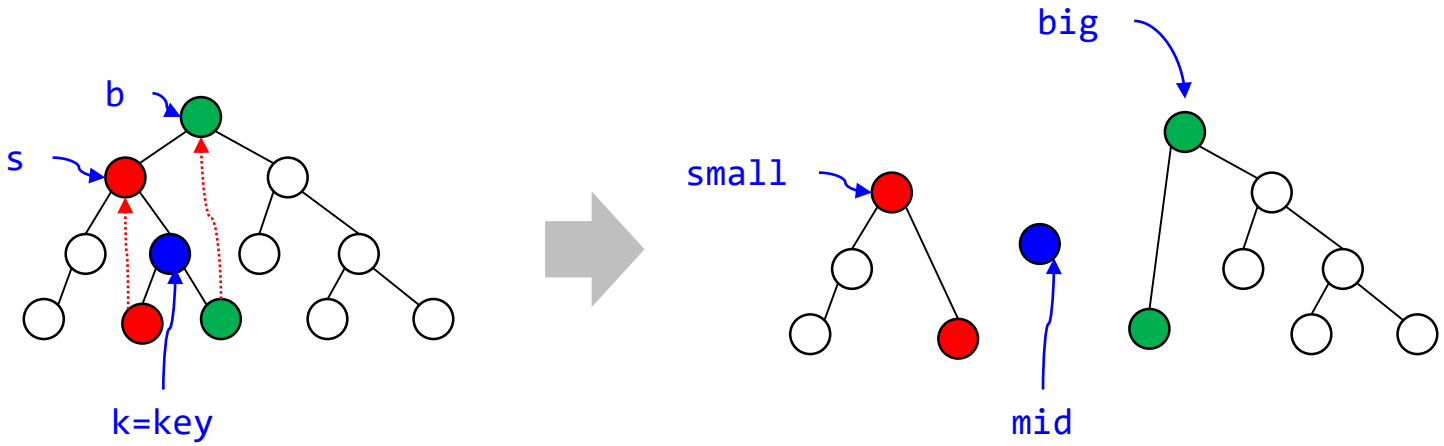


- k > root->data.key
  - small ⊇ 루트, 왼쪽 서브트리



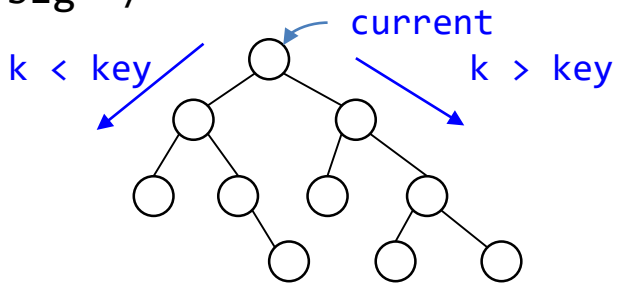
이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)

new

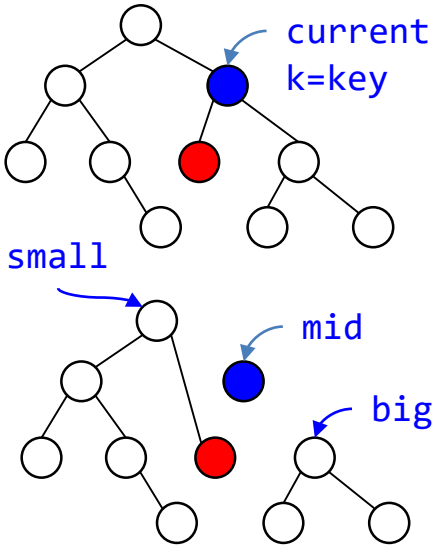


이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)

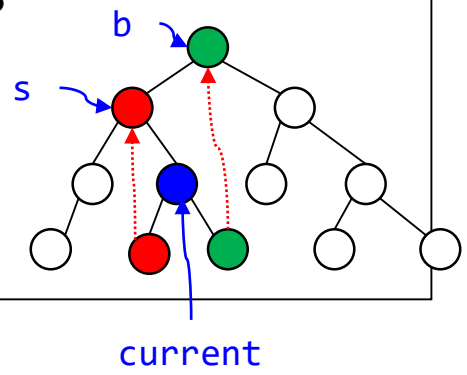
```
void split (nodePointer *theTree, int k, nodePinter *small,
        element *mid, nodePointer *big)
{/* split the binary search tree with respect to key k */
    if (!theTree) {*small = *big = 0;
         (*mid).key = -1; return ;} /* empty tree */
    nodePointer sHead, bHead, s, b, currentNode;
    /* create header nodes for small and big */
    MALLOC(sHead, sizeof(*sHead));
    MALLOC(BHead, sizeof(*bHead));
    s = sHead; b = bHead;
    /*do the split */
    currentNode = *theTree;

    while (currentNode)
        if(k < currentNode->data.key){/*add to big */
            b->leftChild = currentNode;
            b = currentNode; currentNode = currentNode->leftChild;
        }
        else if (k > currentNode->data.key){ /* add to small */
            s->rightChild = currentNode;
            s = currentNode; currentNode = currentNode->rightChild;
        }
```



이원 탐색 트리의 조인과 분할
(Joining and Splitting Binary Trees)




```
        else {/*split at currentNode */
                s->rightChild = currentNode->leftChild;
                b->leftChild = currentNode->rightChild;
                *small = sHead->rightChild; free(sHead);
                *big = bHead->leftChild; free(bHead);
                (*mid).item = currentNode->data.item;
                (*mid).key = currentNode->data.key;
                free(currentNode);
                return;
        }
    /* no pair with key k */
    s->rightChild = b->leftChild = 0;
    *small = sHead->rightChild; free(sHead);
    *big = bHead->leftChild; free(bHead);
    (*mid).key = -1;
    return;
}
```

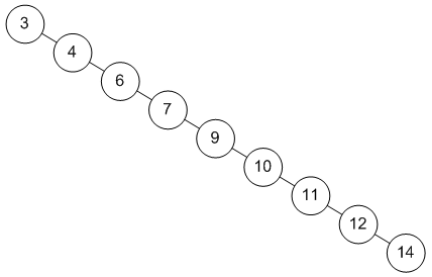


이원 탐색 트리(n)의 높이
(Height of a Binary Search Tree)

- 이원 탐색 트리의 원소 수 : n

- **최악의 경우**
  - 이원 탐색 트리의 높이 = n
  - 키 [1,2,3,...,n]을 순서대로 삽입

skewed binary search tree



- **평균**
  - 이원 탐색 트리의 높이 = O(log n)

- **균형 탐색 트리(balanced search tree)**
  - 최악의 경우에도 height = O(log n)이 되는 트리
  - 탐색, 삽입, 삭제의 시간 복잡도 : O(h) = O(log n)
  - AVL, 2-3, 2-3-4, 레드-블랙(red-black), B 트리, **B+ 트리** 등

B Tree == Balanced Tree
~~**binary tree X**~~

database system의 index

선택 트리
(Selection Trees)

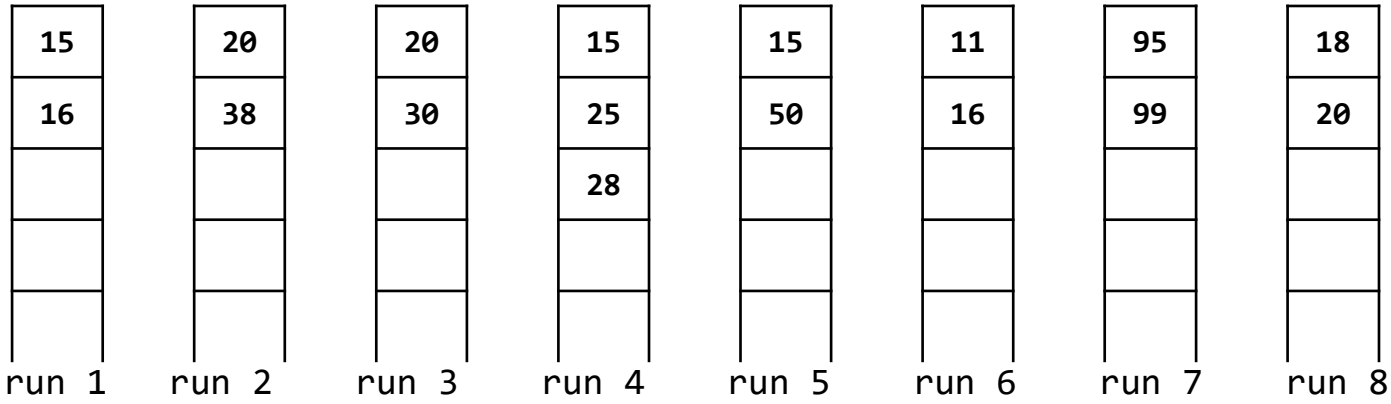
- Complete Binary Tree
- Each node represents "match"
- 토너먼트 트리, 승자 트리, 패자 트리

단말 노드 (leaf node, terminal node) 부터 시작해서, 큰 값 또는 작은 값을 찾아 부모로 올려서 root에 제일 큰 값 또는 제일 작은 값 올리는 방식

개요(Introduction)

- **런(run)**
  - 합병(merge)될 k개의 순서 순차(single ordered sequence)
  - 각 런은 key 필드에 따라 비감소(nondecreasing) 순서로 정렬

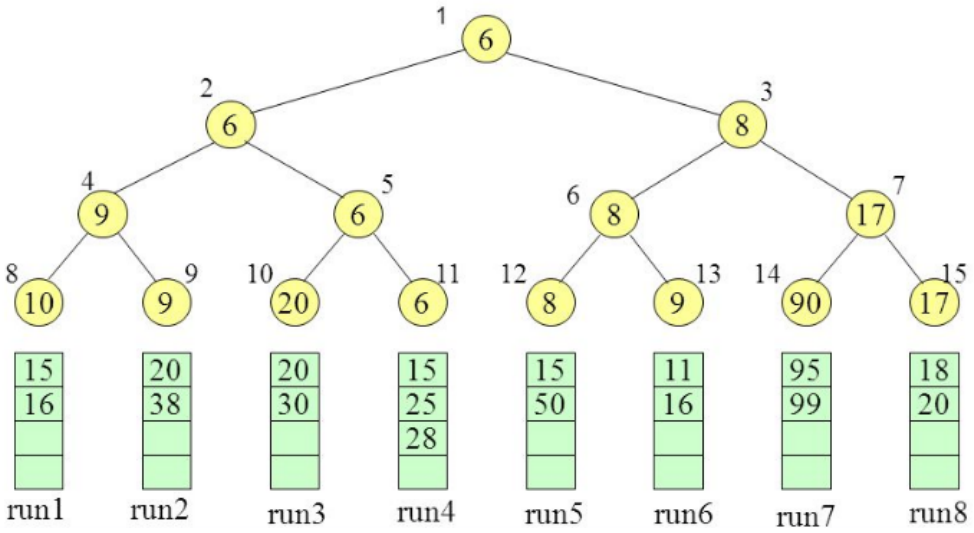
오름차순



- **합병(merge)**
  - 정렬된 데이터 리스트 k 개를 하나의 리스트로 만드는 과정
  - 일반적으로 데이터 리스트가 k 개인 경우 k-1 번 비교함
  - 선택 트리(selection tree)를 이용하여 비교 횟수를 줄일 수 있음
    - 다음 번 가장 작은(또는 큰) 수를 찾는 비교횟수를 줄임

