제 5 장 트 리

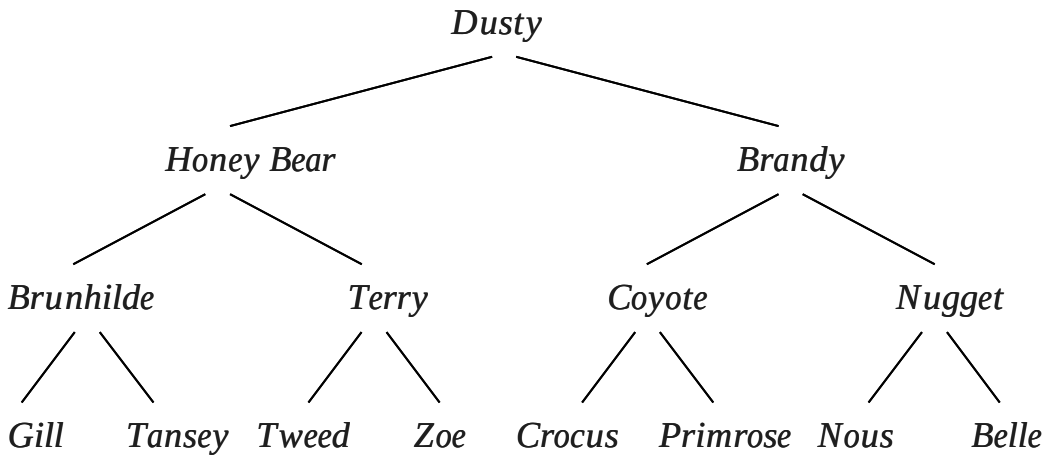
5.1 서 론

5.1.1 기본 용어

□ 트리구조

◆ 정보의 항목들이 가지(branch)로 연결될 수 있게 자료가 조직되는 것

✧ **페이지 192, 그림 5.1 : 가계표의 두 가지 형태**

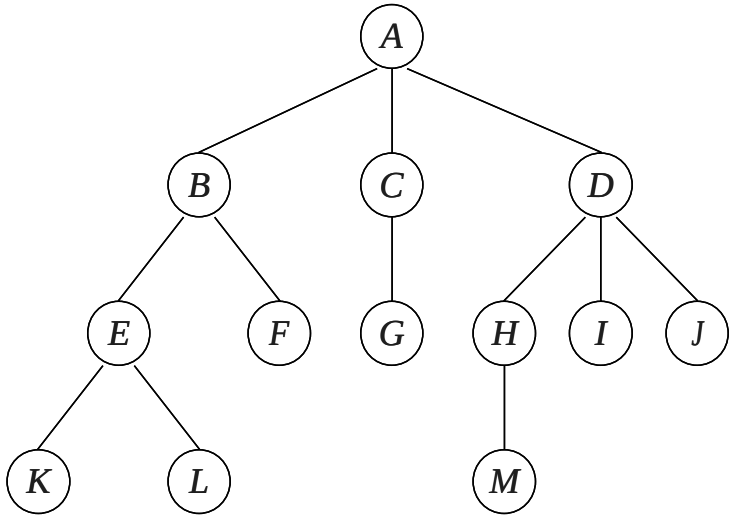


정의

트리는 하나 이상의 노드(node)로 이루어진 유한 집합으로서

- 루트(root)라는 노드가 하나 있고,
- 나머지 노드들은 $n(\geq 0)$개의 분리 집합, T_1, ..., T_n으로 분할될 수 있다. 여기서, T_1, ..., T_n은 각각 하나의 트리이며, 루트의 서브트리(subtree)라고 한다

▶ 페이지 193, 그림 5.2 : 샘플 트리



□ 용어들

- 노드(node) : 한 정보아이템에다 이것으로부터 다른 정보아이템으로 뻗어진 가지를 합친 것
- 차수(degree) : 어떤 노드의 서브트리의 수
  - 분기수라고도 한다
- 단말 노드(terminal node) ↔ 비단말 노드(non-terminal node)
  : 차수 0인 노드로 리프라고도 한다
- 자식노드(child node) : 어떤 노드 *X*의 서브트리들의 루트
- 부모 노드(parent node) : 어떤 노드 *X*의 서브트리들의 루트에 대해 *X*는 부모 노드
- 형제(sibling) : 동일한 부모의 자식들
- 조상(ancestors) : 루트에서부터 어떤 노드에 이르는 경로상의 모든 노드
- 레벨(level) : 루트의 레벨은 1, *l*번째 레벨노드의 자식 노드는 레벨이 *l* + 1
- 포리스트(forest) : $n \geq 0$개의 분리된 트리들의 집합
  트리에서 루트를 제거하면 포리스트가 된다

5.1.2 트리의 표현

□ 리스트 표현

- (*A* (*B* (*E* (*K*, *L*), *F*), *C* (*G*), *D* (*H* (*M*), *I*, *J*)))
- 실제 기억 장소에서의 표현
  - 일정 크기의 메모리 노드 사용

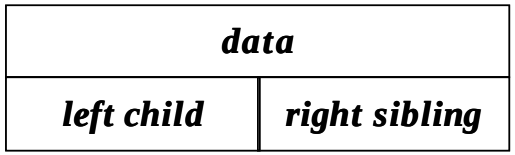
데이터	링크 1	링크 2	...	링크 *n*

- 보조 정리
  - *T*의 차수 = *k*, 노드 수 = *n*
  - 0인 필드의 수 = $n(k-1)+1$

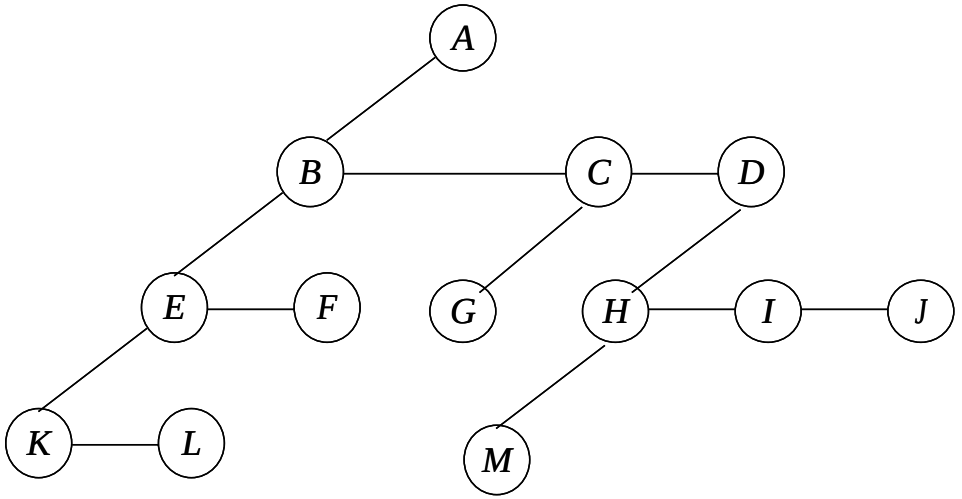
□ 일반적으로 크기가 일정한 노드를 다루는 것이 쉽다

⇒ 왼쪽 자식-오른쪽 형제 표현

✧ **페이지 195, 그림 5.4 : 왼쪽 자식-오른쪽 형제 노드 구조**



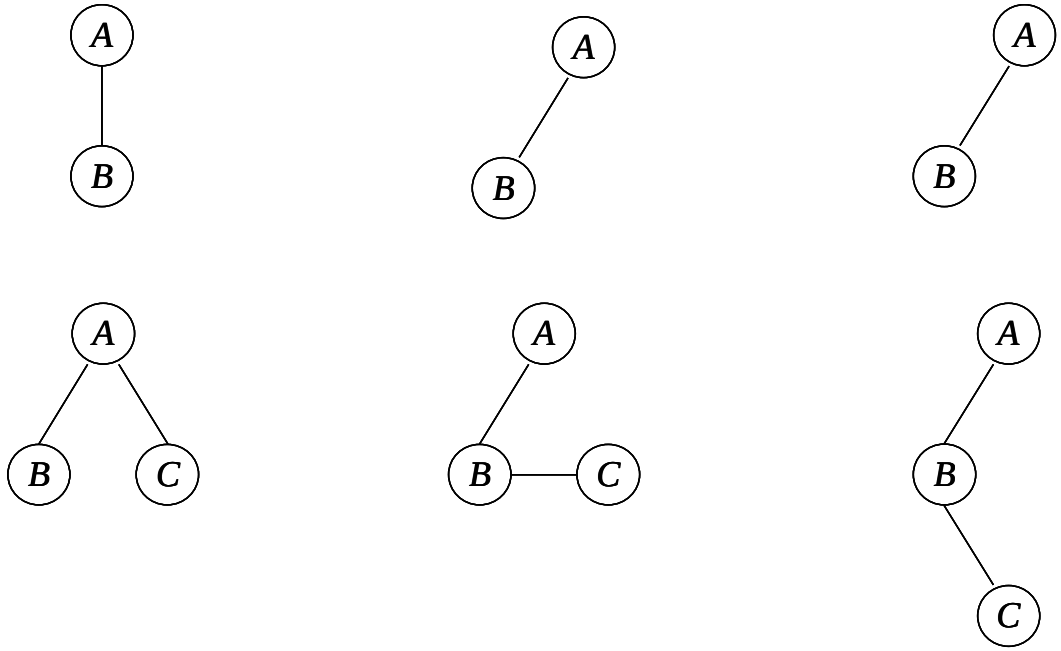
✧ **페이지 195, 그림 5.5 : 트리의 왼쪽 자식-오른쪽 형제 표현**



◆ 차수가 2인 표현

✓ 페이지 196, 그림 5.6 : 트리의 왼쪽 자식-오른쪽 자식의 표현

▶ **페이지 196, 그림 5.7 : 트리 표현**



트리 왼쪽 자식-오른쪽 형제 트리 이진 트리

5.2 이진 트리

□ 이진 트리는 가장 자주 볼 수 있는 트리 구조 중의 하나이다

정의

- 공집합이거나 루트와 왼쪽 서브트리, 오른쪽 서브트리라고 하는 두 개의 분리된 이진 트리로 구성된 노드의 유한 집합

5.2.1 추상 데이터 타입

▶ 페이지 197, 구조 5.1 : 추상 데이터 타입 *Binary_Tree*

structure *Binary_Tree* (줄여서 BinTree)

objects: 공백이거나 루트 노드, 왼쪽 Binary_Tree, 오른쪽 Binary_Tree로
구성되는 노드들의 유한집합

functions:

모든 *bt, bt1, bt2* ∈*BinTree, item* ∈*element*

BinTree Create()	::= 공백 이진 트리를 생성
Boolean IsEmpty(*bt*)	::= if (bt == 공백 이진 트리) return TRUE else return FALSE
BinTree MakeBT(*bt1*, item, *bt2*)	::= 왼쪽 서브트리가 bt1, 오른쪽 서브트리가 bt2, 루트는 데이타를 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 데이타를 반환
BinTree Rchild(*bt*)	::= if (IsEmpty(bt)) return 에러 else bt의 오른쪽 서브트리를 반환

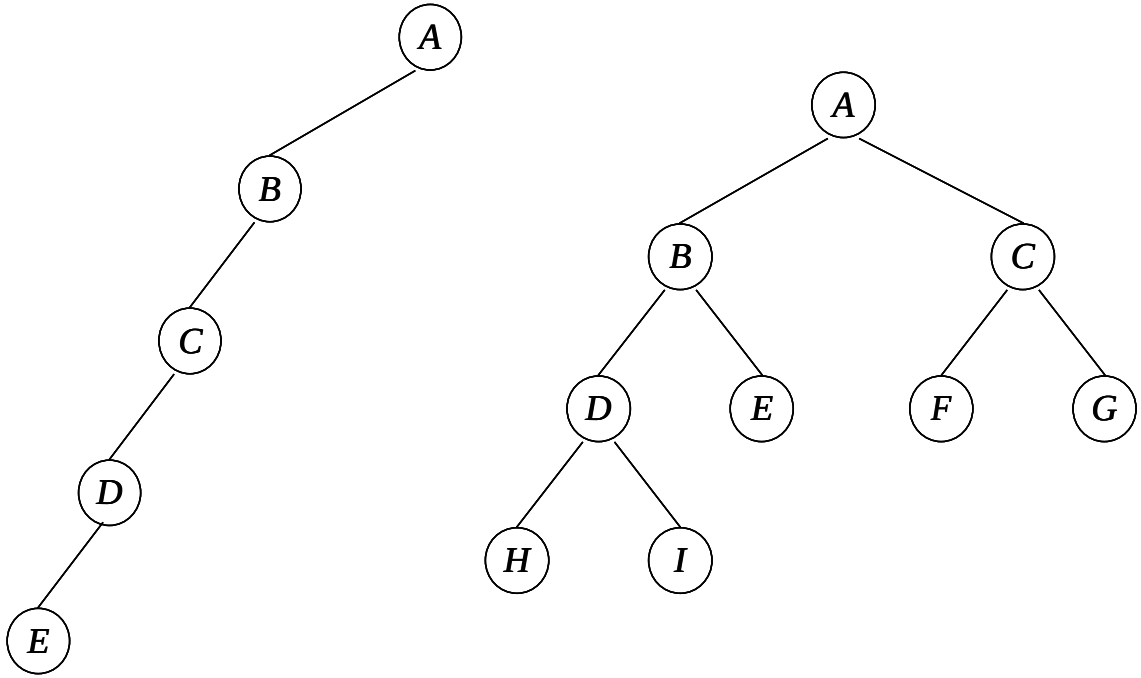
□ 이진 트리 vs. 트리

- 공백 이진 트리가 있다 / 0개의 노드를 가진 트리는 없다
- 이진 트리에서는 자식의 순서를 구분한다

✧ **페이지 198, 그림 5.8 : 서로 다른 두 이진 트리**

□ 특수한 이진 트리들

✧ **페이지 198, 그림 5.9 : 경사 트리와 완전 이진 트리**



경사 트리 완전 이진 트리

5.2.2 이진 트리의 특성

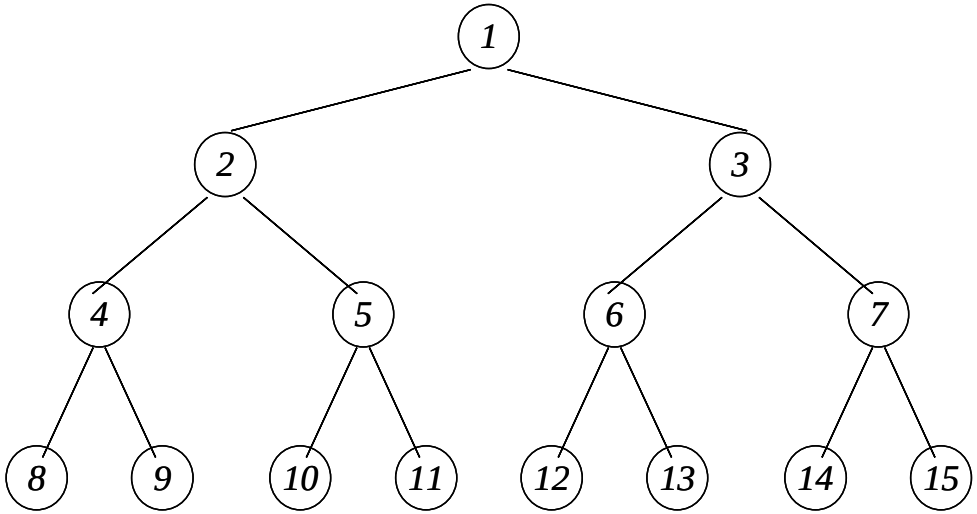
깊이 k인 이진 트리에서 최대 노드수

- 이진 트리에서 레벨 i에서의 최대 노드수는 2^{i-1}, $i \geq 1$
- 깊이가 k인 이진 트리가 가질 수 있는 최대 노드수는 2^{k-1} , $k \geq 1$

단말 노드 수와 차수 2인 노드 수와의 상관 관계

- 모든 이진 트리 T에 대하여 n_0가 단말 노드수이고 n_2가 차수 2인 노드 수이면 $n_0 = n_2 + 1$
- n_1을 차수 1인 노드수라하면 총 노드수는 $n = n_0 + n_1 + n_2$

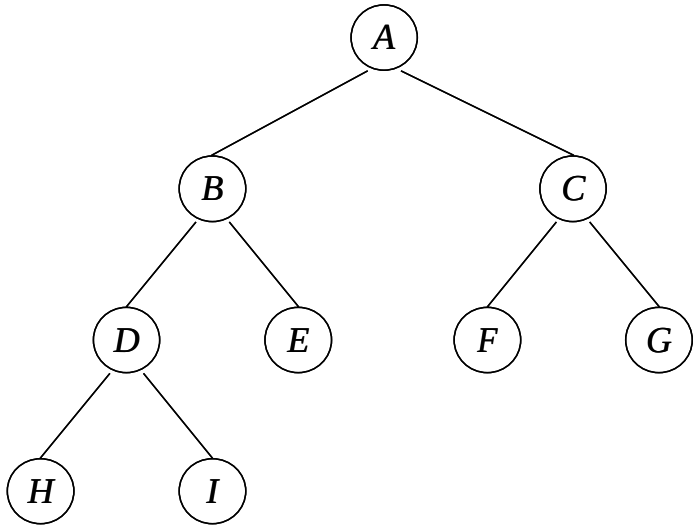
- 깊이가 *k*인 포화 이진 트리(full binary tree)
  - 깊이가 *k*, 노드수가 $2^{k}-1$ $(k \geq 0)$ 인 이진 트리
  - ✧ **페이지 201, 그림 5.10 : 순서 노드 번호를 붙인 깊이 4의 포화 이진 트리**



완전 이진 트리(complete binary tree)

- 깊이가 *k*이고 노드수가 *n*인 이진 트리의 각 노드들이 깊이 *k*인 포화 이진 트리에서 1부터 *n*까지의 번호를 붙인 노드들과 일대일로 일치하면 이 트리는 완전 이진 트리이다

✧ **페이지 198, 그림 5.9 : 완전 이진 트리**



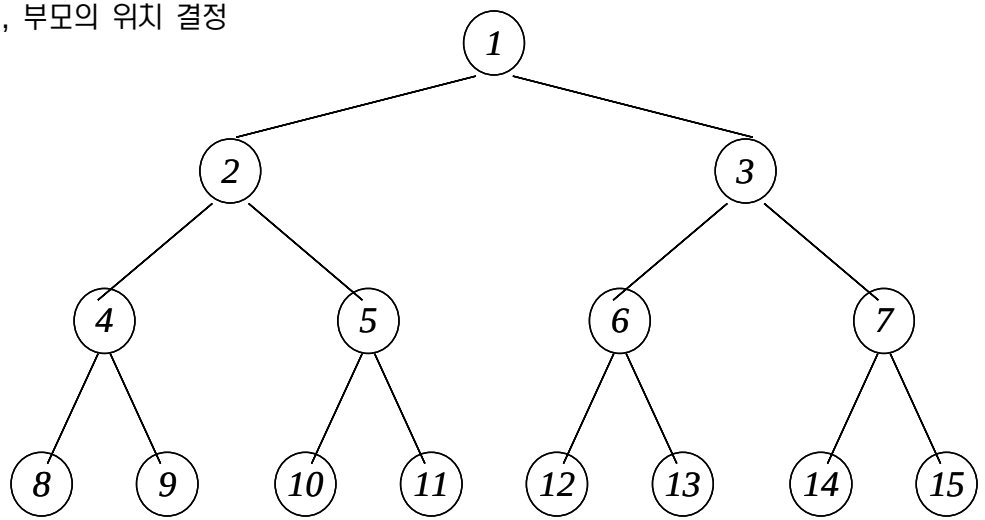
5.2.3 이진 트리 표현

5.2.3.1 배열 표현

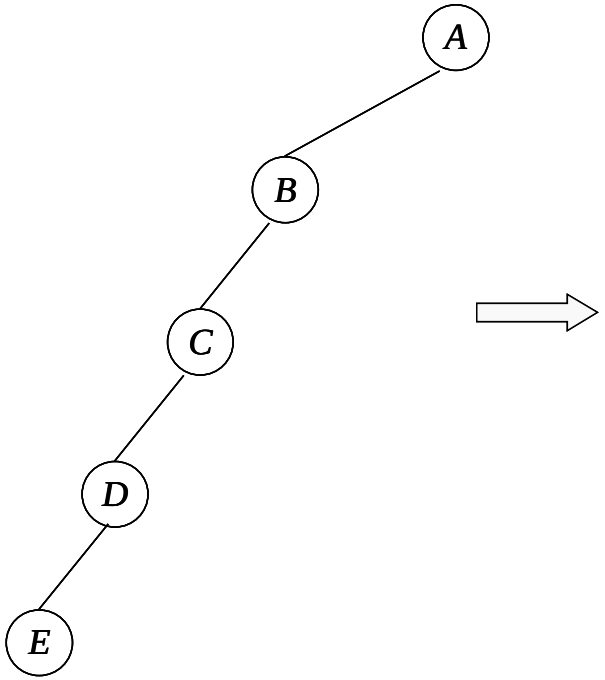
□ 일차원 배열의 사용

□ 이진 트리의 순차적 표현 방법

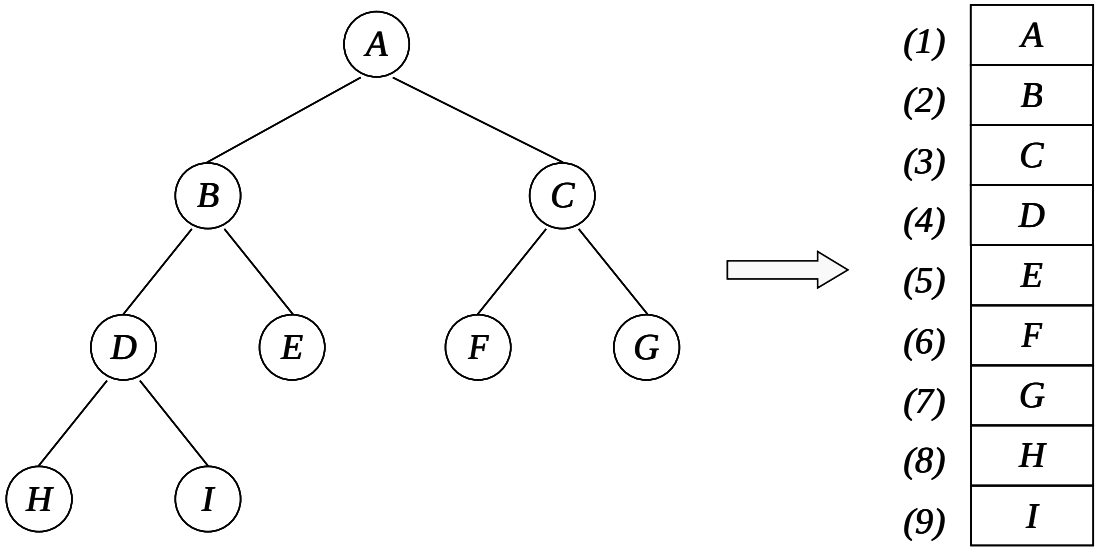
- 레벨 1인 노드부터 점차 높은 레벨 노드 순서로 차례로 번호를 붙이되 같은 레벨에서는 왼쪽에서 오른쪽으로 번호를 붙인다
- 번호가 *i*인 노드를 *TREE*(*i*)에 저장하는 식으로 각 노드를 배열 *TREE*에 저장
- 노드 *i*의 왼쪽 자식, 오른쪽 자식, 부모의 위치 결정



▶ **페이지 202, 그림 5.11 : 이진 트리의 배열 표현**



(1)	A
(2)	B
(3)	-
(4)	C
(5)	-
(6)	-
(7)	-
(8)	D
(9)	-
⋮	⋮
(16)	E


A
B
C
D
E
F
G
H
I
(1) A
(2) B
(3) C
(4) D
(5) E
(6) F
(7) G
(8) H
(9) I

□ n개의 노드를 가진 완전 이진 트리를 순차적으로 표현하면

- *parent*(*i*)는 i ≠ 1일 때 $\lfloor i / 2 \rfloor$의 위치에 있게 된다
- *lchild*(*i*)는 $2i \leq n$일 때 $2i$의 위치에 있게 된다
- *rchild*(*i*)는 $2i + 1 \leq n$일 때 $2i + 1$의 위치에 있게 된다

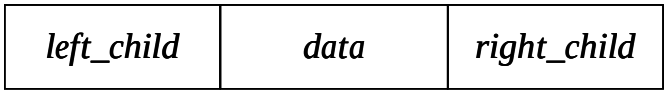
□ 단점

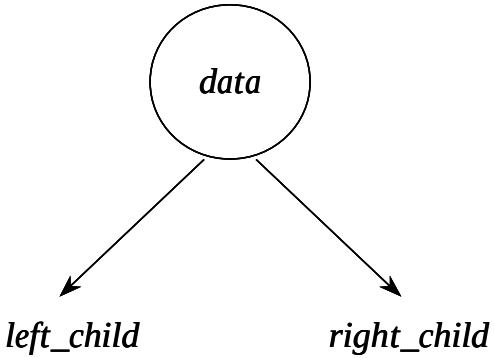
- 완전 이진 트리의 경우는 공간의 낭비가 전혀 없으나 일반 이진 트리에서는 낭비가 매우 크다
- 최악의 경우 2^{k-1} 기억 장소 중 k만을 사용
- 트리의 중간에 노드를 삽입하거나 제거할 때 이 노드들의 레벨 변경에 따라 많은 노드들의 위치가 변해야 한다