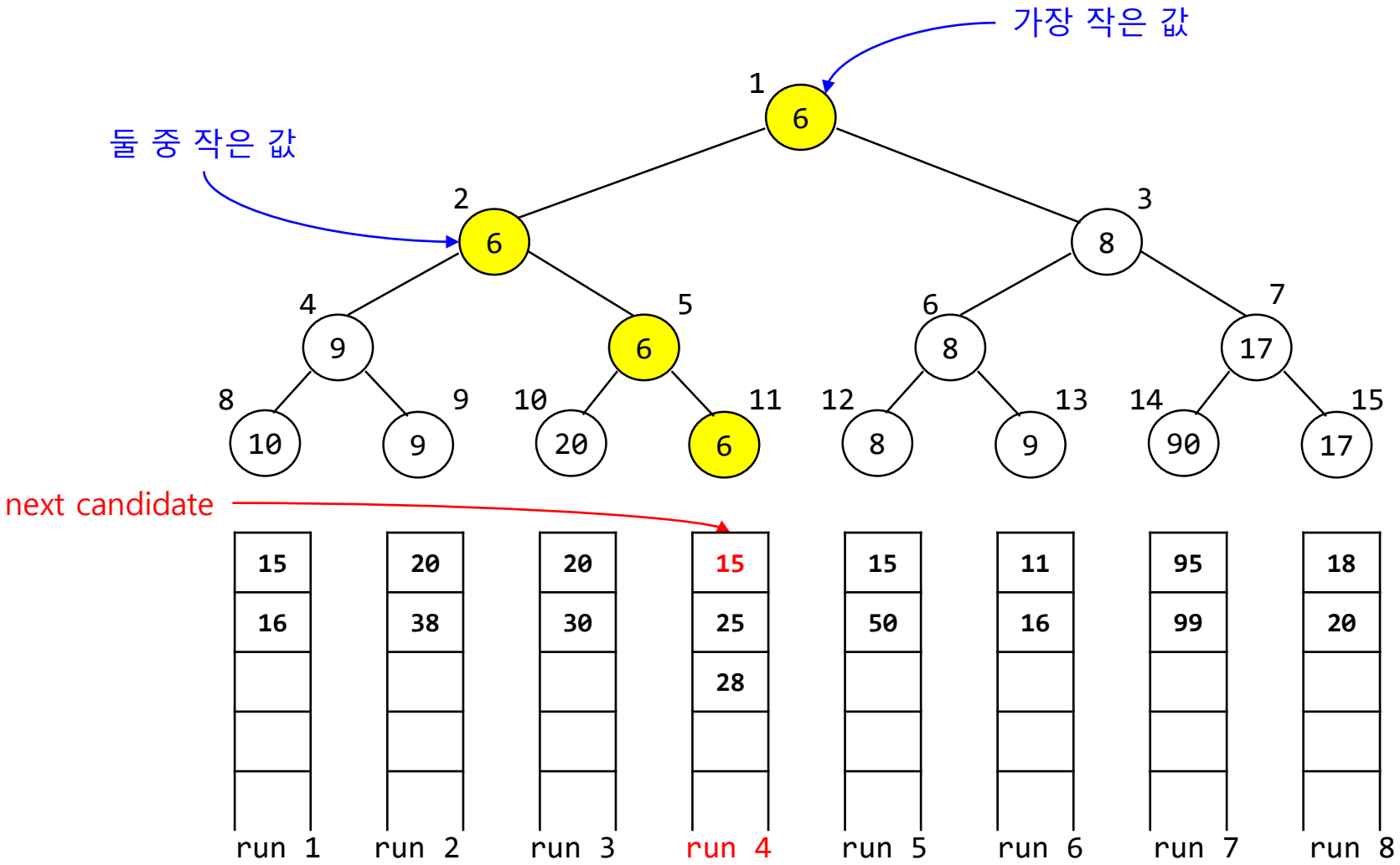
승자 트리(Winner Trees)

- **승자 트리(winner tree)**
  - 각 노드가 자식 노드 중 더 작은 노드를 나타내는 완전 이진 트리
  - 루트 노드 : 트리에서 가장 작은 노드
  - 리프 노드 : 각 런의 첫 번째 레코드
  - 비리프 노드 : 토너먼트의 승자
  - 이진 트리에 대한 순차 할당 기법으로 표현
    - 최종 승자가 된 run에서 다음 데이터를 읽어들임



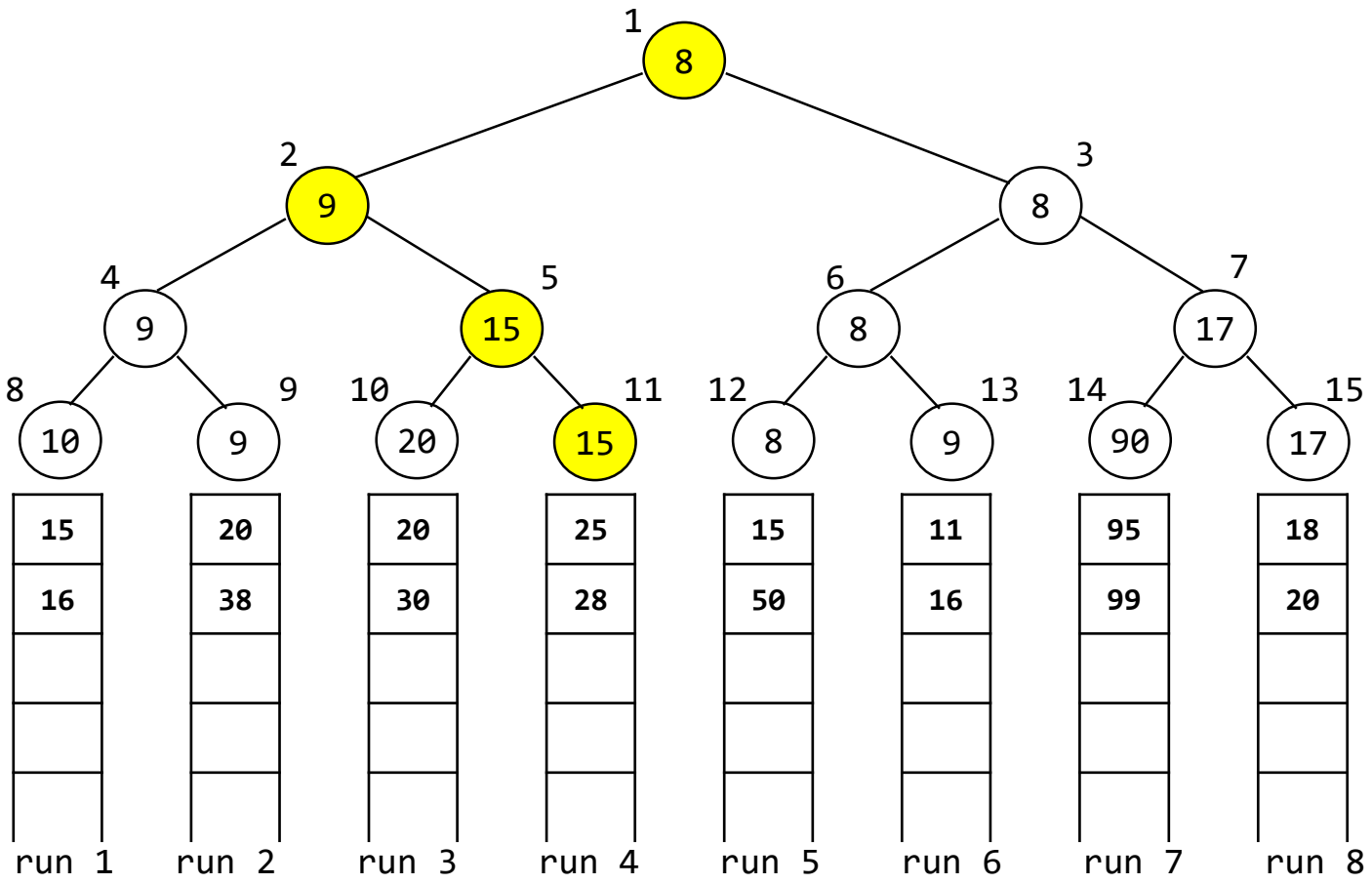
승자 트리(Winner Trees)

- k=8인 경우의 승자 트리



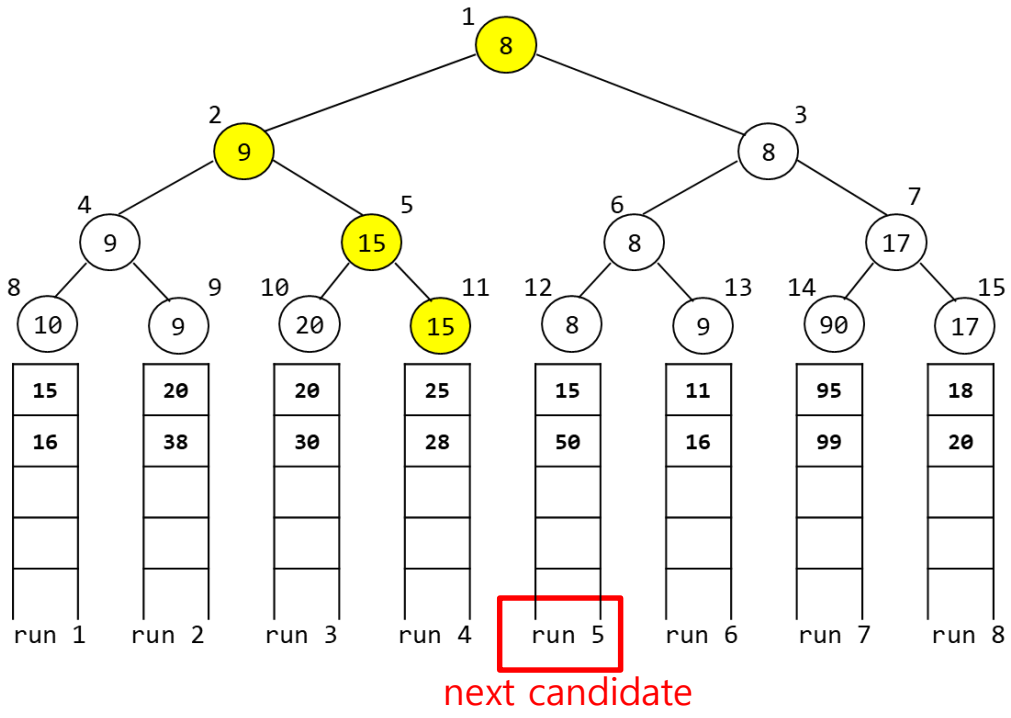
승자 트리(Winner Trees)

- **승자 트리에서 한 레코드가 출력되고 나서 재구성**
  - 새로 삽입된 노드에서 루트까지의 경로를 따라 토너먼트 재수행



승자 트리(Winner Trees)

- 런 합병 분석



tree의 레벨의 수
= $\lceil \log_2 k + 1 \rceil$

k = # of runs
= # of leaf nodes

# of nodes at level i
= 2^{i-1} = k (run)

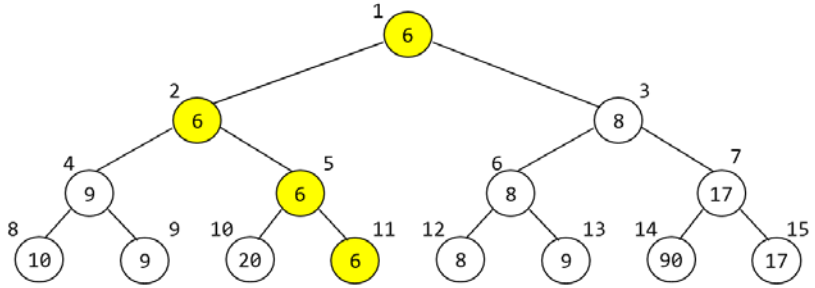
첫 승자 트리 구성 시간
= O(k)

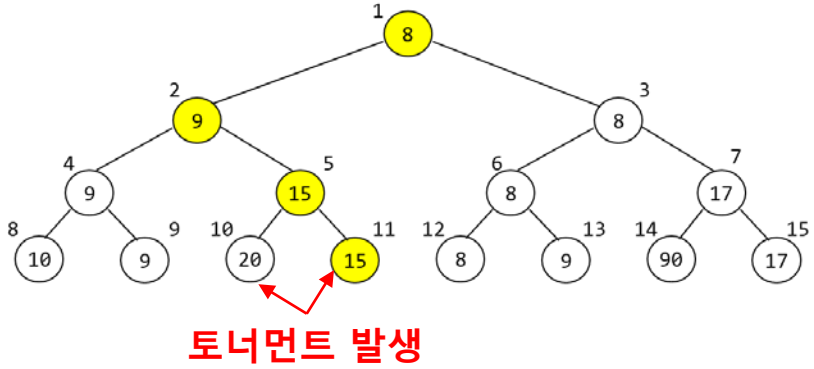
tree 재구성
= $O(\log_2 k)$

n개의 record 합병
= $O(n \log_2 k)$

패자 트리(Loser Trees)