5.2.3.2 링크 표현

▶ **페이지 203, 그림 5.12 : 이진 트리의 노드 표현**

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
};
```

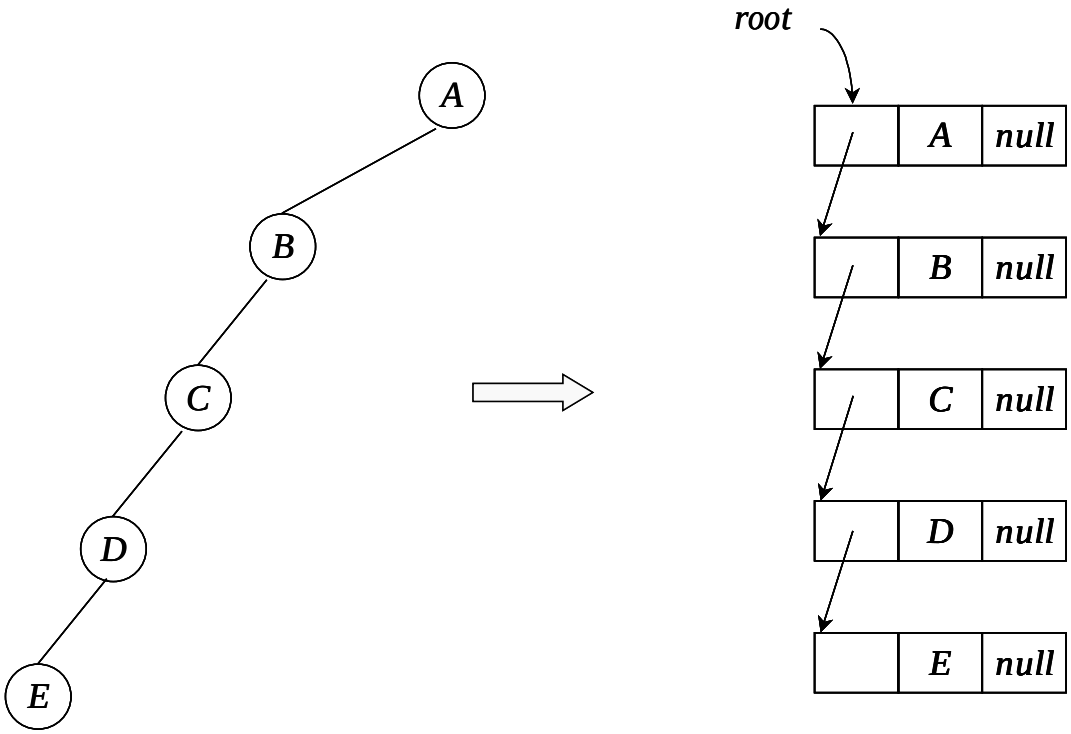


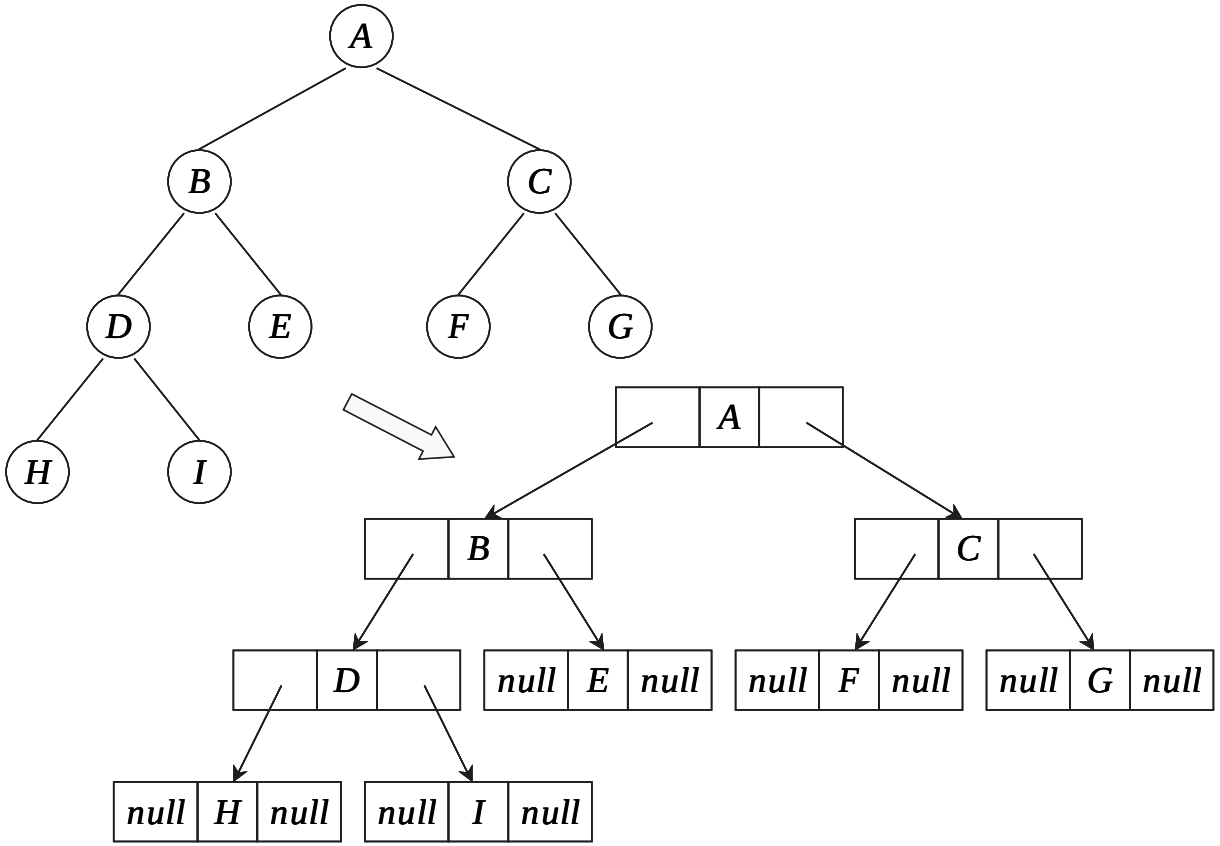


□ 단점

- 한 노드의 부모를 알기 어렵다
- *parent* 필드를 둠으로써 해결한다

▶ **페이지 203, 그림 5.13 : 이진 트리에 대한 링크 표현**




A
B
C
D
E
F
G
H
I
A
B
C
D
null E null
null F null
null G null
null H null
null I null

5.3 이진 트리 순회

☐ 순회란

- 트리의 각 노드를 정확히 한번씩 방문
- 완전한 순회는 트리에 있는 정보의 선형 순서를 생성

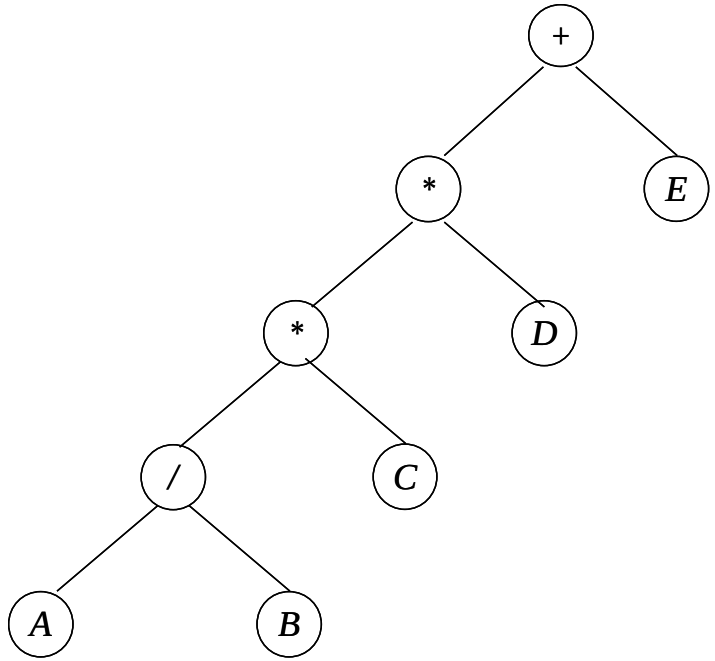
☐ 가능한 순회 방법들

- LVR, LRV, VLR, VRL, RVL, RLV
  - L : moving left, V : visiting the node, R : moving right

☐ 왼쪽을 오른쪽 보다 먼저 순회하기로 한다면

- 중위 순회(inorder traversal) - LVR
- 후위 순회(postorder traversal) - LRV
- 전위 순회(preorder traversal) - VLR

▶ **페이지 205, 그림 5.15 : 산술식을 표현한 이진 트리**

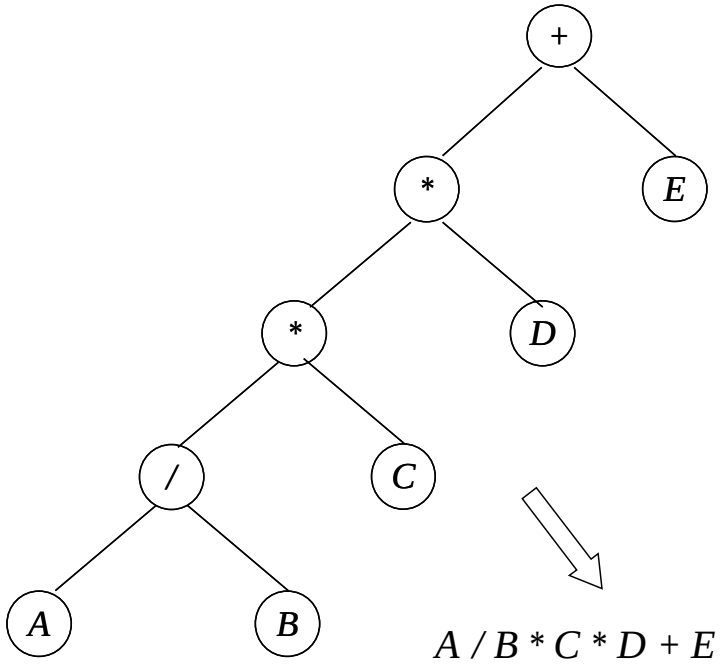


□ 중위 순회

- 왼쪽 서브트리를 중위 순회한 뒤 루트를 방문하고 그 다음 오른쪽 서브트리를 중위 순회하는 방법

✧ **페이지 206, 프로그램 5.1 : 이진 트리의 중위 순회**

```
void inorder(tree_pointer ptr)
/* 중위 트리 순회 */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child);
    }
}
```

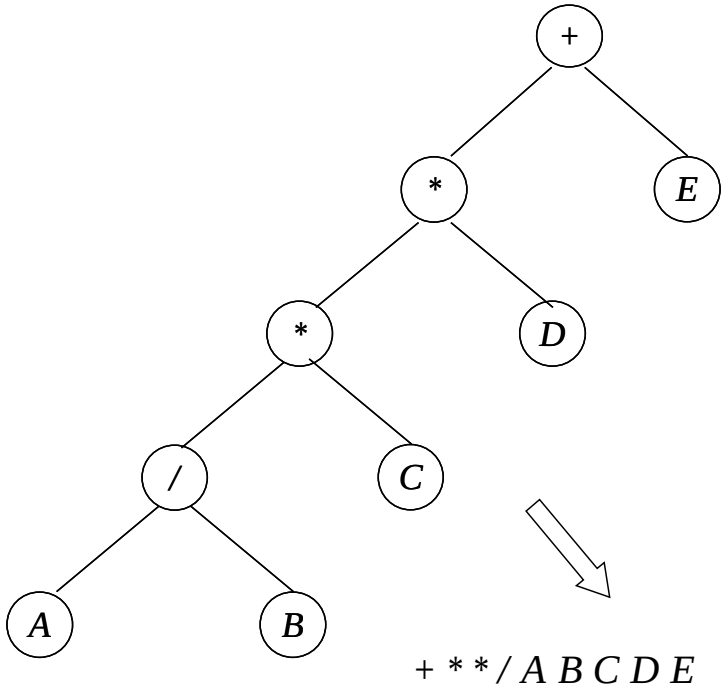


□ 전위 순회

- 루트 노드를 방문하고 왼쪽 서브 트리를 전위 순회한 뒤에 오른쪽 서브 트리를 전위순회

✧ **페이지 207, 프로그램 5.2 : 이진 트리의 전위 순회**

```
void preorder(tree_pointer ptr)
/* 전위 트리 순회 */
{
     if (ptr) {
          printf("%d", ptr->data);
          preorder(ptr->left_child);
          preorder(ptr->right_child);
   }
}
```

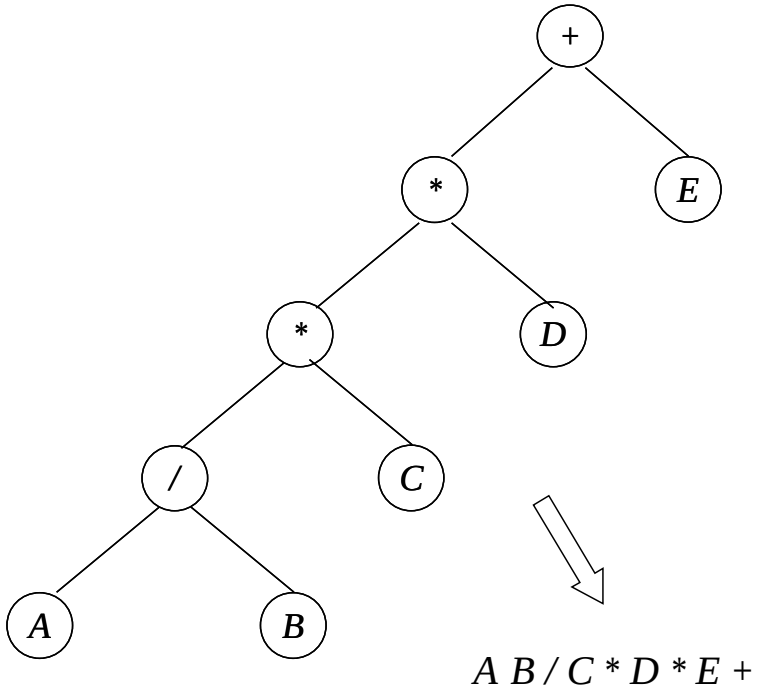


□ 후위 순회

- 왼쪽 서브 트리를 먼저 순회한 다음 오른쪽 서브트리를 순회하고 마지막으로 루트 트리 방문

✧ **페이지 207 ~ 208, 프로그램 5.3 : 이진 트리의 후위 순회**

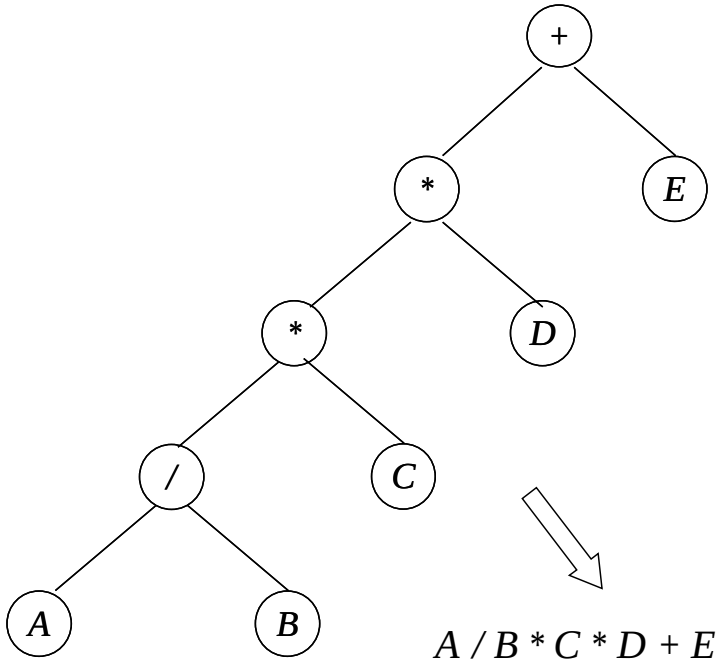
```
void postorder(tree_pointer ptr)
/* 후위 트리 순회 */
{
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```



□ 반복적 중위 순회

✧ 페이지 208, 프로그램 5.4 : 반복적 중위 순회

```
void iter_inorder(tree_pointer node)
{
    int top = -1;     /* 스택 초기화 */
    tree_pointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node = node->left_child)
            add(&top, node);   /* 스택에 삽입 */
        node = delete(&top);   /* 스택에서 삭제 */
        if (!node) break;              /* 공백 스택 */
        printf("%d", node->data);
        node = node->right_child;
    }
}
```



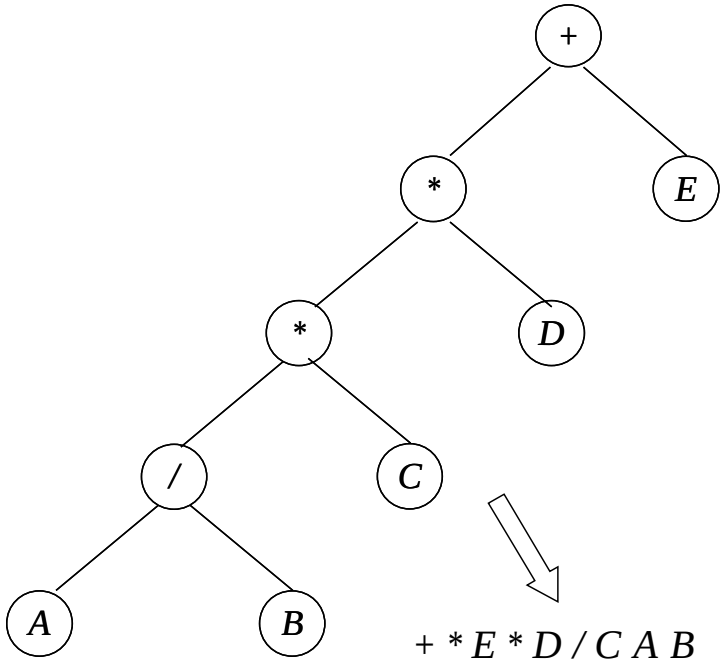
□ 레벨 순서 순회

- ◆ 순차적 표현 방법을 따른 노드의 방문

→ 루트, 왼쪽 자식, 오른쪽 자식의 순서로 방문

✧ **페이지 209 ~ 210, 프로그램 5.5 : 이진 트리의 레벨 순서 순회**

```
void level_order(tree_pointer ptr)
/* 레벨 순서 트리 순회 */
{
     int front = 0, rear = 0;
     tree_pointer 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->left_child)
                    addq(front, &rear, ptr->left_child);
               if (ptr->right_child)
                    addq(front, &rear, ptr->right_child);
          }
          else break;
     }
}
```

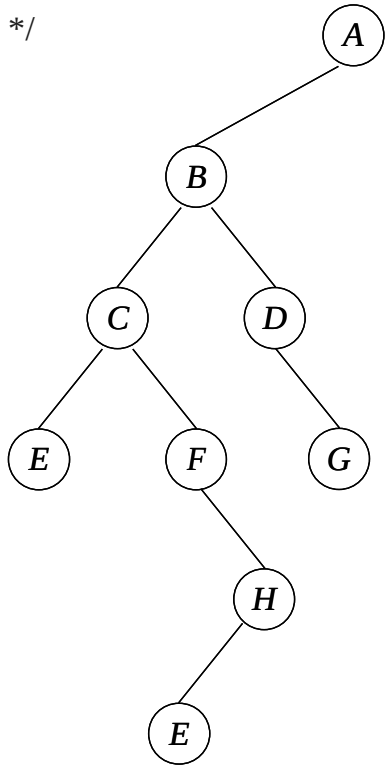


5.4 이진 트리 추가 연산

5.4.1 이진 트리의 복사

▶ **페이지 211, 프로그램 5.6 : 이진 트리의 복사 ← postorder의 변형**

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

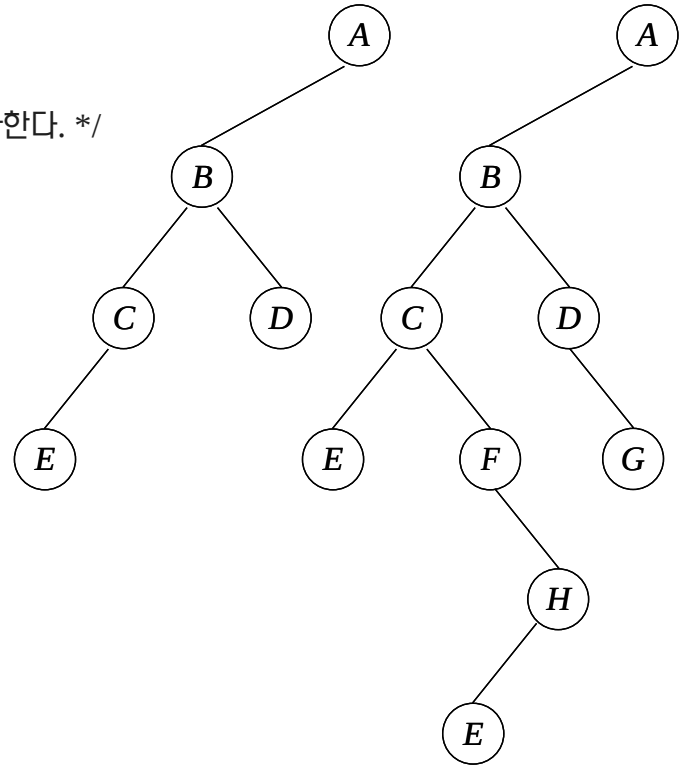


5.4.2 동일성 검사

□ 두 이진 트리가 같은 구조를 가지고 대응되는 노드에 있는 정보들이 일치하면 두 트리는 동일하다

▶ 페이지 212, 프로그램 5.7 : 이진 트리의 동일 검사 ← 전위 순회의 변형

```
int equal(tree_pointer first, tree_pointer second)
{
/* 두 이진 트리가 동일하면 TRUE, 그렇지 않으면 FALSE를 반환한다. */
    return ((!first && !second) ||
            (first && second &&
            (first->data == second->data) &&
            equal(first->left_child, second->left_child) &&
            equal(first->right_child, second->right_child));
}
```



5.4.3 만족성 문제

☐ 조건식의 표현

- 변수 $x_1, x_2, x_3, \ldots, x_n$과 연산자 $\wedge$(and), $\vee$(or), $\neg$(not)으로 이루어지는 식을 다룬다
- 규칙
  - 변수 자체도 하나의 식이다
  - x, y 식일 때 $x \wedge y$, $x \vee y$, $\neg x$도 식이다
  - 연산 순서는 $\neg$, $\wedge$, $\vee$의 순으로 괄호로서 이 순서를 바꿀 수 있다

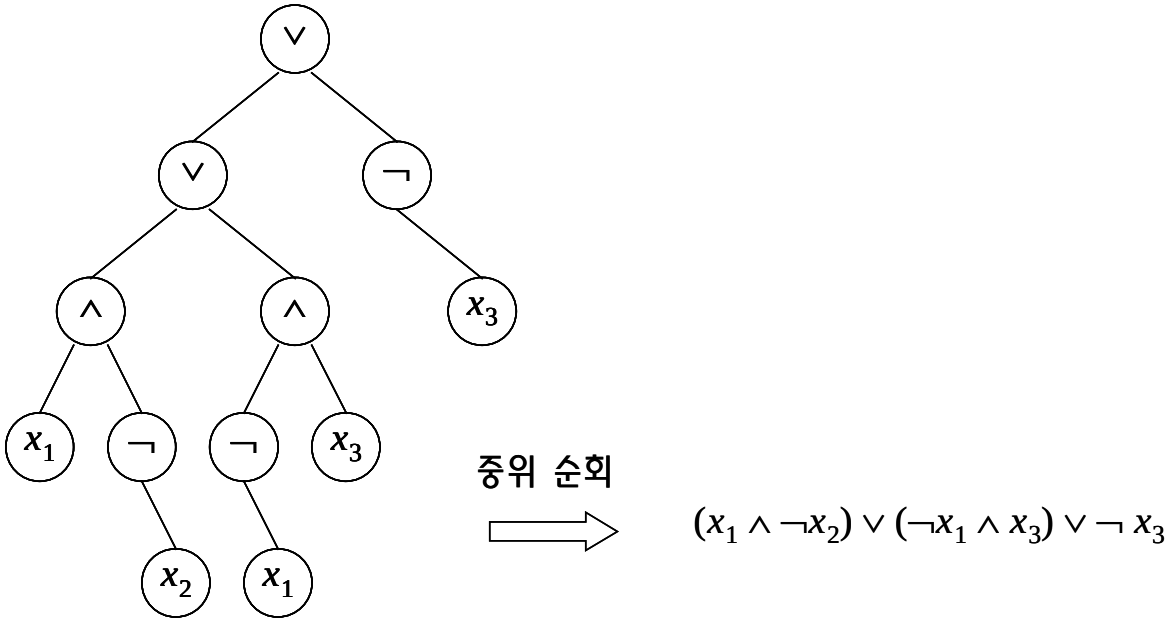
☐ 만족성 문제

- 식의 값이 참이 되도록, 변수에 값을 지정할 수 있는 방법이 있는가를 결정하는 것

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

□ 이진 트리 표현

✧ **페이지 213, 그림 5.18 : 이진 트리로 표현된 명제식**



✧ **페이지 214, 그림 5.19 : 만족성 문제를 위한 노드 & 페이지 214, 노드 구조의 선언**


```
typedef enum { not, and, or, true, false } logical;
typedef struct node *tree_pointer;
typedef struct node {
    tree_pointer    left_child;
    logical         data;
    short int       value;
    tree_pointer    right_child;
};
```

□ n개의 변수에 대해 2^n의 조합 가능

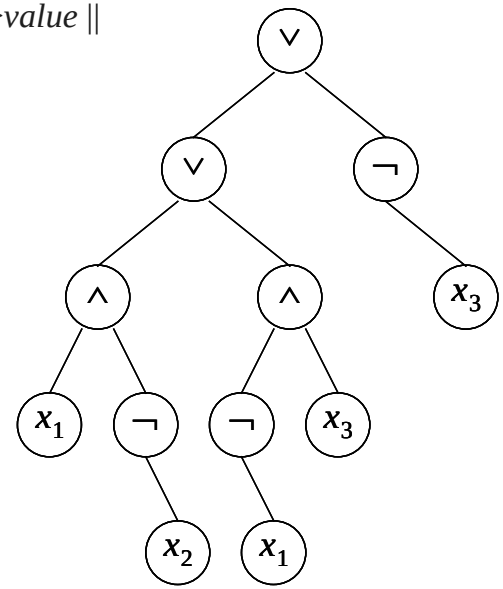
- $O(g2^n)$의 시간 필요

✧ **페이지 214, 프로그램 5.8 : 만족성 알고리즘의 첫번째 버전**

```
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");
```

▶ 페이지 215, 프로그램 5.9 : *post_order_eval* 함수 ← 후위 순회의 수정

```
void post_order_eval(tree_pointer node)
/* 명제 해석 트리를 계산하기 위해 수정된 후의 순회 */
{
    if (node) {
        post_order_eval(node->left_child); post_order_eval(node->right_child);
        switch (node->data) {
            case not:   node->value = !node->right_child->value; break;
            case and:   node->value = node->right_child->value &&
                            node->left_child->value;
                        break;
            case or: node->value = node->right_child->value ||
                            node->left_child->value;
                        break;
            case true:  node->value = TRUE;
            case false: node->value = FALSE;
        }
    }
}
```



5.5 스레드 이진 트리

□ 이진 트리

- 실제 포인터보다 널 링크가 더 많다

→ 링크를 다른 노드를 가리키는 포인터로 대치

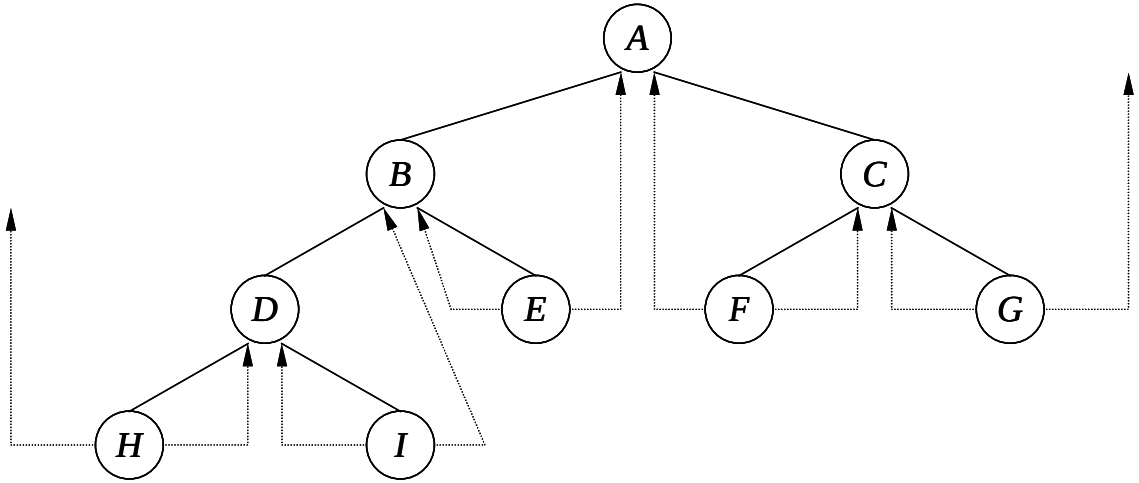
⇒ 스레드 (thread)

- left child가 널이면
  - 트리를 중위 순회시 이전에 방문할 노드를 가리킨다

  → 중위 선행자(inorder predecessor)
- right child가 널이면
  - 트리를 중위 순회시 전에 방문한 노드를 가리킨다

  → 중위 후속자(inorder successor)