- **승자 트리의 문제점**
  - 재구성 할 때 루트까지의 경로를 따라 형제 노드들 사이에 토너먼트 발생
  - 형제 노드는 전에 시행되었던 토너먼트의 패자들

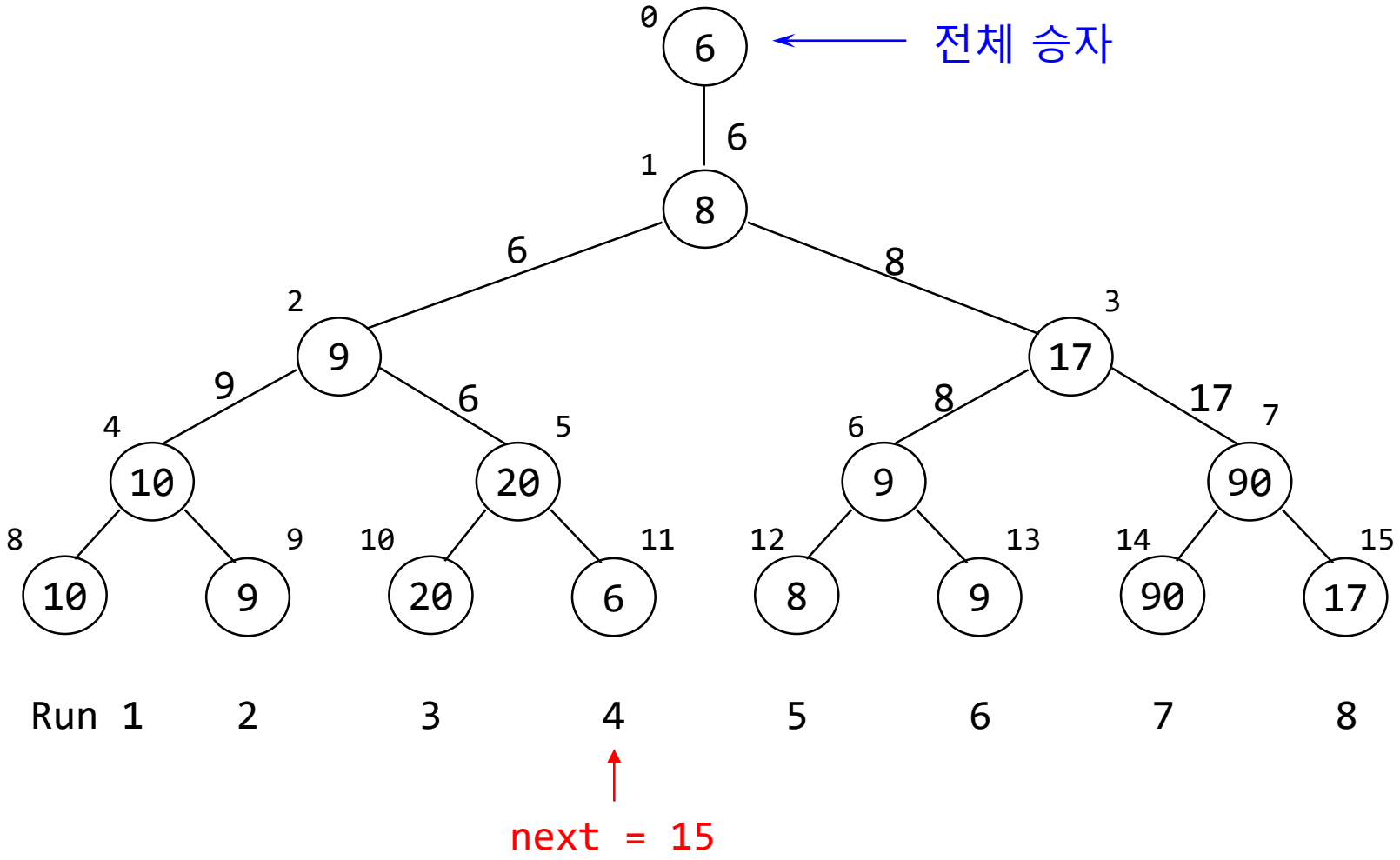




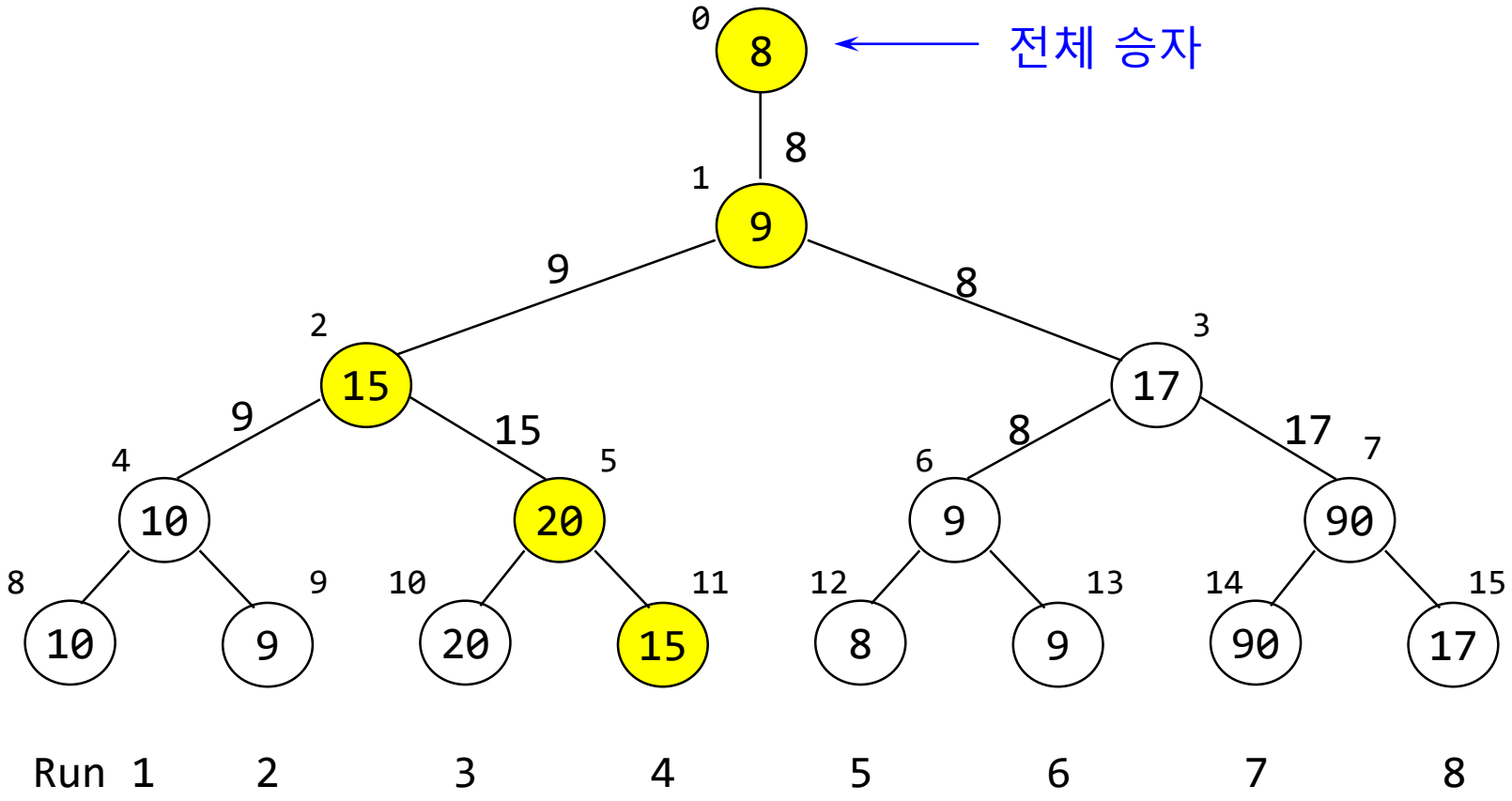
- **패자 트리(loser tree)**
  - 비리프 노드가 패자 레코드에 대한 포인터 유지
  - 전체 토너먼트의 승자를 표현하기 위한 노드 0 첨가
  - 토너먼트는 부모와 비교
  - 형제 노드에는 접근 하지 않음

최종 승자는 0번 노드에 저장

패자 트리(Loser Trees)

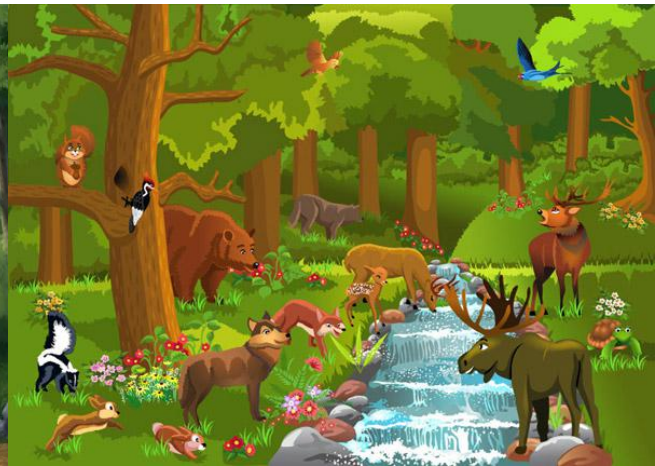


패자 트리(Loser Trees)



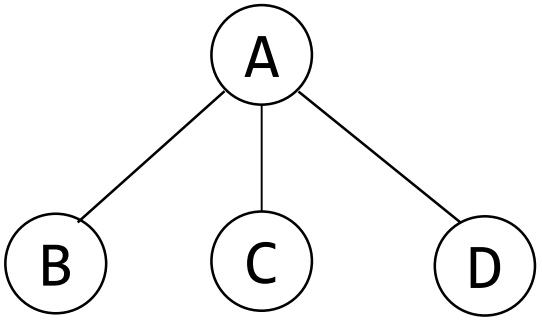
포리스트
(Forests)

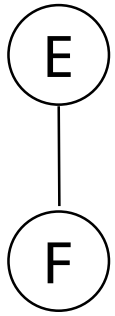


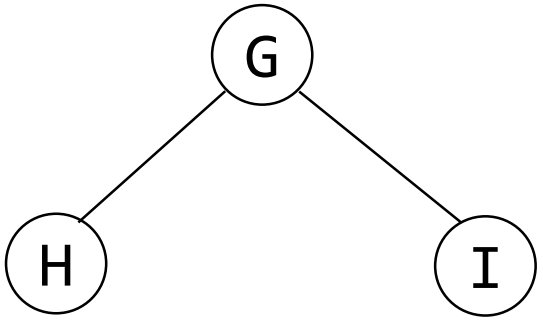


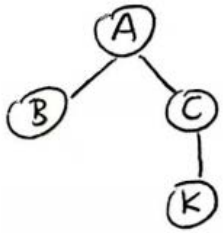
정의(Definition)

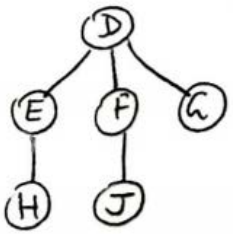
- **포리스트(forest)**
  - n(≥ 0)개의 분리(disjoint) 트리(tree)들의 집합