▶ **페이지 217, 그림 5.21 : 스레드 트리**



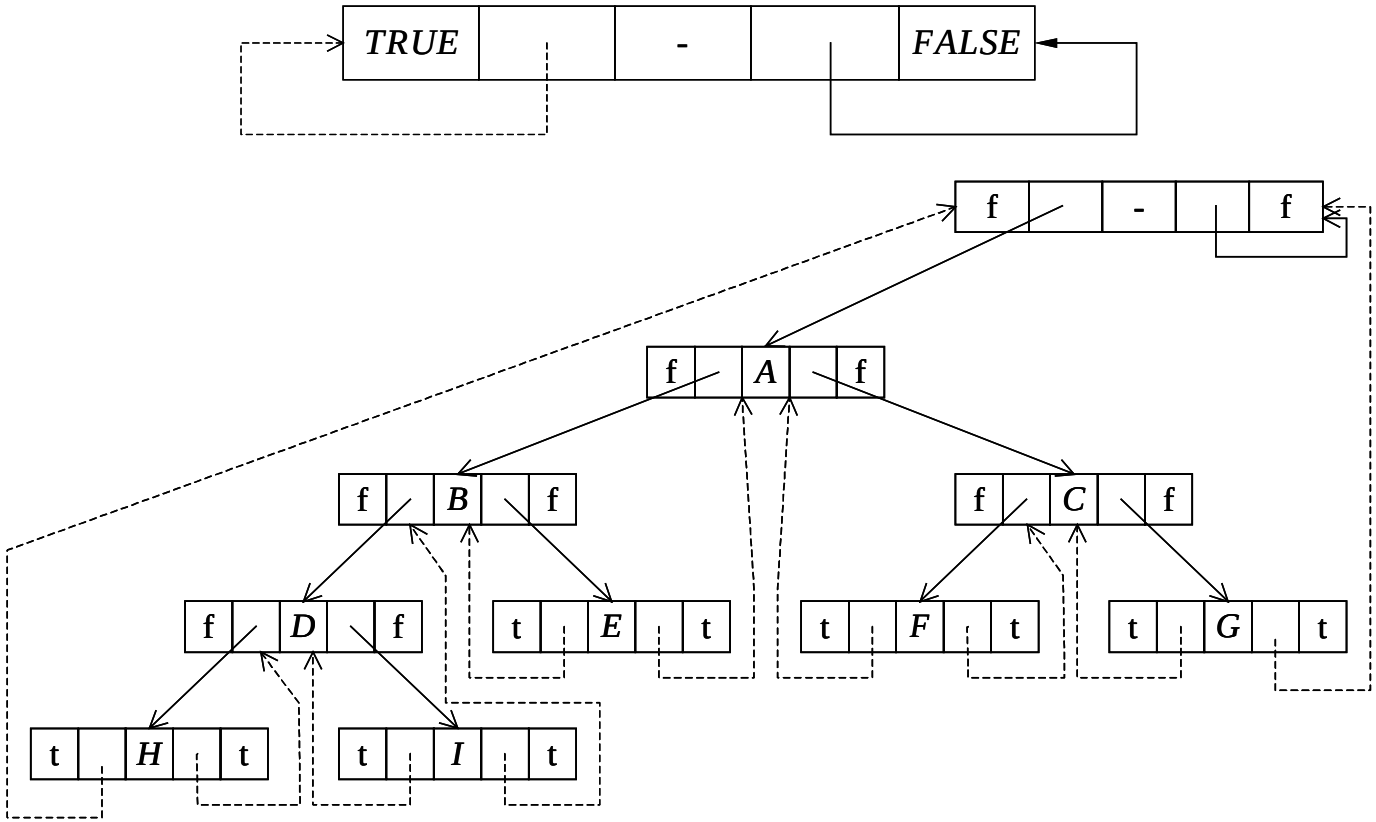
□ 스레드 이진 트리 표현

✧ **페이지 217, 노드 구조 선언 & 페이지 218, 그림 5.22 : 공백 스레드 트리**

left_thread	*left_child*	*data*	*right_child*	*right_thread*

```
typedef struct threaded_tree *thread_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;
}
```

✧ **페이지 218, 그림 5.23 : 스레드 트리의 메모리 표현**

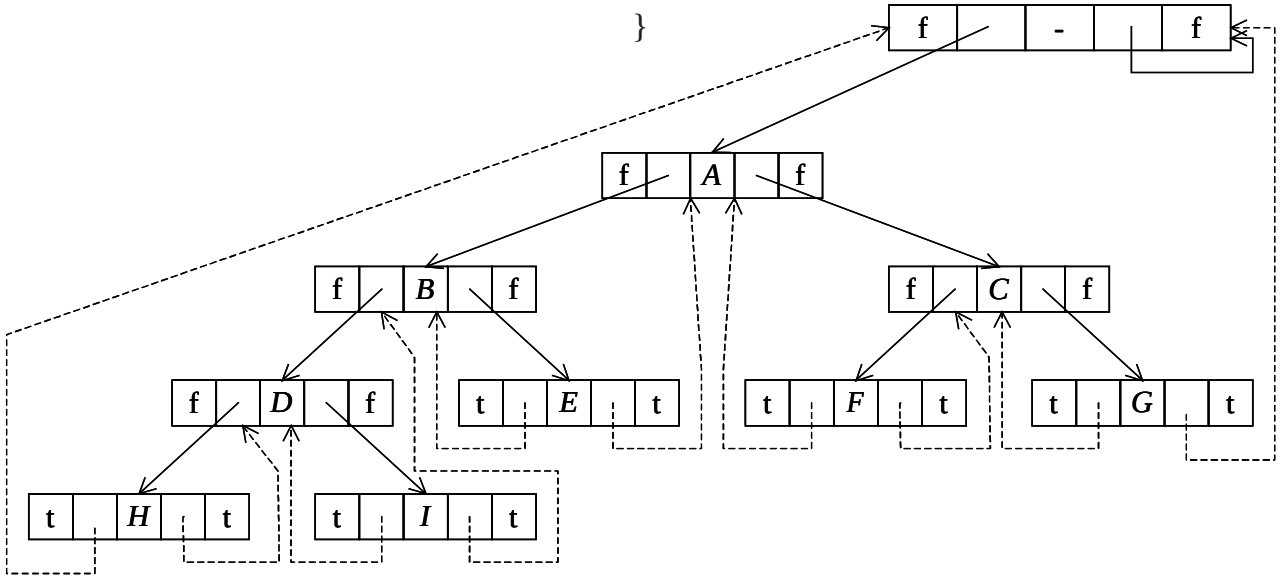


□ 스레드 이진 트리의 중위 순회

✧ 페이지 219, 프로그램 5.10 : 한 노드의 중위 후속자를 찾는 함수
& 페이지 219 ~ 220, 프로그램 5.11 : 스레드 이진 트리의 중위 순회

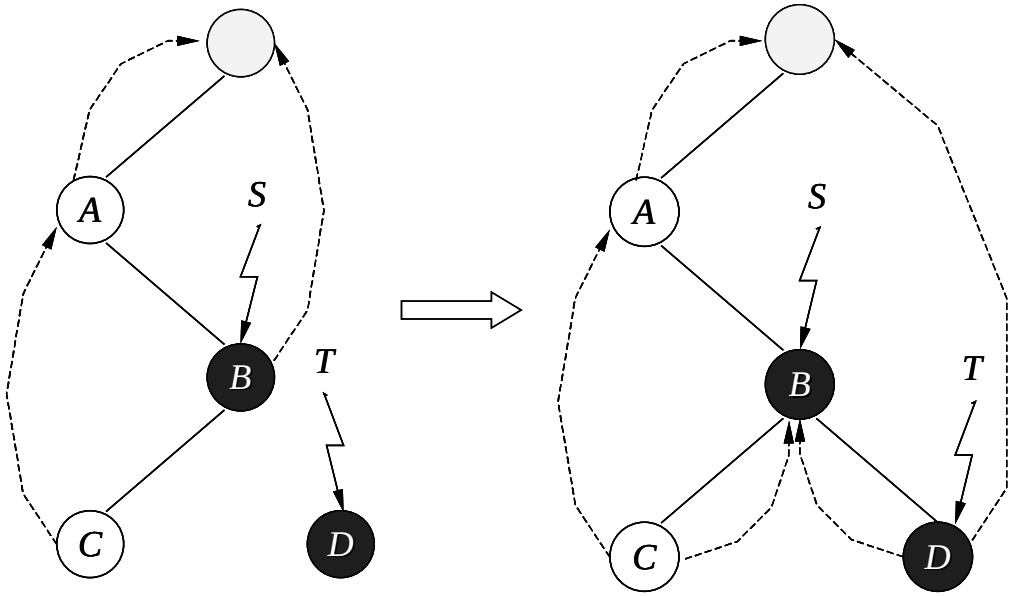
```
threaded_pointer insucc(threaded_poninter tree)
/* 스레드 이진 트리에서 중위 후속자를 찾는다. */
{
    threaded_pointer temp;
    temp = tree->right_child;
    if (!tree->right_thread)
        while (!temp->left_thread)
            temp = temp->left_child;
    return temp;
}
```

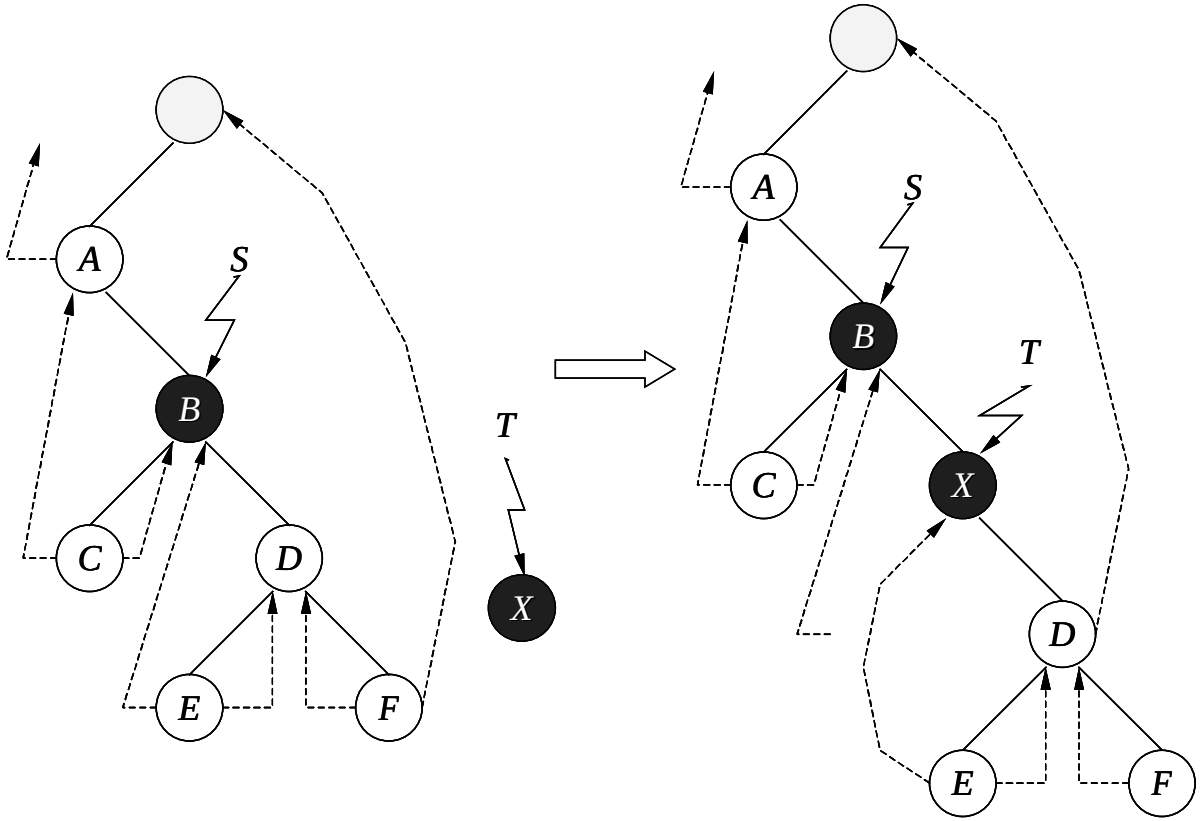
```
void tinorder(threaded_pointer tree)
/* 스레드 이진 트리의 중위 순회 */
{
    threaded_pointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp = tree) break;
        printf("%3c", temp->data);
    }
}
```



□ 스레드 이진 트리의 노드 삽입

✧ **페이지 221, 그림 5.24 : 스레드 이진 트리에서 *parent*의 오른쪽 자식으로 *child*를 삽입**




A
S
B
C
D
E
F
T
X
A
S
B
T
C
X
D
E
F

✧ **페이지 220 ~ 221, 프로그램 5.12 : 스레드 이진 트리에서의 오른쪽 삽입**

```
void insert_right(threaded_pointer parent, threaded_pointer child)
/* 스레드 이진 트리에서 child를 parent의 오른쪽 자식으로 삽입 */
{
    threaded_pointer temp;
    child->right_child = parent->right_child;
    child->right_thread = parent->right_thread;
    child->left_child = parent; child->left_thread = TRUE;
    parent->right_child = child; parent->right_thread = FALSE;
    if (!child->right_thread) {
        temp = insucc(child);
        temp->left_child = child;
    }
}
```

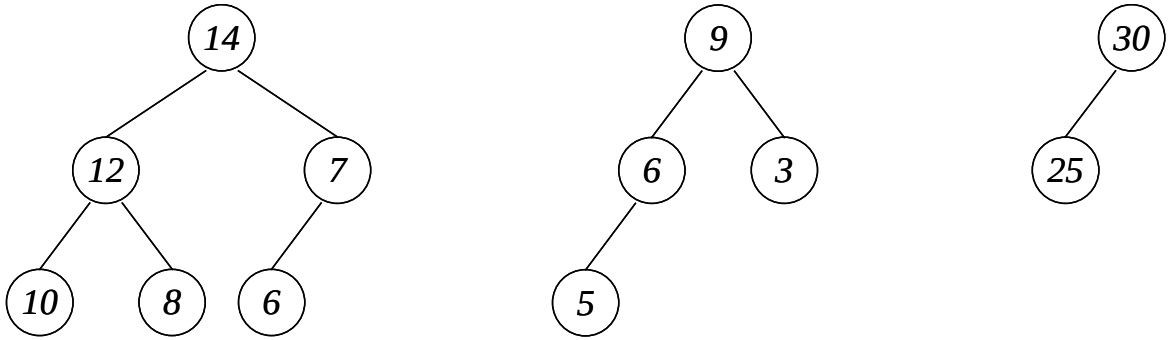
5.6 히 프

5.6.1 히프 추상 데이터 타입

정의

- 최대 트리
  - 각 노드의 키 값이 그 자식의 키 값보다 작지 않은 트리
- 최대 히프
  - 최대 트리이면서 완전 이진 트리
- 최소 트리
  - 각 노드의 키 값이 그 자식의 키 값보다 크지 않은 트리
- 최소 히프
  - 최소 트리이면서 완전 이진 트리

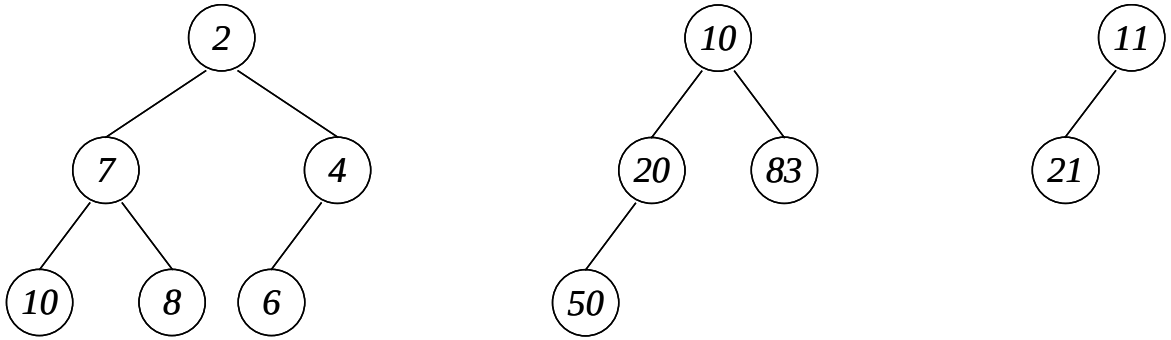
▶ **페이지 223, 그림 5.25 : 최대 히프의 예**



□ 최대 히프에 대한 기본 연산

- 공백 히프의 생성
- 히프에 새로운 원소의 삽입
- 히프에서 가장 큰 원소의 삭제

▶ **페이지 223, 그림 5.26 : 최소 히프의 예**



▶ 페이지 223 ~ 224, 구조 5.2 : 추상 데이터 타입 *MaxHeap*

structure *MaxHeap*

objects : 각 노드의 값이 그 자식들의 것보다 작지 않도록 조직된 $n > 0$ 원소의 완전 이진 트리

functions :

모든 *heap* ∈ *MaxHeap*, *item* ∈ *Element*, *n*, *max_size* ∈ *integer*

MaxHeap Create(*max_size*)	::= 최대 *max_size*개의 원소를 가질 수 있는 공백 히프를 생성
Boolean HeapFull(*heap*, *n*)	::= if (*n* == *max_size*) return TRUE else return FALSE
MaxHeap Insert(*heap*, *item*, *n*)	::= if (!HeapFull(*heap*, *n*)) *item*을 히프에 삽입하고 그 히프를 반환 else return 에러
Boolean HeapEmpty(*heap*, *n*)	::= if (*n* > 0) return TRUE else return FALSE
Element Delete(*heap*, *n*)	::= if (!HeapEmpty(*heap*, *n*)) 히프에서 가장 큰 원소를 제거하고 그 원소를 반환 else return 에러

5.6.2 우선 순위 큐

✓ 운영 체제의 작업 스케줄러

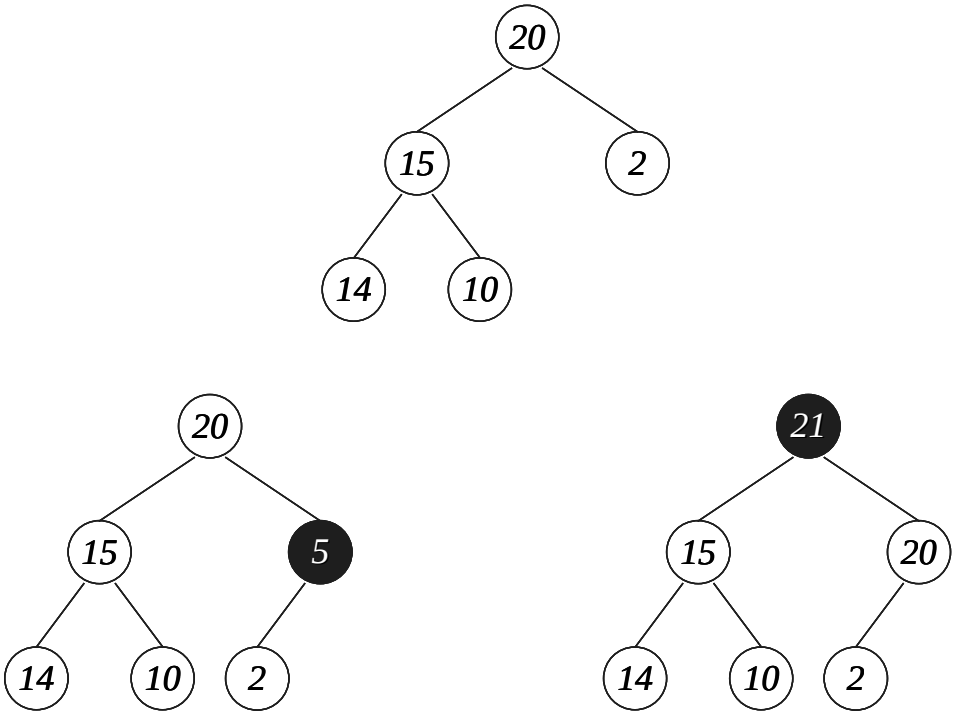
- 관리자에게 가장 높은 우선 순위 부여
  - → 가장 높은 우선 순위를 가진 원소의 삭제
  - → 최대 히프로 구현
- 예상 수행 시간에 기초하여 짧은 작업에 높은 우선 순위 부여
  - → 가장 낮은 우선 순위를 가진 원소의 삭제
  - → 최소 히프로 구현

▶ **페이지 224, 그림 5.27 : 우선 순위 큐의 표현방법**

표현 방법	삽 입	삭 제
순서없는 배열	$O(1)$	$O(n)$
순서없는 연결 리스트	$O(1)$	$O(n)$
정렬된 배열	$O(n)$	$O(1)$
정렬된 연결 리스트	$O(n)$	$O(1)$
최대 히프	$O(\log_2 n)$	$O(\log_2 n)$

5.6.3 최대 히프에서의 삽입

▶ **페이지 225, 그림 5.28 : 최대 히프에서의 삽입**



▶ 페이지 226, 히프의 선언

```
#define MAX_ELEMENTS 200 /* 최대 히프 크기 + 1 */
#define HEAP_FULL(n)  (n == MAX_ELEMENTS – 1)
#define HEAP_EMPTY(n)  (!n)
typedef struct {
    int key;
    /* 다른 필드들 */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```

▶ 페이지 226 ~ 227, 프로그램 5.13 : 최대 히프에 삽입

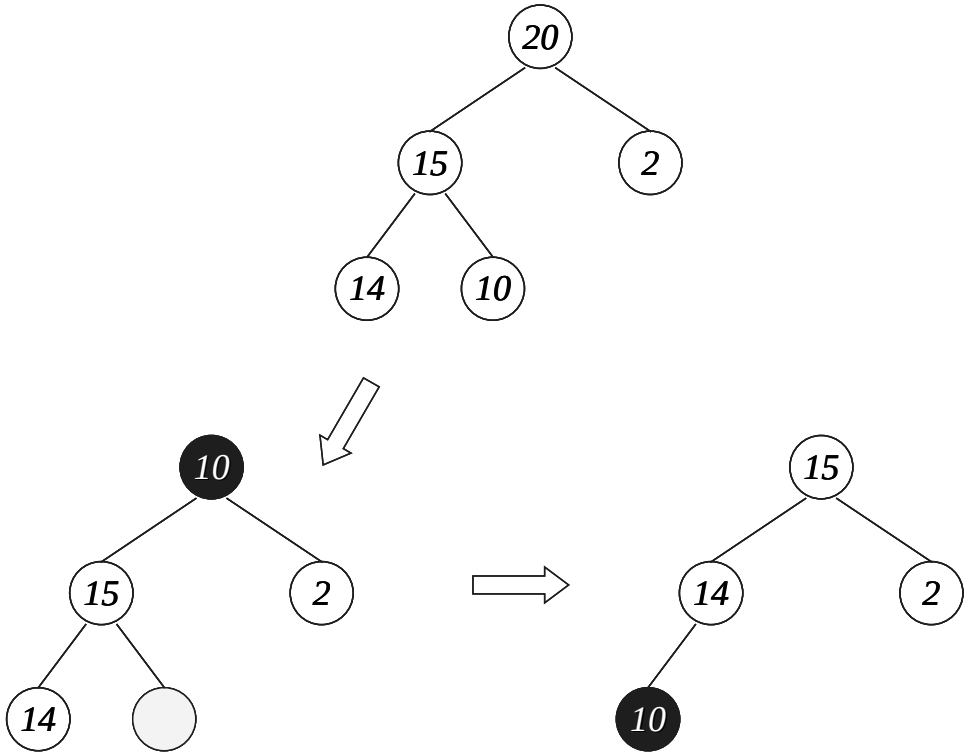
```
void insert_max_heap(element item, int *n)
/* n개의 원소를 갖는 최대 히프에 item을 삽입한다. */
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. \n");
        exit(1);
    }
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```

✍ 분석

- O(log *n*)

5.6.4 최대 히프에서의 삭제

▶ **페이지 227, 그림 5.29 : 최대 히프에서의 삭제**



▶ 페이지 228, 프로그램 5.14 : 최대 히프에서의 삭제

```
element delete_max_heap(int *n)
{   /* 가장 큰 값의 원소를 히프에서 삭제 */
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) { fprintf(stderr, "The heap is empty\n"); exit(1); }
    item = heap[1];        /* 가장 큰 키값을 저장 */
    temp = heap[(*n)--]; /* 히프를 재구성하기위해 마지막 원소를 이용 */
    parent = 1; child = 2;
    while (child <= *n) {
        /* 현 parent의 가장 큰 자식을 탐색 */
        if ((child < *n) && (heap[child].key) < heap[child + 1].key) child++;
        if (temp.key >= heap[child].key) break;
        /* 아래 단계로 이동 */
        heap[parent] = heap[child]; parent = child; child *= 2;
    }
    heap [parent] = temp;
    return item;
}
```

- 분석
  - O(log *n*)

5.7 이진 탐색 트리

5.7.1 소 개

□ 히프

- 임의의 원소를 삭제해야 하는 응용에는 적합하지 않다

→ 이진 탐색 트리 (binary search tree)

정의 : 이진 탐색 트리

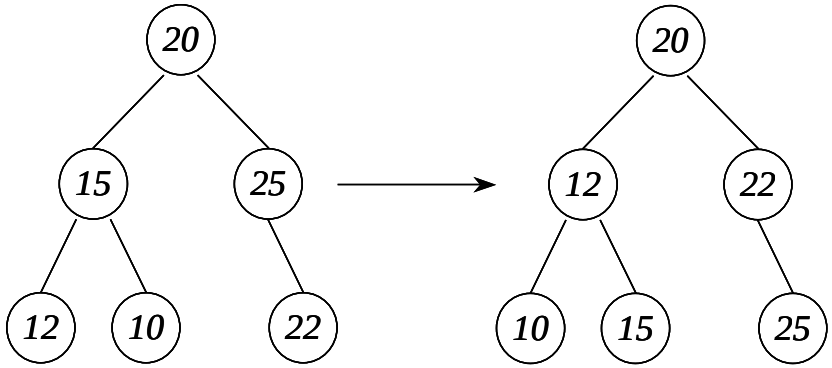
이진 트리로서 공백이 가능하다

만약 공백이 아니라면 다음 성질을 만족한다

- 모든 원소는 키를 가지며, 어떤 두 원소도 동일한 키를 갖지 않는다
- 왼쪽 서브 트리의 키들은 그 서브 트리 루트의 키보다 작다
- 오른쪽서브 트리에 있는 키들은 그 서브 트리 루트의 키보다 크다
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리이다

⇒ 탐색, 삽입, 삭제시 다른 자료 구조보다 좋은 성능을 보인다

▶ **페이지 230, 그림 5.30 : 이진 트리**



□ 이진 탐색 트리의 높이

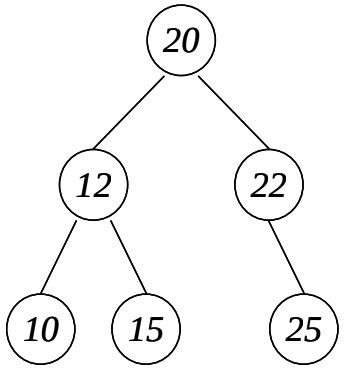
- 평균적으로 O(log *n*)

⇒ 균형 탐색 트리 (balanced search trees)

- 최악의 경우 높이가 O(log *n*)

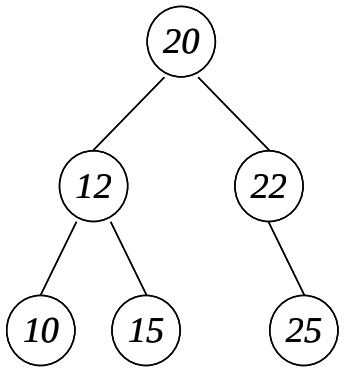
5.7.2 이진 탐색 트리의 탐색

▶ **페이지 231, 프로그램 5.15 : 이진 탐색 트리의 순환적 탐색**




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

▶ 페이지 231, 프로그램 5.16 : 이진 탐색 트리의 반복적 탐색

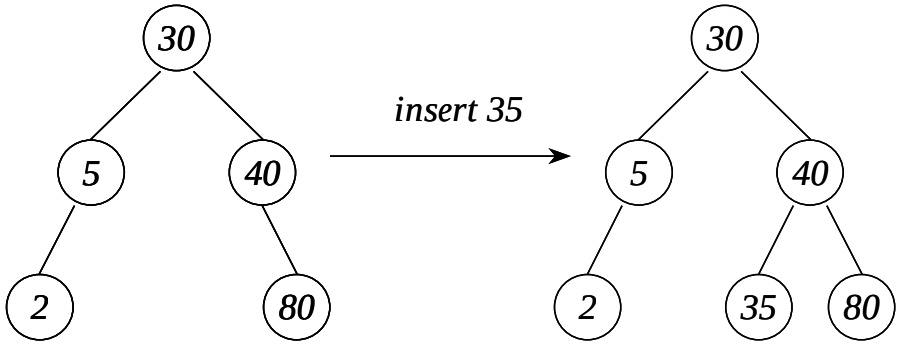



```
tree_pointer search2 (tree_pointer tree,  int key)
/* 키값이 key인 노드에 대한 포인터 반환. 그런 노드가 없는 경우는 NULL을 반환 */
{
    while (tree) {
        if (key == tree->data) return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
    return NULL;
}
```