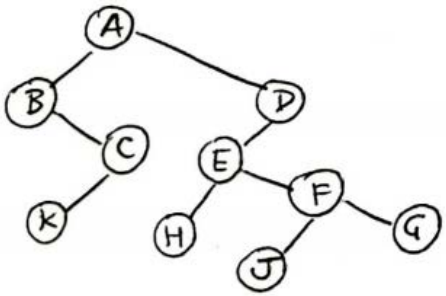






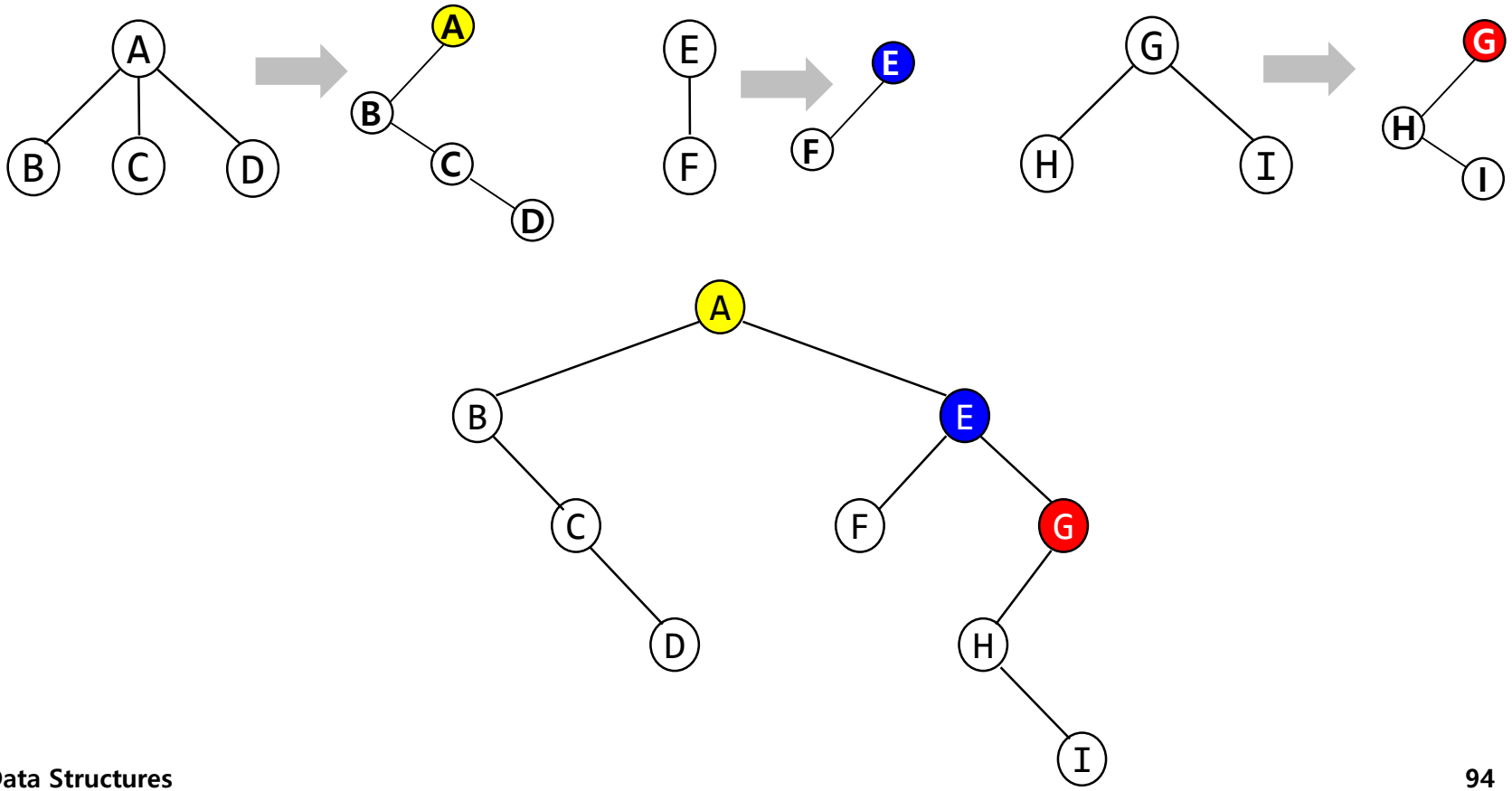






포리스트를 이진 트리로 변환
(Transforming a Forest into a Binary Tree)

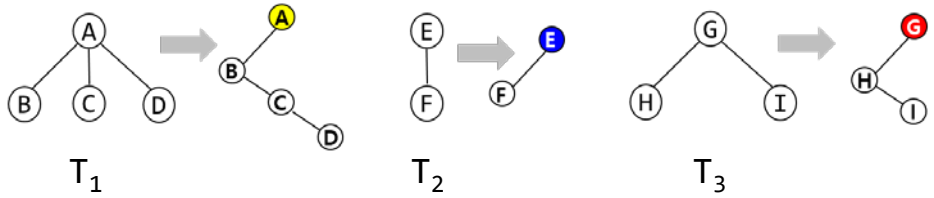
- 각 트리를 이진 트리로 변환
- 변환된 모든 이진 트리들을 루트의 rightChild로 연결



포리스트를 이진 트리로 변환
(Transforming a Forest into a Binary Tree)

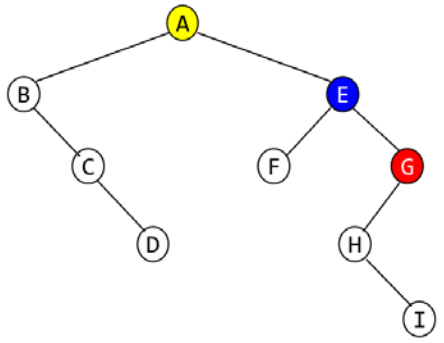
- **정의**
  - $T_1, T_2, \ldots, T_n$ 트리로 구성된 포리스트에 대응하는 이진 트리, $B(T_1, T_2, \ldots, T_n)$은

1) n=0이면 공백

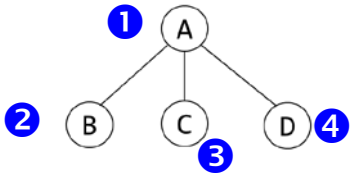


T_1 T_2 T_3

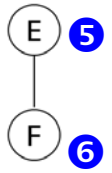
2) B의 루트 = T_1의 루트
   - 왼쪽 서브 트리 = $B(T_{11}, T_{12}, \ldots, T_{1m})$, $T_{11}, T_{12}, \ldots, T_{1m}$는 T_1의 루트의 서브 트리
   - 오른쪽 서브트리 = $B(T_2, \ldots, T_n)$

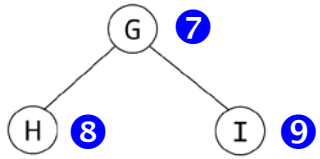


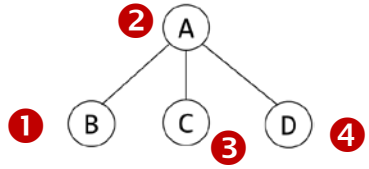
포리스트 순회(Forest Traversals)



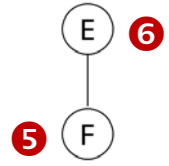
- **포리스트 전위(forest preorder)**
  - F가 공백이면 복귀
  - F의 첫 번째 트리의 루트 방문
  - 첫 번째 트리의 서브트리들을 포리스트 전위 순회
  - F의 나머지 트리들을 포리스트 전위로 순회

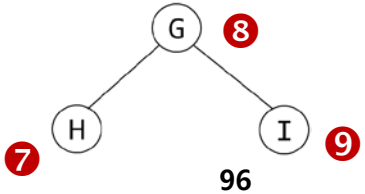




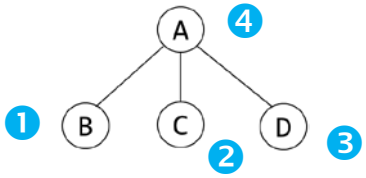


- **포리스트 중위(forest inorder)**
  - F가 공백이면 복귀
  - F의 첫 번째 트리의 서브트리를 포리스트 중위로 순회
  - 첫 번째 트리의 루트 방문
  - 나머지 트리들을 포리스트 중위로 순회

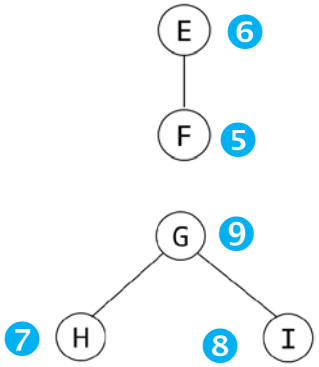


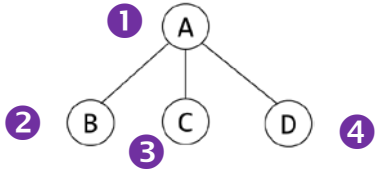


포리스트 순회(Forest Traversals)

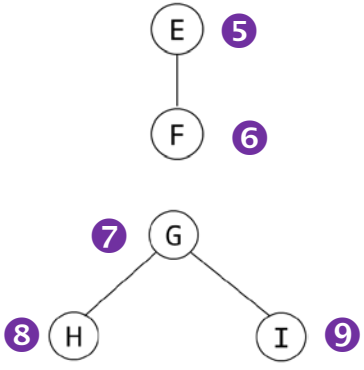


- **포리스트 후위(forest postorder)**
  - F가 공백이면 귀환
  - F의 첫 트리의 서브트리를 포리스트 후위로 순회
  - 나머지 트리들을 포리스트 후위로 순회
  - F의 첫 트리의 루트를 방문





- **포리스트 레벨 순서 순회(level order traversal)**
  - 포리스트의 각 루트부터 시작,레벨 순으로 방문
  - 레벨 내에서는 왼쪽에서 오른쪽으로 차례로 방문



Additional Slides

이진 트리의 개수 계산
(Counting Binary Trees)

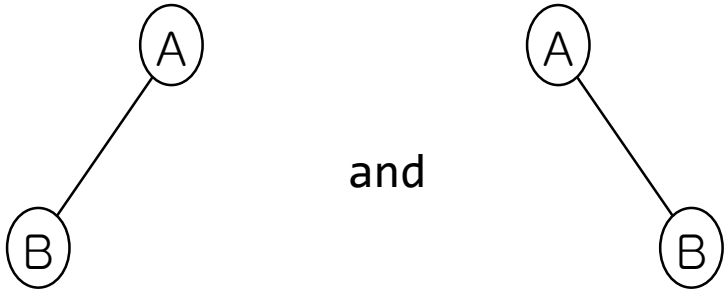
조합론에서, 카탈란 수(Catalan數, 영어: Catalan number)는
이진 트리의 수 따위를 셀 때 등장하는 수열이다.

---wikipedia

외젠 샤를 카탈랑(프랑스어: Eugène Charles Catalan, 1814년 5월 30일 – 1894년 2월 14일)은 벨기에의 수학자이다. 수론에 기여하였다.

상이한 이진 트리(Distinct Binary Tree)

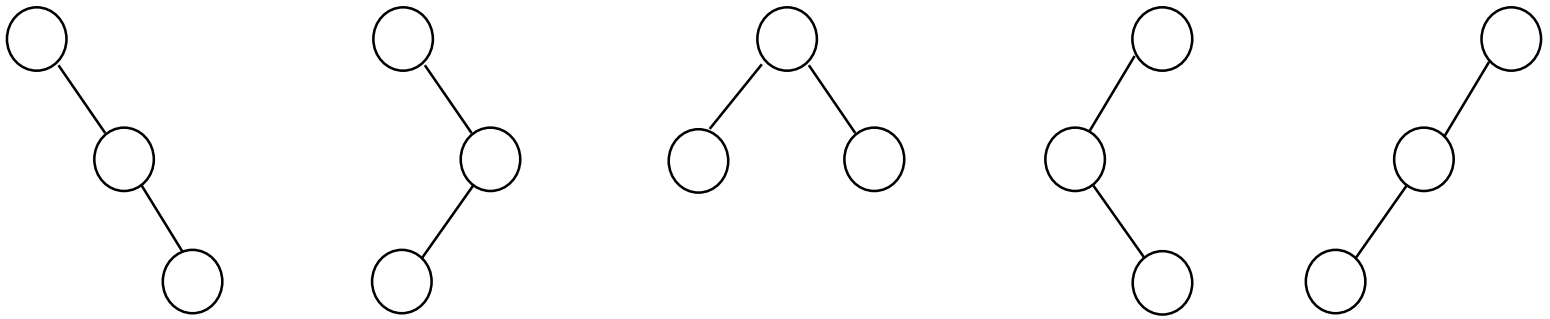
- n=2 일 때 서로 다른 이진 트리 : 2개



n개의 노드를 가진 서로 다른 이진 트리의 개수는?

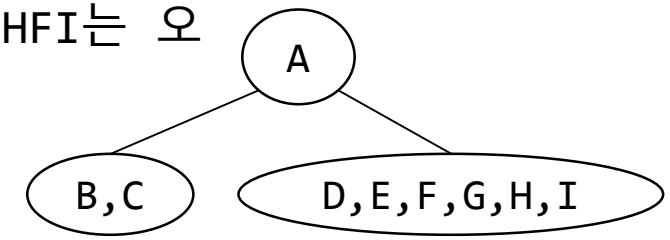
$$\mathbf{b_n} = \frac{(2n)!}{n!\,(n+1)!}$$

- n=3 일 때 서로 다른 이진 트리 : 5개

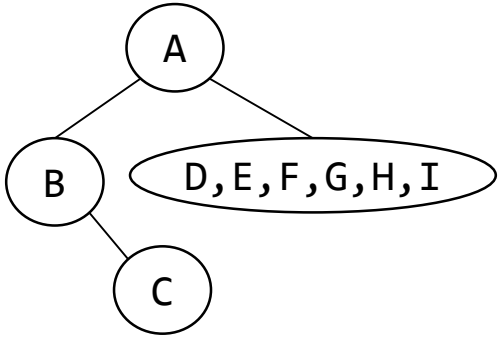


스택 순열(Stack Permutations)

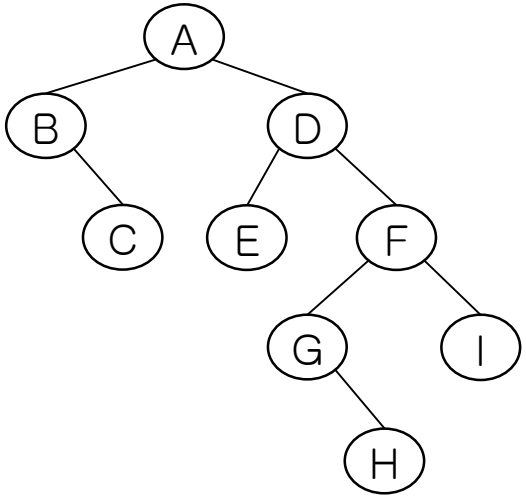
- **전위 순서 ABCDEFGHI, 중위 순서 BCAEDGHFI**
  - 전위 순서: A는 트리의 루트
  - 중위 순서: BC는 A의 왼쪽 서브트리, EDGHFI는 오른쪽 서브트리



  - 전위 순서: B가 다음번 루트
  - 중위 순서: B의 왼쪽 서브트리는 공백, 오른쪽은 C

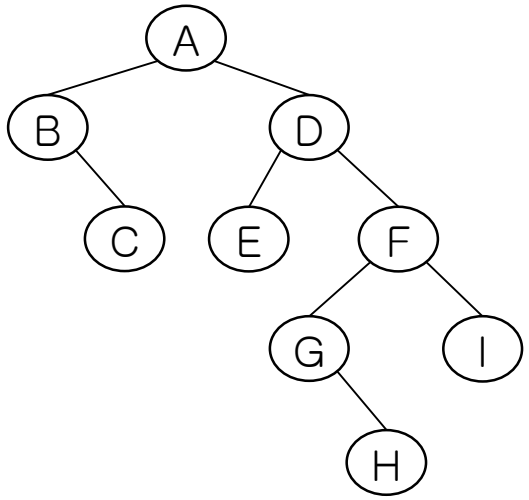


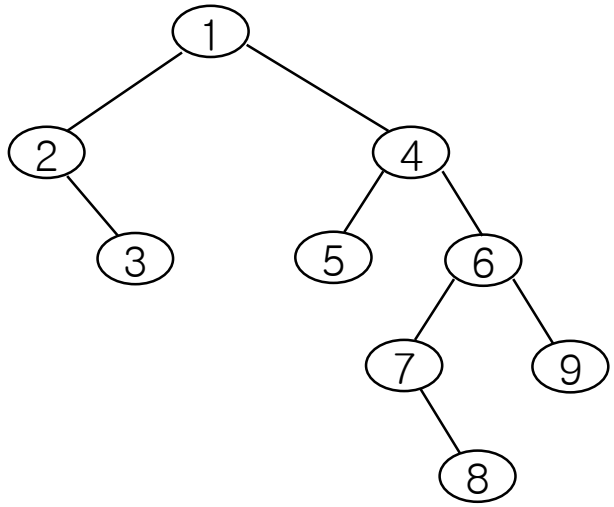
  - 반복



스택 순열(Stack Permutations)

- **전위 순열(preorder permutation)**
  - 이진 트리를 전위 순회에 따라 방문한 노드들의 순서

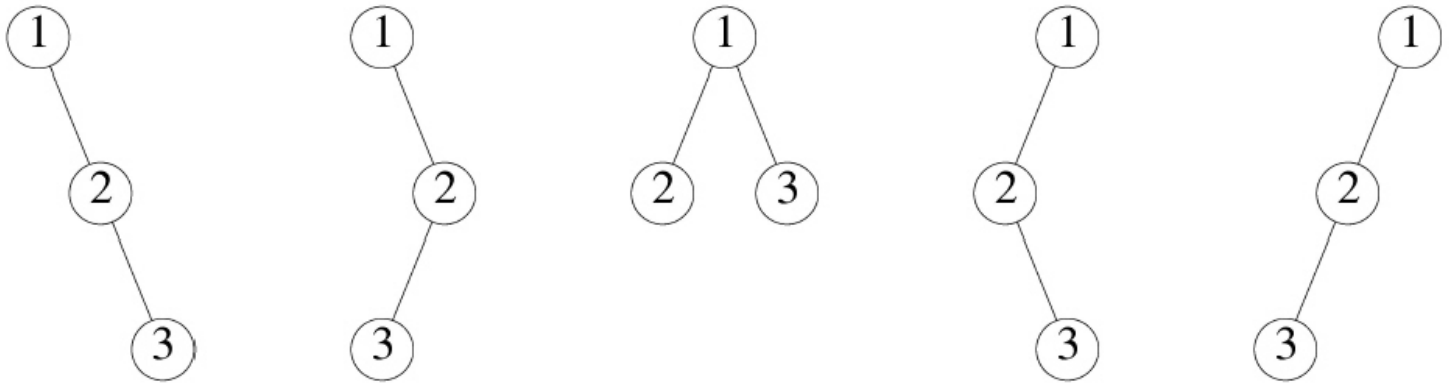




- **중위 순열(inorder permutation)**
  - 이진 트리를 중위 순회에 따라 방문한 노드들의 순서

스택 순열(Stack Permutations)

- **모든 이진 트리는 유일한 전위-중위 순서 쌍을 가짐**
- **n개의 노드를 가진 서로 다른 이진트리의 수**
  - 1,2,...,n의 전위 순열을 가지는 이진 트리로 부터 얻을 수 있는 중위 순열의 수
  - 1부터 n까지의 수를 스택에 넣었다가 가능한 모든 방법으로 삭제하여 만들 수 있는 상이한 순열의 수

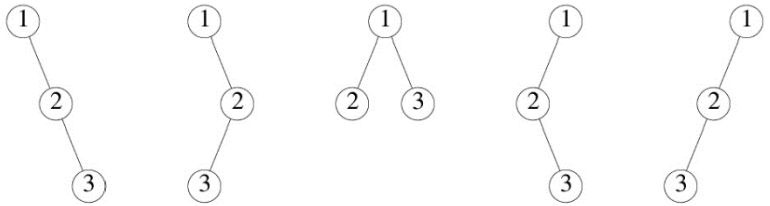


preorder = 1, 2, 3 & inorder = 2, 3, 1 이라면,
유일한 이진트리를 선택할 수 있는가?

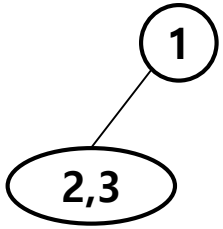
스택 순열(Stack Permutations)

preorder = 1, 2, 3 & inorder = 2, 3, 1

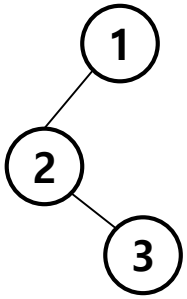
유일한 이진트리를 선택할 수 있는가?



- preorder로 첫 번째 노드 = 1 → root = 1
- inorder로 첫 번째 노드 = 2 → node 1의 왼쪽 node = 2



- inorder로 두 번 째 노드 = 3 → node 2의 오른쪽 node = 3



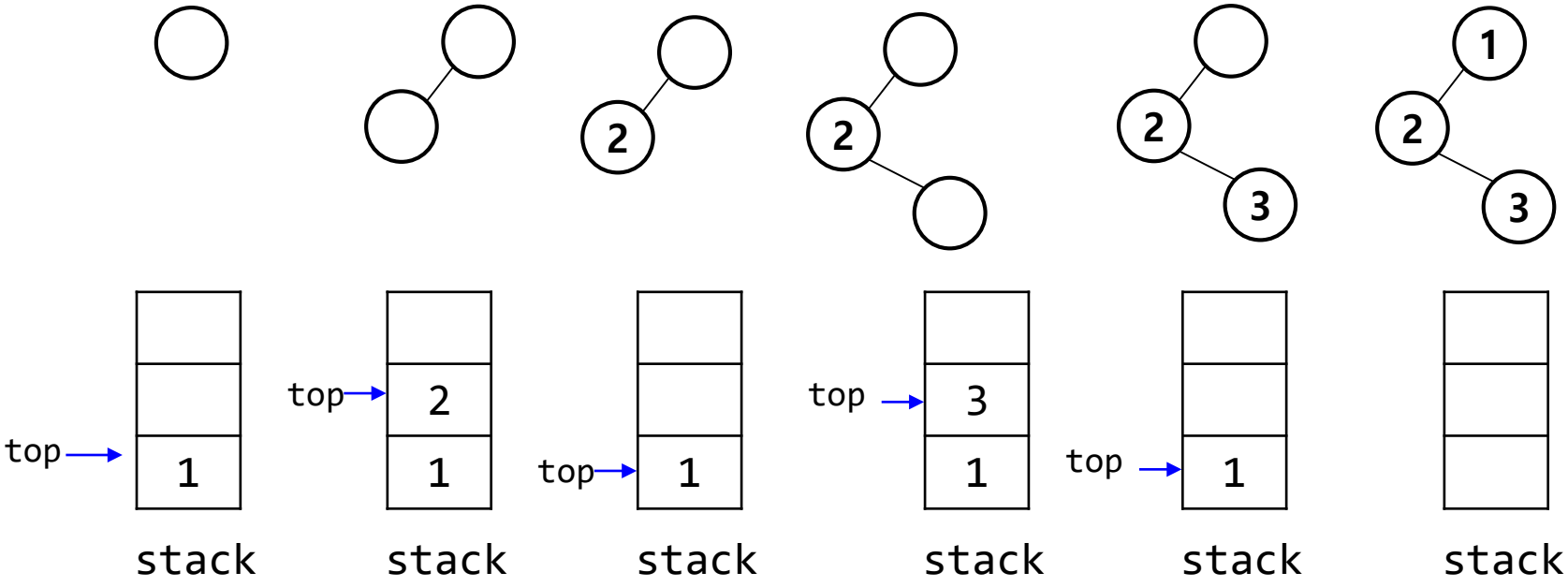
스택 순열(Stack Permutations)

- **stack operations**
  - push() : 빈 노드를 만들고 왼쪽으로
  - pop() : 빈 노드에 값을 채우고 오른쪽으로

push() pop()

preorder = 1, 2, 3 & inorder = 2, 3, 1

push(1), push(2), pop(2), push(3), pop(3), pop(1)



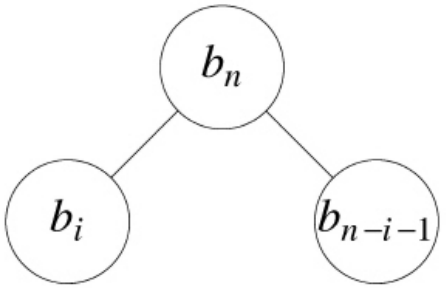
행렬곱셈(Matrix Multiplication)

- **n개 행렬의 곱셈**
  $M_1*M_2*M_3*M_4* \ ... \ *M_n$
- **n개의 행렬을 곱하는 방법의 수 b_n**
  - $b_2=1$, $b_3=2$, $b_4=5$
- **$M_{ij} = M_i*M_{i+1}* \ ... \ *M_j$**
- **$M_{1n} = M_{1i}*M_{i+1,n}$**
  - M_{1i} 계산 방법 수 : b_i
  - $M_{i+1,n}$ 계산 방법 수 : b_{n-i}

$$b_n = \sum_{i=1}^{n-1} b_i b_{n-i}, n > 1$$

- **b_n = n개의 노드를 가진 서로 다른 이진트리의 수**
  - 루트,노드 수가 b_i, b_{n-i-1}인 서브트리로 된 이진트리들

$$b_n = \sum_{i=1}^{n-1} b_i b_{n-i-1}, n \geq 1, and\ b_0 = 1$$



상이한 이진 트리의 수

catalan number

(Number of Distinct Binary Trees)