✍ 분석 = O(h) = O(log n)

5.7.3 이진 탐색 트리에 대한 삽입

▶ **페이지 232, 그림 5.31 : 이진 탐색 트리에 삽입**



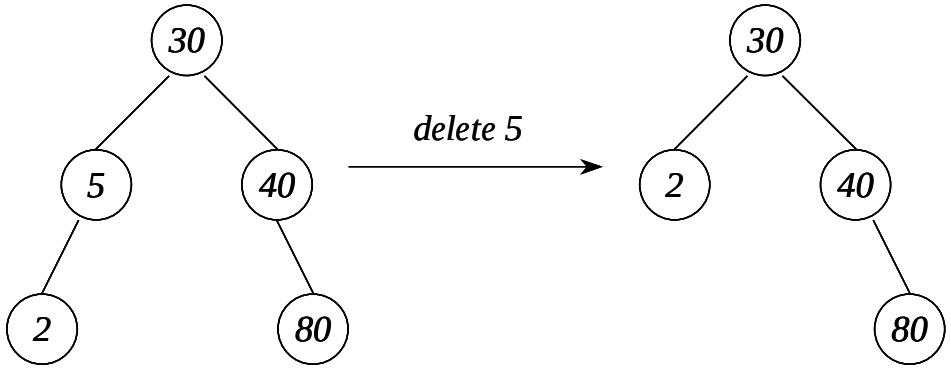
▶ 페이지 233, 프로그램 5.17 : 이진 탐색 트리에 원소를 삽입

```
void insert_node(tree_pointer *node,  int num)
/* 트리내의 노드가 num을 가리키고 있으면 아무 일도 하지 않음; 그렇지 않은 경우는
   data=num인 새 노드를 첨가 */
{
     tree_pointer ptr, temp = modified_search(*node, num);
     if (temp || !(*node)) {     /* num이 트리내에 없음 */
          ptr =  (tree_pointer)malloc(sizeof(node)) ;
          if (IS_FULL(ptr)) {
               fprintf(stderr, "The memory is full\n"); exit(1);
          }
          ptr->data = num; ptr->left_child = ptr->right_child = NULL;
          if (*node)   /* temp의 자식으로 삽입 */
          if (num < temp->data)  temp->left_child = ptr;
          else temp->right_child = ptr;
          else *node = ptr:
     }
}
```

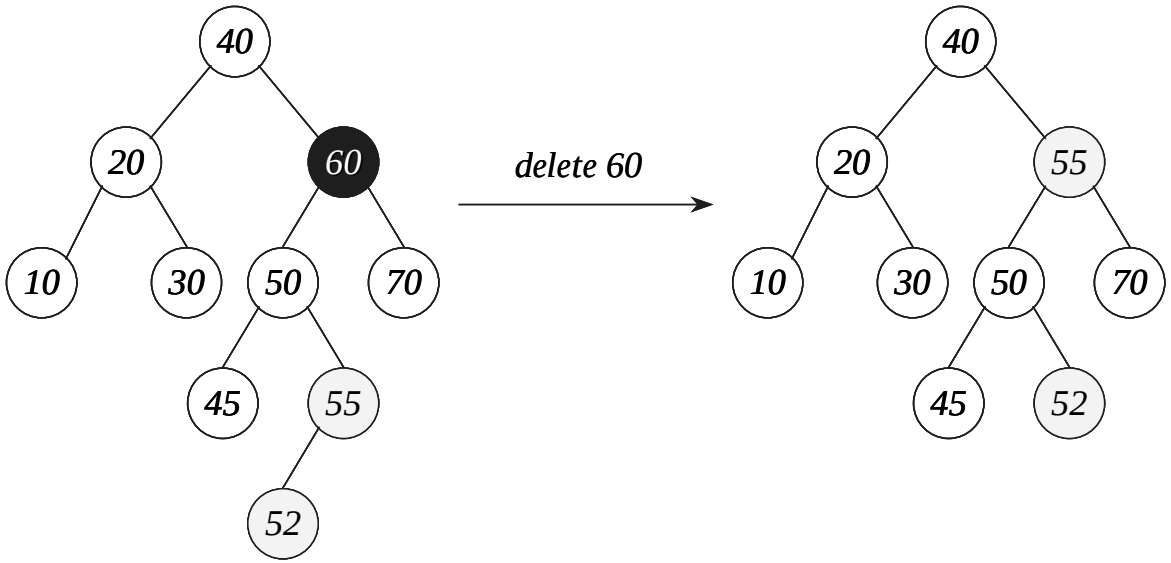
✍ 분석 = O(*h*) = O(log *n*)

5.7.4 이진 탐색 트리에 대한 삭제

▶ **페이지 234, 그림 5.32 : 이진 탐색 트리에 삭제**



▶ **페이지 234, 그림 5.33 : 두 자식을 가진 노드의 삭제**



- 분석 = O(h) = O(log n)

5.7.5 이진 탐색 트리의 높이

- n개의 원소를 갖는 이진 탐색 트리의 높이
  - 평균 : $O(\log_2 n)$
  - 최악의 경우 : $O(n)$
    - 키 값 1, 2, 3, …, n을 순서대로 삽입하는 경우

- 균형 탐색 트리(balanced search tree)
  - 최악의 경우에도 높이가 $O(\log_2 n)$인 탐색 트리
  - AVL tree, 2-3 tree, red-black tree, …

5.8 선택 트리

- *k*개의 순서 순차(ordered sequence)를 하나의 순서 순차로 합병하고자 하는 경우
  - 런 (run) : key 필드에 따라 비감소 순서로 정렬된 몇 개의 레코드로 구성된 순차

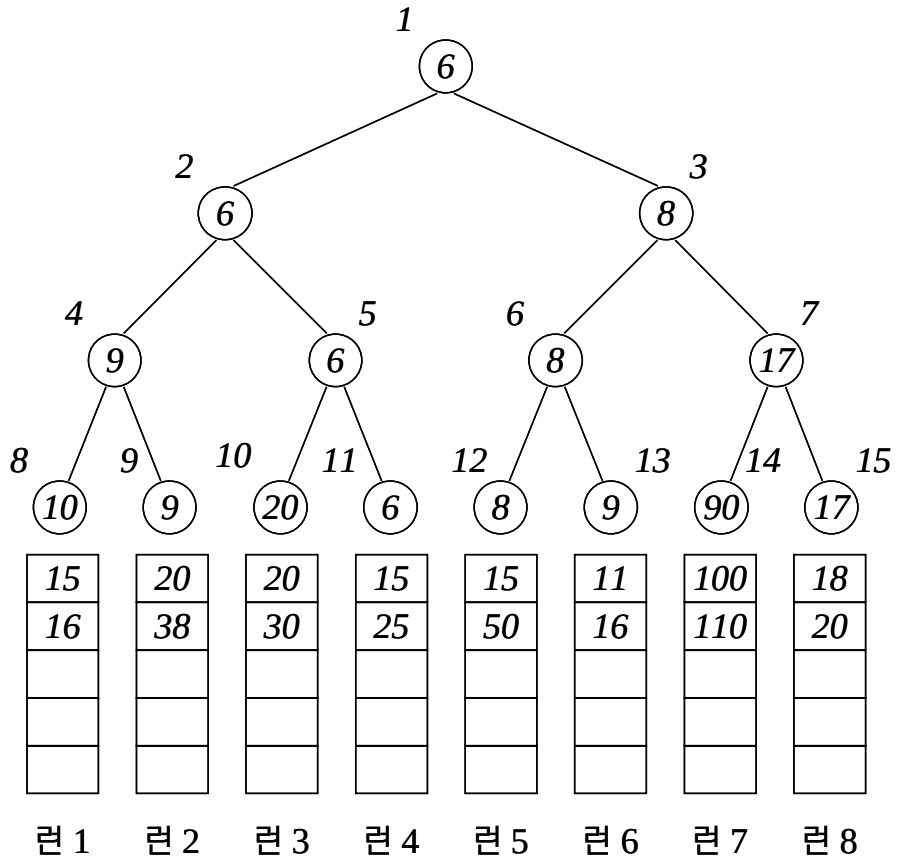
- 선택 트리 (selection tree)
  - 각 노드가 두 개의 자식 노드 중 더 작은 노드를 표현하는 이진 트리
    - 루트 노드는 가장 작은 노드
    - 리프 노드는 해당 런의 첫번째 레코드
  - *k*개의 런에서 다음으로 작은 레코드를 발견하는데 필요한 비교 횟수를 줄인다

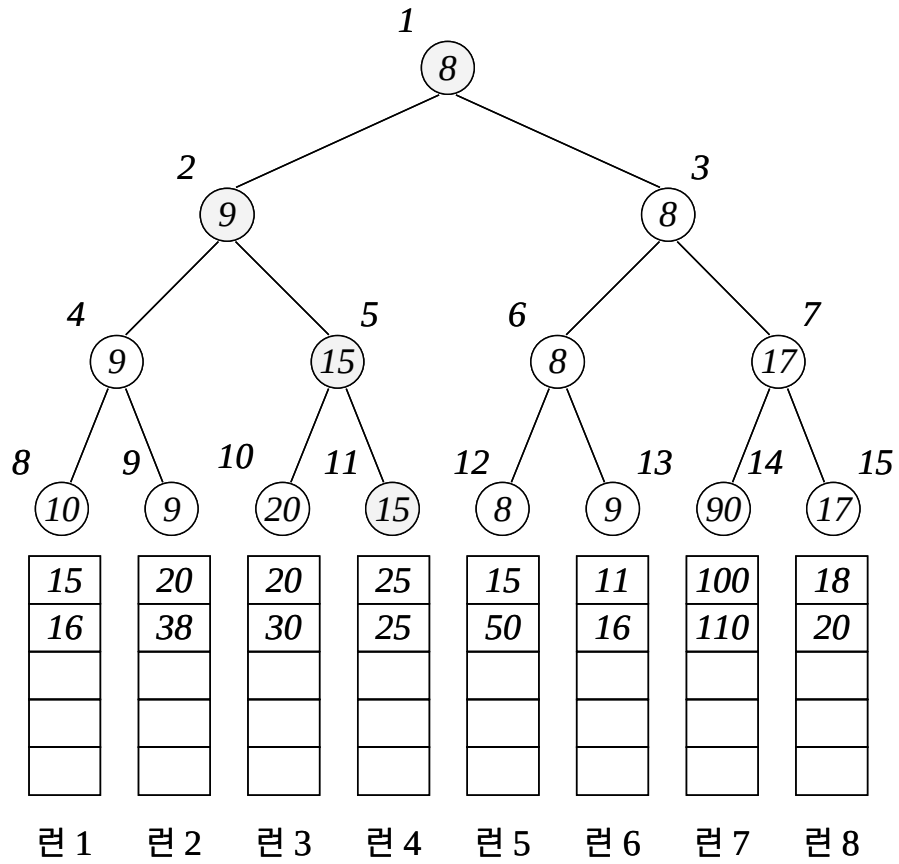
- 승자 트리와 패자 트리
  - 승자 트리
    - 각 노드가 두 개의 자식 노드 중 더 작은 노드를 나타내는 완전 이진 트리
  - 패자 트리
    - 각 단말 노드가 패자에 대한 포인터를 유지

5.8.1 승자 트리

▶ **페이지 236, 그림 5.34 : k = 8인 경우의 선택 트리**

& 페이지 237, 그림 5.35 : 재구성된 재구성된 모습




1
8
2
9
3
8
4
9
5
15
6
8
7
17
8
10
9
9
10
20
11
15
12
8
13
9
14
90
15
17
15
16
20
38
20
30
25
25
15
50
11
16
100
110
18
20
런 1
런 2
런 3
런 4
런 5
런 6
런 7
런 8

□ 비교 횟수를 줄이는 방법으로 선택 트리 기법 사용

- 선택 트리의 구성 : $O(k)$
- 선택 트리의 재구성 : $O(\log_2 k)$
- n개 레코드의 합병에 걸리는 레벨당 시간은 $O(n\log_2 k)$

→ 내부 처리 시간은 $O(n\log_2 k \cdot \log_k m) = O(n\log_2 m)$

→ k와 무관하나 트리 유지시간 때문에 증가

→ 패자 트리로 해결

5.8.2 패자 트리

□ 승자 트리

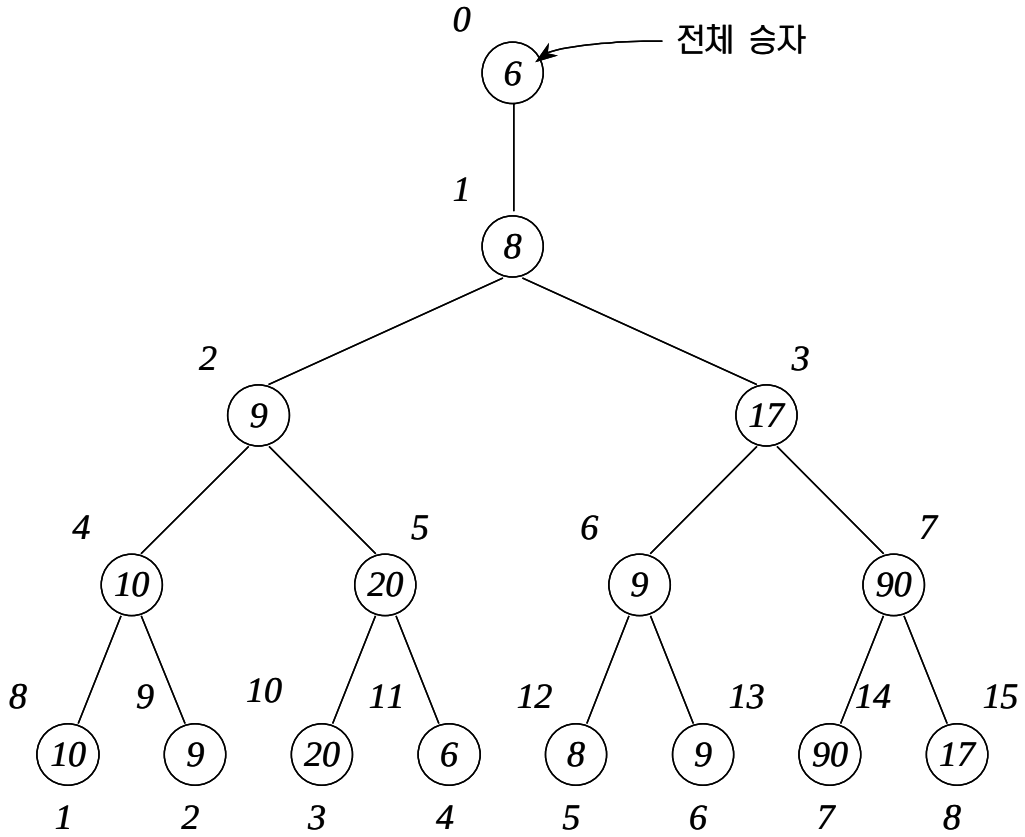
- 토너먼트는 노드 11로부터 루트까지의 경로를 따라 형제 노드들 사이에서 일어난다