- 이진 트리의 수를 구하는 생성 함수(generating function)

$$B(x) = \sum_{i \geq 0} b_i x^i$$

- 순환 관계에 의하여, $xB^2(x)=B(x)-1$
- $B(0)=b_0=1$을 이용하여 이차 방정식을 풀면

$$B(x) = \frac{1-\sqrt{1-4x}}{2x}$$

- 이항 정리를 이용하여 $(1-4x)^{1/2}$를 확장하면

$$B(x) = \frac{1}{2x}\left(1 - \sum_{n \geq 0} \binom{1/2}{n} (-4x)^n\right) = \sum_{m \geq 0} \binom{1/2}{m+1} (-1)^m 2^{2m+1} x^m$$

상이한 이진 트리의 수
(Number of Distinct Binary Trees)

- B(x)에서 x^n의 계수인 b_n은 아래와 같다.

$$\binom{1/2}{n+1}(-1)^n 2^{2n+1}$$

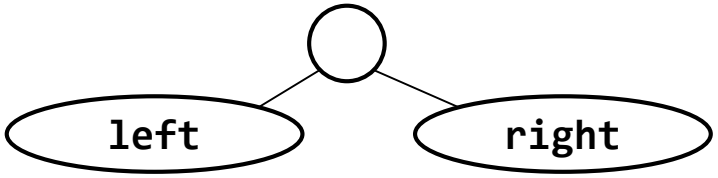
$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

$$\mathbf{b_n = \frac{(2n)!}{n!\,(n+1)!}}$$

$$b_n = O(4^n / n^{3/2})$$

상이한 이진 트리의 수
(Number of Distinct Binary Trees)

노드 n개에 대해 상이한 이진트리의 개수 = b_n



subtree의 node의 수 0 ~ n-1 n-1 ~ 0

b_i b_{n-1-i}

$$b_n = \sum_{i=0}^{n-1} b_i \, b_{n-1-i} \qquad b_0 = 1, n \geq 1$$

미지수의 계수가 수열의 각 항으로 되어 있는 멱급수 형태의 함수를 **생성함수** (generating function) B(x)를 정의

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \cdots = \sum_{k=0}^{\infty} a_k x^k$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \ldots$$

$$B(x) = \sum_{i \geq 0} b_i x^i$$

상이한 이진 트리의 수
(Number of Distinct Binary Trees)

$$B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 \ldots \qquad B(x) = \sum_{i \geq 0} b_i x^i$$

$$\begin{aligned}
(B(x))^2 &= (b_0 + b_1x + b_2x^2 + b_3x^3 \ldots)(b_0 + b_1x + b_2x^2 + b_3x^3 \ldots) \\
&= (b_0 + b_1x + b_2x^2 + b_3x^3 \ldots)(b_0 + b_1x + b_2x^2 + b_3x^3 \ldots) \\
&= b_0(b_0 + b_1x + b_2x^2 \ldots) + b_1x(b_0 + b_1x + \cdots) + b_2x^2(b_0 + b_1x + b_2x^2 \ldots) + \cdots \\
&= (b_0)^2 + b_0b_1x + b_0b_2x^2 + b_0b_3x^3 + \cdots \\
&\quad + b_1b_0x + b_1{}^2x^2 + b_1b_2x^3 + \cdots \\
&\quad + b_1b_2x^2 + b_2b_1x^3 + \cdots \\
&\quad + b_3b_0x^3 + \cdots \\
&= (b_0)^2 + (b_0b_1 + b_1b_0)x + (b_0b_2 + b_1{}^2 + b_1b_2)x^2 + (b_0b_3 + b_1b_2 + b_2b_1 + b_3b_0)x^3 + \cdots \\
&= (b_1) + (b_2)x + (b_3)x^2 + (b_4)x^3 + \cdots
\end{aligned}$$

$$(B(x))^2 = (b_1) + (b_2)x + (b_3)x^2 + (b_4)x^3 + \cdots$$

$$x(B(x))^2 = (b_1)x + (b_2)x^2 + (b_3)x^3 + (b_4)x^4 + \cdots = B(x) - b_0 = B(x) - 1$$

$$x(B(x))^2 = B(x) - 1$$

$$x(B(x))^2 - B(x) + 1 = 0$$

$$B(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

$B(0) = b_0$
+가 될 수 없다.

상이한 이진 트리의 수
(Number of Distinct Binary Trees)

- 이항 정리를 이용하여 $(1-4x)^{1/2}$를 확장하면

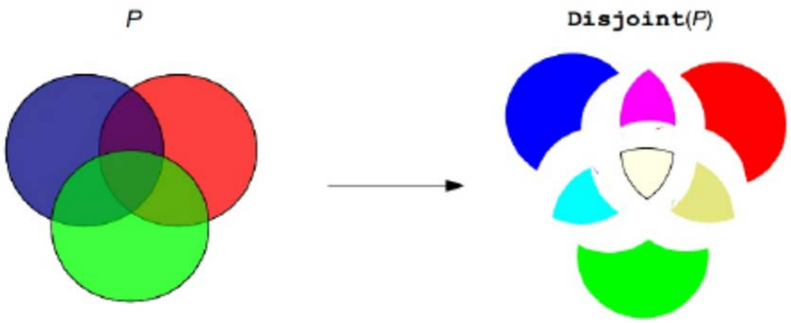
$$B(x) = \frac{1-\sqrt{1-4x}}{2x}$$

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k = x^n + nx^{n-1}y + \frac{n(n-1)}{2} x^{n-2}y^2 + \cdots + y^n$$

$$B(x) = \frac{1}{2x}\left(1 - \sum_{n\geq 0} \binom{1/2}{n} (-4x)^n\right) = \sum_{m\geq 0} \binom{1/2}{m+1} (-1)^m 2^{2m+1} x^m$$

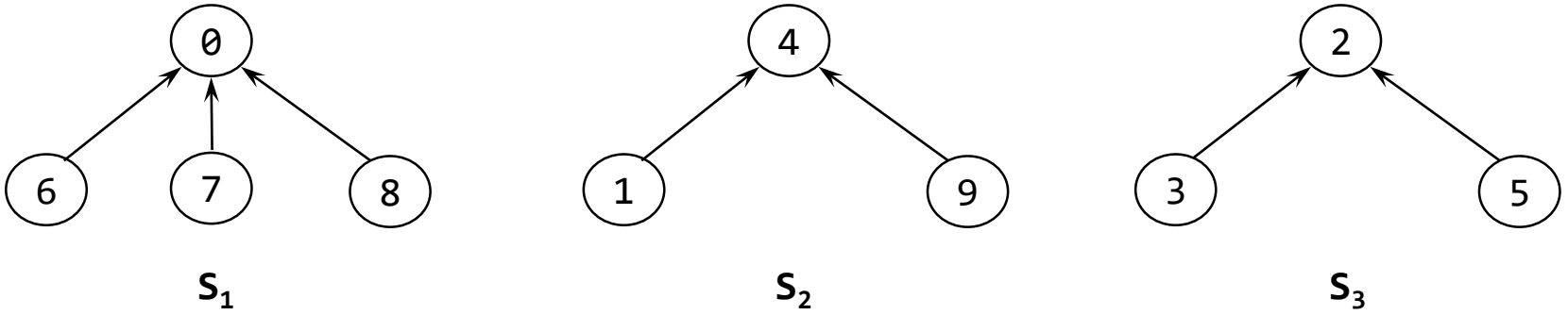
$$b_n = \frac{1}{n+1}\binom{2n}{n} \qquad \mathbf{b_n = \frac{(2n)!}{n!\,(n+1)!}}$$

분리 집합의 표현
(Representation of Disjoint Sets)



정의(Definition)

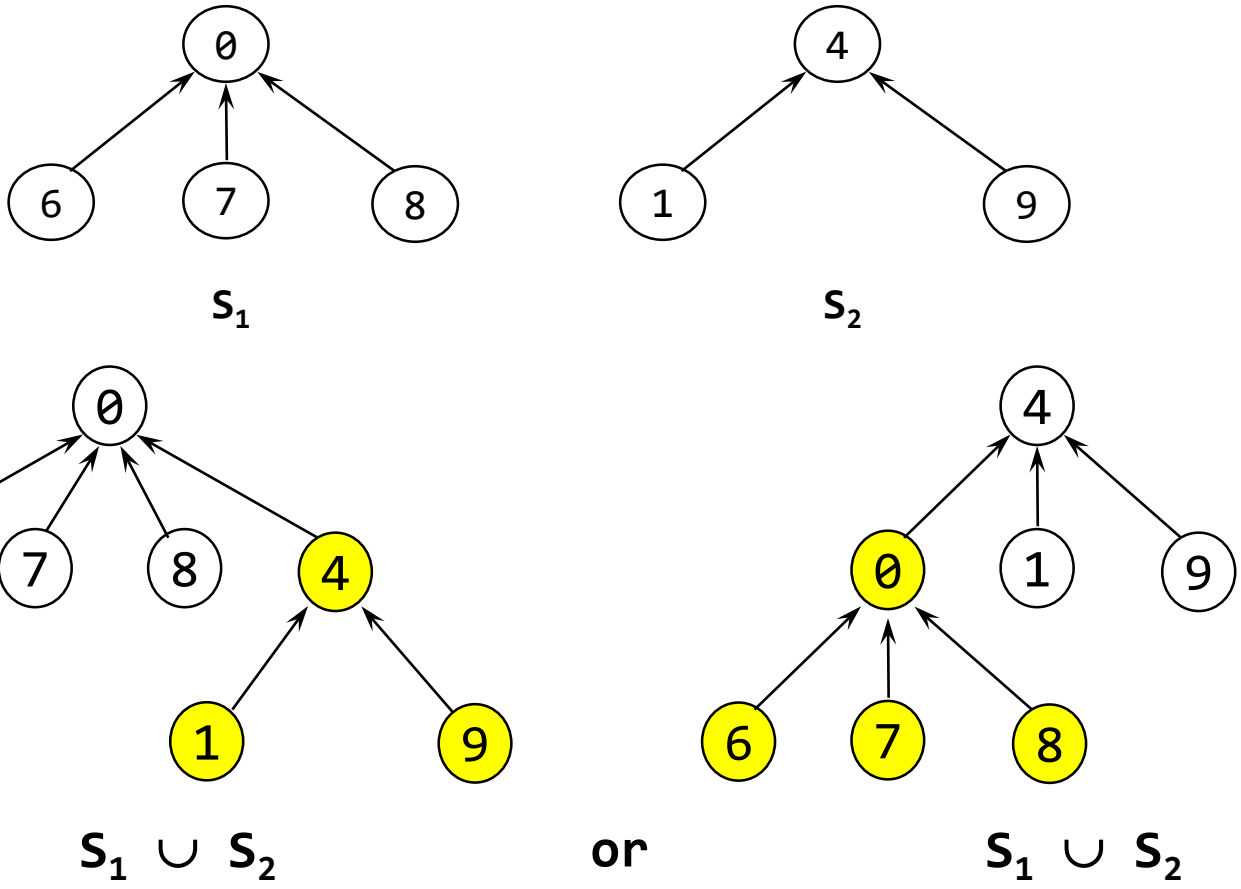
- **분리 집합(disjoint set)의 트리 표현**
  - 집합의 모든 원소는 수 0,1,2,...,n-1이라고 가정
  - 모든 집합들은 쌍별(pairwise)로 분리 된다고 가정
    - 임의의 두 집합은 어떤 원소도 공유하지 않음 ($i \neq j \rightarrow S_i \cap S_j = \varnothing$)
  - 자식에서부터 부모로 가는 링크로 연결



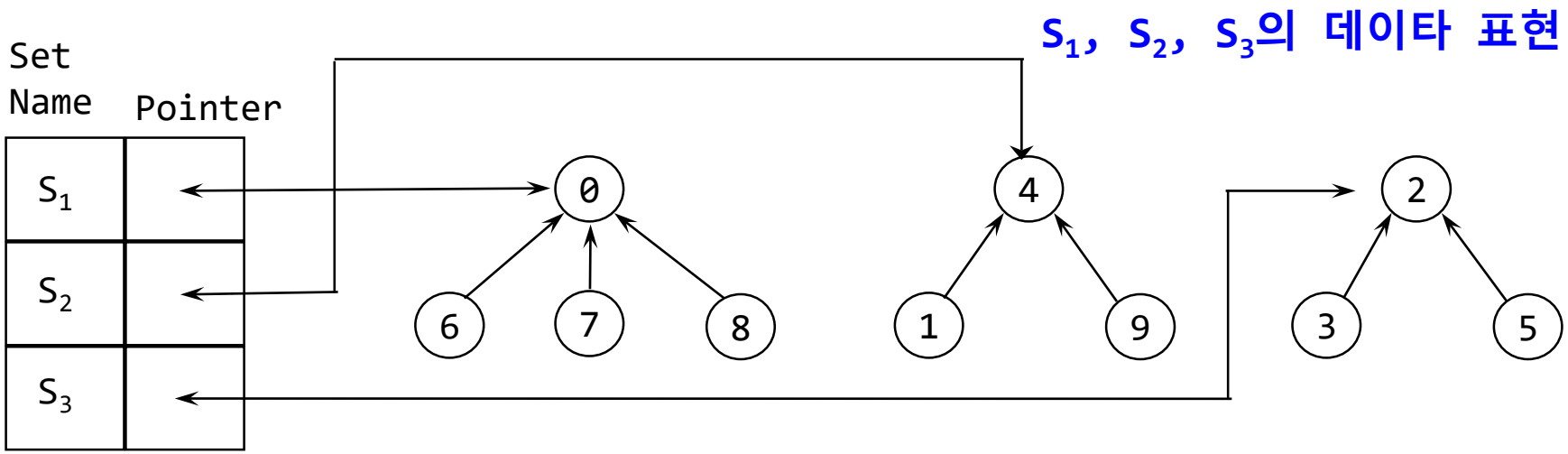
- **연산**
  - 분리 합집합: $S_i \cup S_j$ = {x|x는 S_i 또는 S_j 에 포함되는 모든 원소}
  - 탐색: 원소 i를 포함하는 집합 탐색

분리 합집합(Disjoint Set Union)

- $S_i \cup S_j = \{x | x$는 S_i 또는 S_j에 포함되는 모든 원소$\}$
- 두 트리 중의 하나를 다른 트리의 서브트리로 넣음



분리 합집합(Disjoint Set Union)



- **S_1, S_2, S_3의 배열 표현**
  - i번째 원소 : 원소 i를 포함하는 트리 노드
  - 원소 : 대응되는 트리 노드의 부모 포인터
  - 루트의 parent는 -1

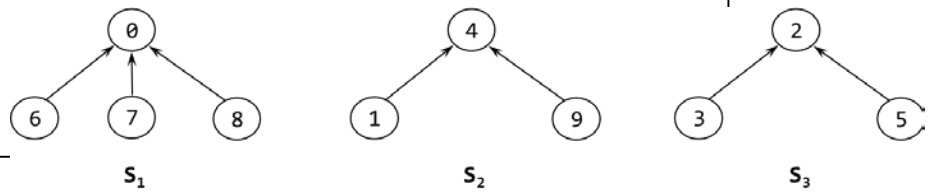
2가 root인 Set

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

합집합과 탐색을 위한 간단한 함수
(Initial union-find Functions)

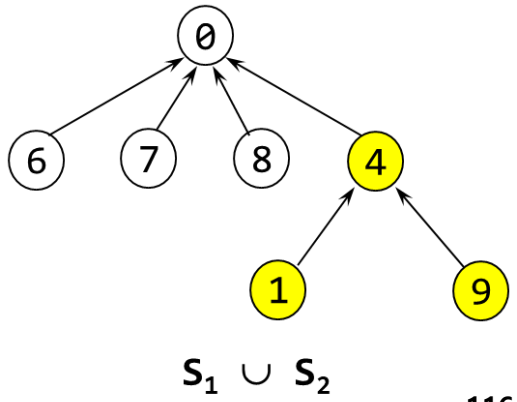
```
int simpleFind(int i)
{
    for(; parent[i] >= 0 ; i = parent[i])
                ;
    return i;
}

void simpleUnion(int i, int j)
{
    parent[i] = j;
}
```

i, j = set의 root node



i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	0	2	0	0	0	4



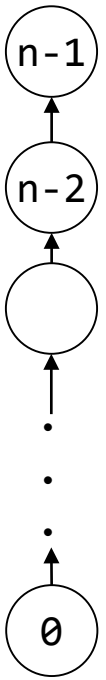
SimpleUnion과 SimpleFind 분석

- **변질(degenerate) 트리**
  - p개의 원소가 각각 p개의 집합에 하나씩 포함된 경우
    - S_i={i} (0 ≤ i < p)
  - 초기상태 : p개의 노드들로 이루어진 포리스트
  - parent[i]= -1 (0 ≤i≤p)

  - union(0,1), union(1,2), union(2,3), ..., union(n-2,n-1)
    - p-1번의 합집합이 O(n) 시간에 수행

  - find(0), find(1), ..., find(n-1) 수행
    - 레벨 i에 있는 원소에 대한 수행 시간 : O(i)

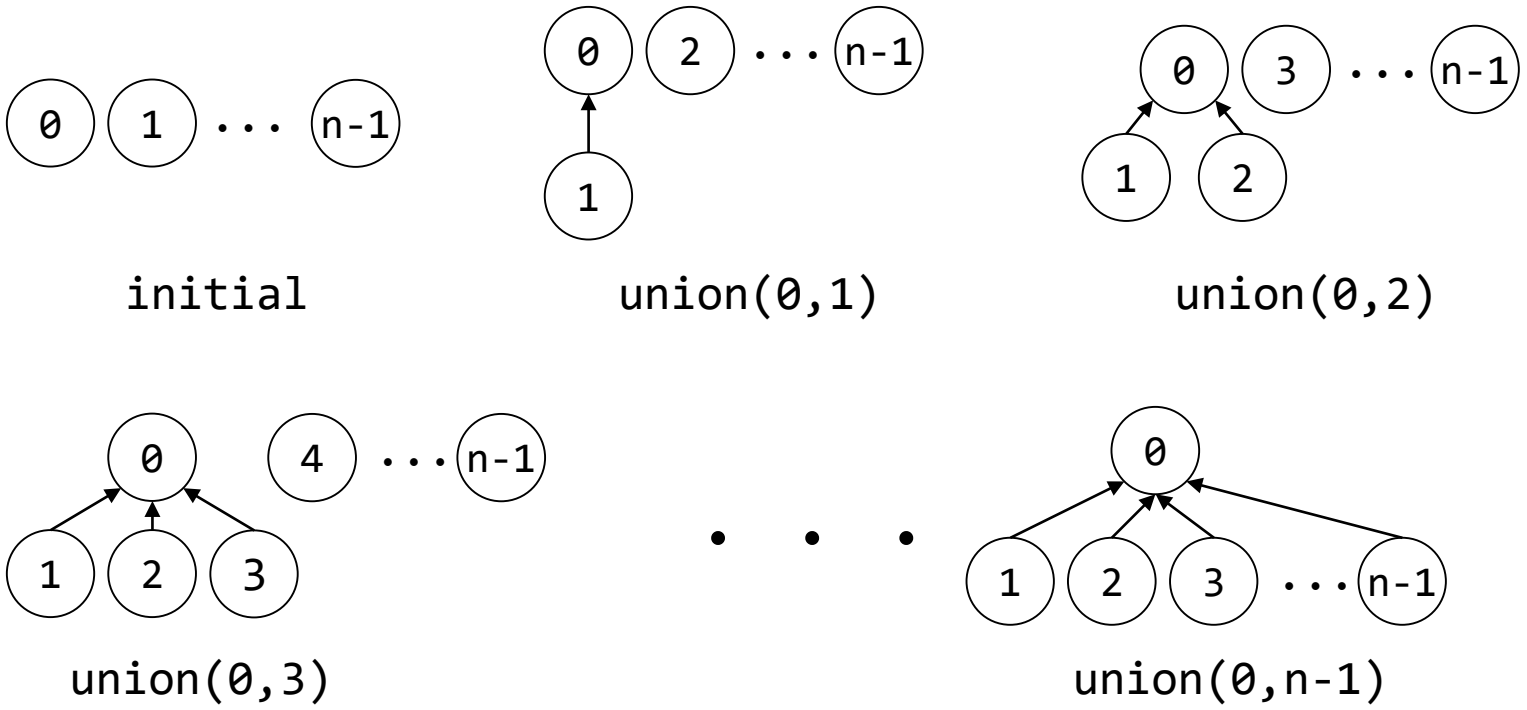
    - n-1번의 Find 수행 : $O(\sum_{i=2}^{n} i)$ = $O(n^2)$



변질(degenerate) 트리

가중 규칙을 적용한 합집합
(Weighting Rule for Union)

- union(i,j)를 위한 가중 규칙
  - 루트 i의 노드 수 < 루트 j의 노드 수: j를 i의 부모로
  - otherwise: i를 j의 부모로

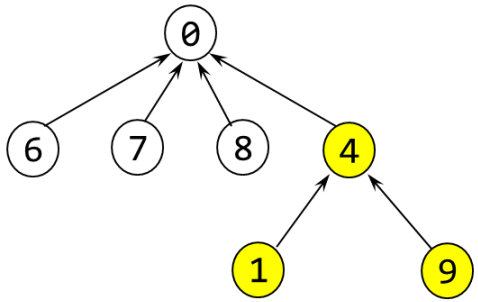


가중 규칙을 적용한 합집합
(Weighting Rule for Union)

- 모든 트리의 루트에 count(계수) 필드 유지
  - 루트의 parent는 -1이므로, 루트의 parent에 -count 유지

```
void weightedUnion(int i , int j)
{
    /* union the sets with roots i and j, i !=j, using
    the weighting rule. parent[i] = -count[i] and
    parent[j] = -count[j] */                ← count[j] = tree j의 node 수
                                            ← parent[j] = -(root j의 node 수)

    int temp = parent[i] + parent[j];       ← union할 두 tree의 node 수(음수)
    if(parent[i]>parent[j]){                ← sizof(tree j) > sizeof(tree i)
        parent[i] = j; /* make j the new root */
        parent[j] = temp;
    }
    else {
        parent[j] = i; /*make i the new root */
        parent[i] = temp;
    }
}
```

tree의 node 수를 알아야 함.



i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-7	4	-1	2	0	2	0	0	0	4

$S_1 \cup S_2$

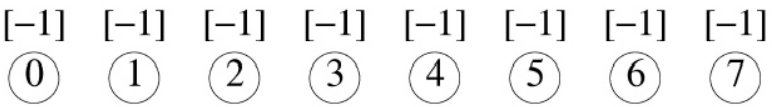
WeightedUnion과 WeightedFind의 분석

- **WeightedUnion : O(1)**

- **WeightedFind(=SimpleFind) : O(log m)**
  - 합집합의 결과가 m개의 노드를 가진 트리의 높이≤ $\lfloor \log_2 m \rfloor + 1$

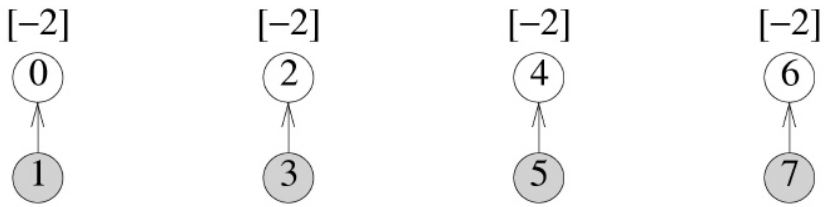
- **u-1개의 합집합 + f개의 탐색 : O($u + f \log u$)**
  - 트리의 노드 수 ≤ u

Weighted Union의 최악의 경우의 트리
(Worst Case Tree)

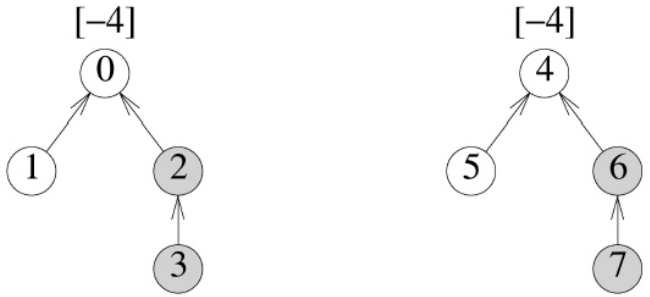


(a) 초기 높이-1 트리

← parent[i]=-count[i]=-1 0≤i<n=8

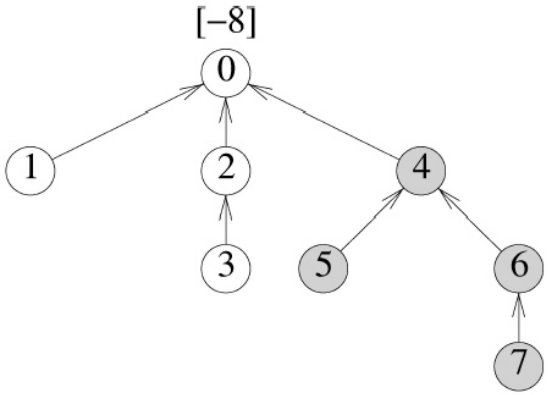


(b) Union(0, 1), (2, 3), (4, 5), (6, 7) 다음의 높이-2 트리



(c) Union(0, 2), (4, 6) 다음의 높이-3 트리

m개의 노드를 가진 트리의 높이≤ $\lfloor \log_2 m \rfloor + 1$

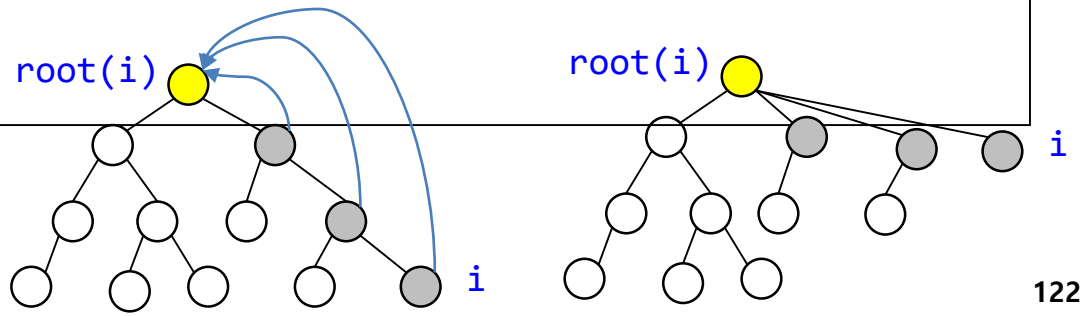


(d) Union(0, 4) 다음의 높이-4 트리

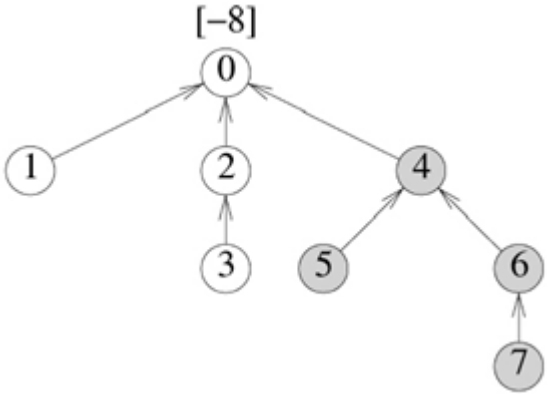
붕괴규칙을 이용한 탐색 알고리즘
(Collapsing Rule)

– 만일 j가 i에서 루트로 가는 경로 상에 있고 parent[i]≠root(i)이면 parent[j]를 root(i)로 지정

```
int collapsingFind(int i)
{
     /* find the root of the tree containing element i. Use the
     collapsing rule to collapse all nodes from i to root */

     int root, trail, lead;
     for(root = i; parent[root] >= 0; root = parent[root])   ← root(i) 찾기
                  ;
     for(trail = i; trail !=root; trail =  lead){
          lead = parent[trail];
          parent[trail] = root;                              ← i -> root까지 경로에 있는
     }                                                         모든 node의 parent를 root로
     return root;
}
```



붕괴규칙을 이용한 탐색 알고리즘
(Collapsing Rule)

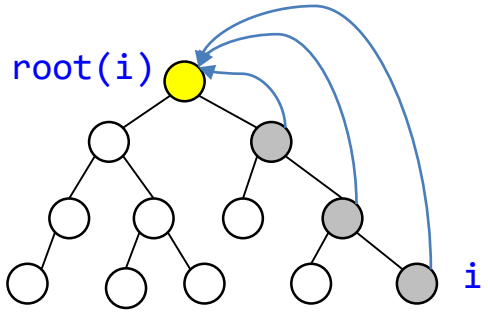


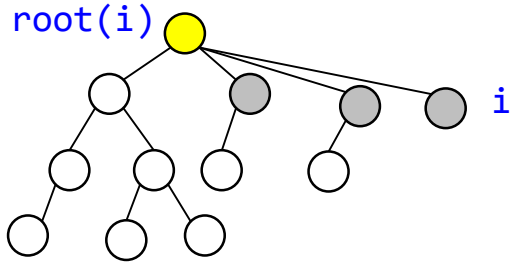
find(7) = node 7의 root 노드를 찾는다.

find(7)을 8번 수행한다고 하면…

simpleFind → find(7) 3개의 parent 링크 필드를 타고 루트로 접근 → 3x8 = 24번 이동 필요

collapsingFind → find(7) 3개의 parent 링크 필드를 타고 루트로 접근 → 모든 링크를 root의 child로(link 재설정) → 3(parent access) + 3(link 재설정) + 7x1 = 13번 이동 필요

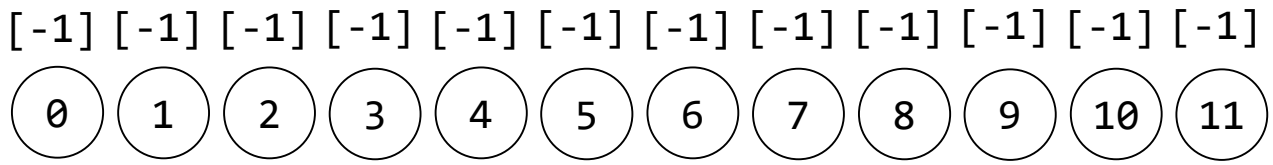




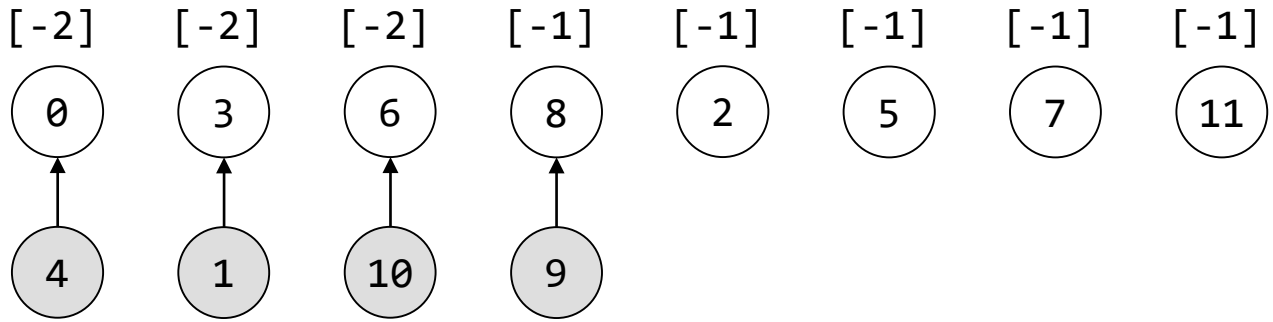
동치 부류의 응용(Application to Equivalence Classes)

- **동치 쌍(equivalence pair)의 처리**
  - 동치 부류(equivalence class)를 집합으로 간주
  - i≡j
    - i와 j를 포함하고 있는 집합 찾기
    - 다른 집합에 포함된 경우 합집합으로 대체
    - 같을 때는 아무 작업도 수행할 필요 없음
  - n개의 변수, m개의 동치 쌍
    - 초기 포리스트 형성 : O(n)
    - 2m개의 탐색, 최대 min{n-1,m}개의 합집합
      : O(n+mα(2m, min{n-1,m}))

동치 부류의 응용 예제(Equivalence Pairs Example)

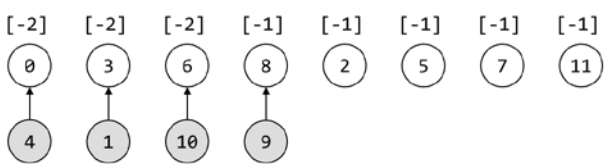


(a) Initial trees

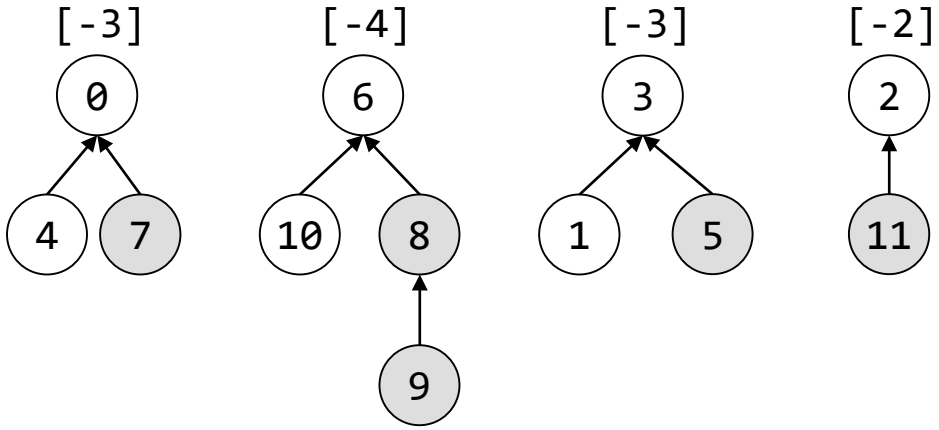


(b) 0≡4, 3≡1, 6≡10, and 8≡9 다음의 높이-2 트리

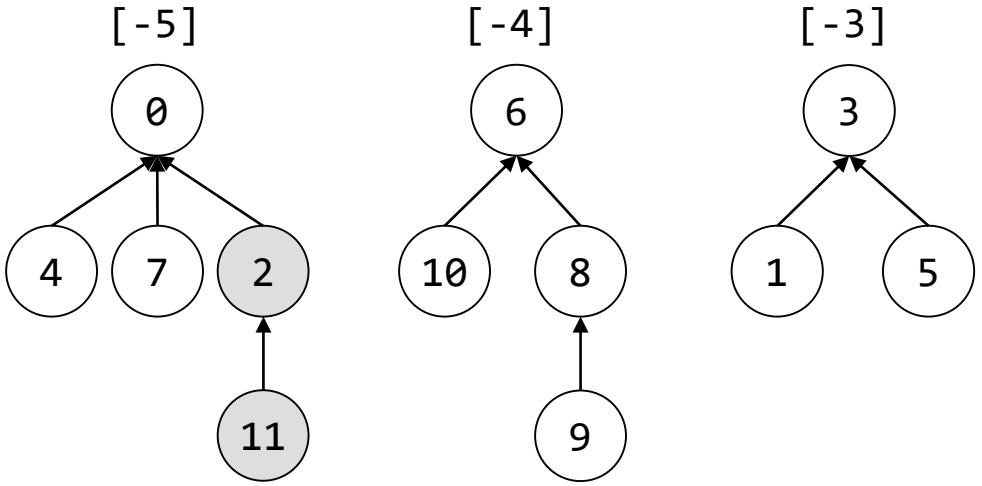
동치 부류의 응용 예제(Equivalence Pairs Example)



(b) 0≡4, 3≡1, 6≡10, and 8≡9 다음의 높이-2 트리



(c) 7≡4, 6≡8, 3≡5,and 2≡11 다음의 트리



(d) 11≡0 다음의 트리