→ 형제 노드가 전에 시행된 토너먼트의 패자 레코드에 대한 포인터를 위치시킨다

⇒ 패자 트리

- 각 비단말 노드에 토너먼트의 승자 대신 패자 레코드에 대한 포인터를 위치시킴으로써 재구성 절차를 간소화
- 단말 노드는 각 런의 첫번째 레코드를 표현
- 결과적으로 11부터 1까지 경로상의 형제는 접근되지 않는다

▶ **페이지 238, 그림 5.36 : 패자 트리**

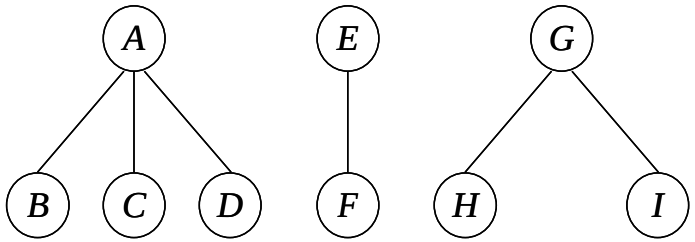


5.9 포리스트

정의

- 포리스트(forest)는 $n \geq 0$개의 분리(disjoint) 트리들의 집합이다

▶ 페이지 240, 그림 5.37 : 세 개의 트리로 구성된 포리스트



5.9.1 포리스트의 이진 트리 변환

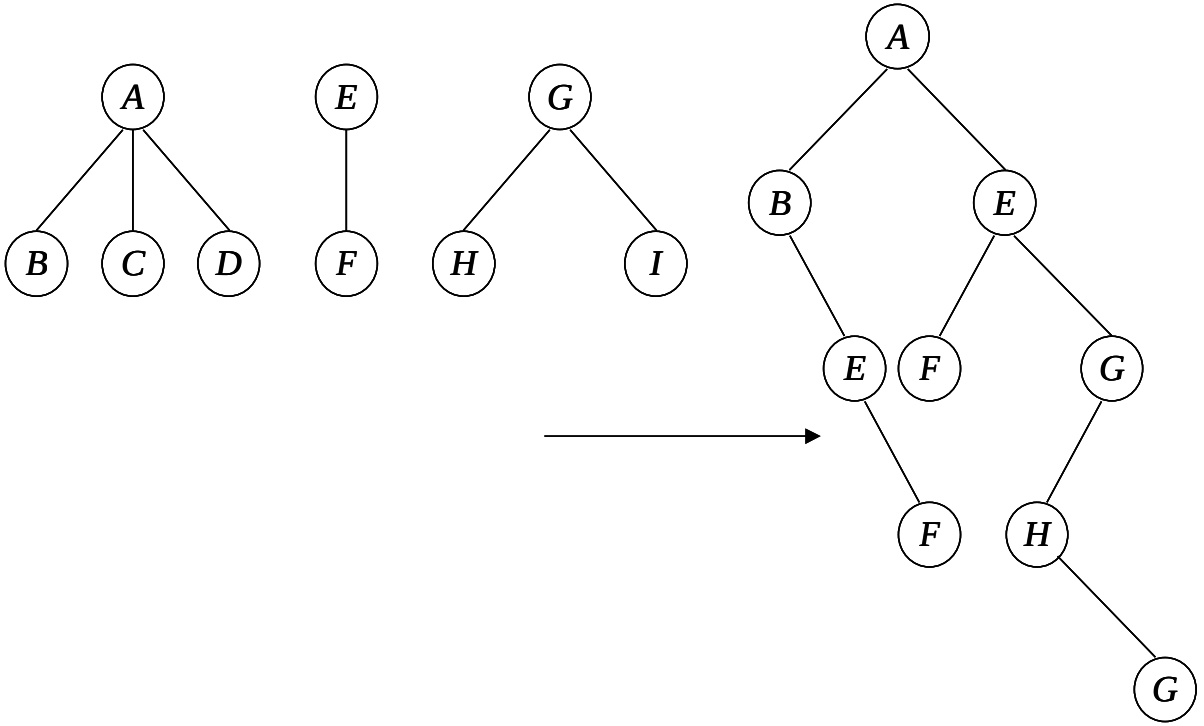
□ 이진 트리로의 변환

- 포리스트의 각 트리를 이진 트리로 변환
- 모든 이진 트리들을 루트의 형제 필드를 통하여 연결

□ 변형 방법의 정의

- T_1, T_2, …, T_n의 트리들로 이루어진 포리스트가 있을 때 이 포리스트에 대응하는 이진 트리는 $B(T_1, T_2, \ldots, T_n)$으로 표시한다
  - $n = 0$이면 B는 공백이다
  - B는 T_1의 루트와 같은 루트를 가지며 왼쪽 서브트리로 $B(T_{11}, T_{12}, \ldots, T_{1m})$을 가진다
  - T_{11}, T_{12}, …, T_{1m}은 T_1의 루트의 서브 트리,
    오른쪽 서브트리는 $B(T_2, T_3, \ldots, T_n)$

▶ **페이지 240, 그림 5.38 : 이진 트리 표현**



5.9.2 포리스트 순회

☐ 전위 순회(preorder traversal)

*T*의 전위 순회는 *F*의 노드들의 트리 전위 순회와 같다

- *F*가 공백이면 귀환
- → *F*의 첫트리의 루트 방문
- → 첫 트리의 서브 트리의 전위 순회
- → *F*의 남은 트리의 전위 순회

☐ 중위 순회(inorder traversal)

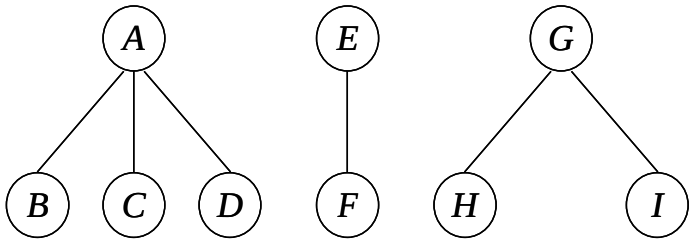
*T*의 중위 순회는 *F*의 노드들의 트리 중위 순회와 같다

- *F*가 공백이면 귀환
- → 첫 트리의 서브 트리의 중위 순회
- → 첫트리의 루트 방문
- → 남은 트리의 전위 순회

☐ 후위 순회 (postorder retrieval)

*T*의 후위 순회는 *F*의 노드들의 트리 후위 순회와 일치하지 않는다

- *F*가 공백이면 귀환
- → 첫 트리의 서브 트리의 후위 순회
- → 남은 트리의 후위 순회
- → 첫 트리의 루트 방문



5.10 집합 표현

□ 가정

- 집합들의 모든 원소는 1, 2, ..., n으로 구성
  - 이 수들은 심볼테이블에서 실제 원소의 이름이 저장된 곳의 인덱스일 수도 있다
- 모든 집합들은 서로 분리 관계(disjoint)를 가진다

□ 연산

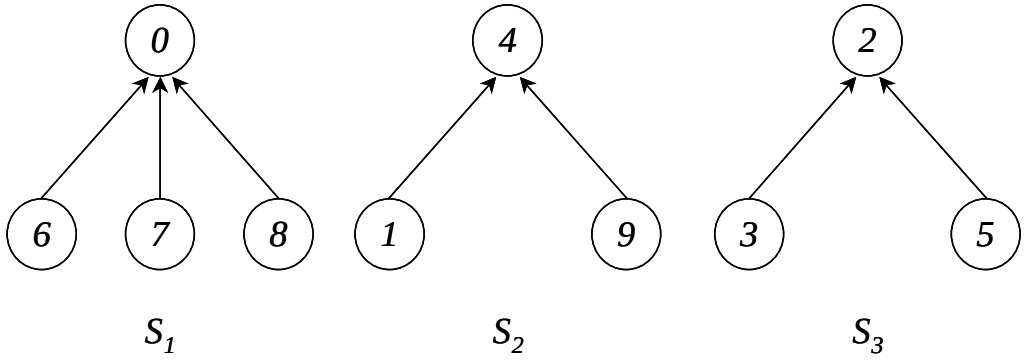
- union : 합집합
- find(*i*) : 원소 *i*를 포함하는 집합을 찾는다

□ 집합의 트리 표현

- ◆ 노드들의 부모 자식 관계 연결 (루트를 제외한 각 노드는 부모에 연결)

✓ S_1 = { 0, 6, 7, 8 }, S_2 = { 1, 4, 9 }, S_3 = { 2, 3, 5 }

✧ **페이지 242, 그림 5.39 : 집합의 포리스트 표현**

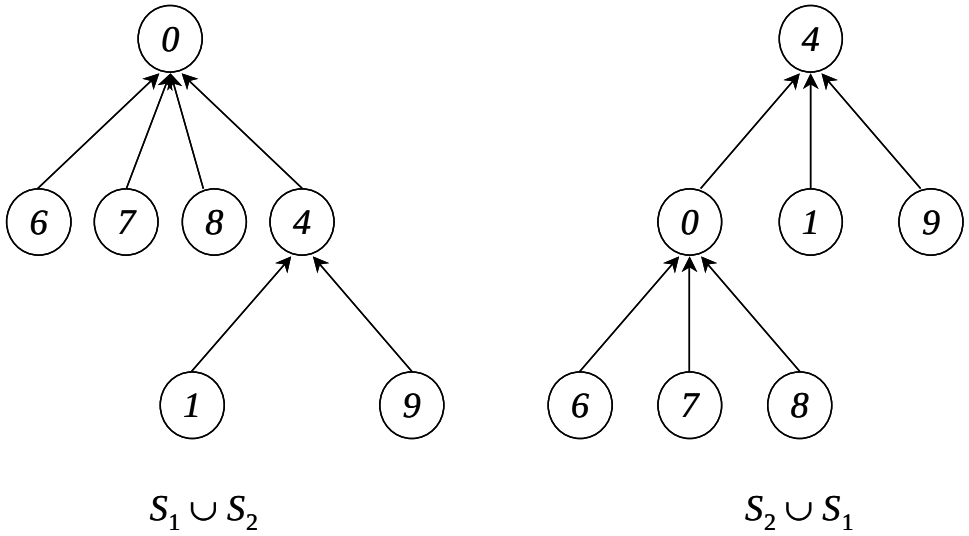


5.10.1 Union과 Find 연산

□ $S_1 \cup S_2$의 표현

- 한 루트의 부모 링크가 다른 트리의 루트를 가리킨다

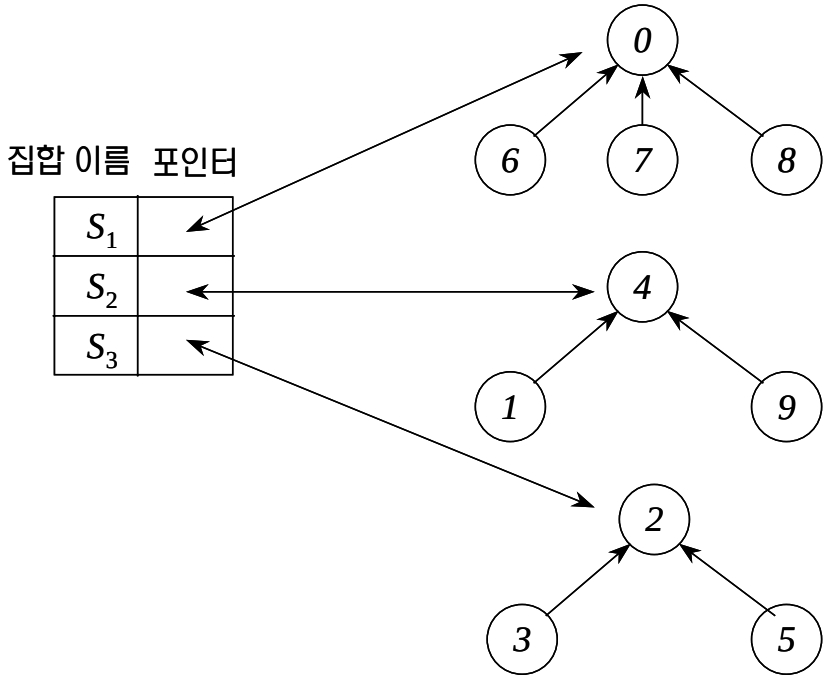
✧ **페이지 243, 그림 5.40 : $S_1 \cup S_2$의 표현**



$S_1 \cup S_2$ $S_2 \cup S_1$

□ 연산을 쉽게 행하기 위한 집합의 표현

✧ **페이지 243, 그림 5.41 : S_1, S_2, S_3의 데이터 표현**



□ 배열 표현

✧ **페이지 244, 그림 5.42 : S_1, S_2, S_3의 배열 표현**

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

✧ **페이지 244, 프로그램 5.18 : union-find 함수의 초기 구현**

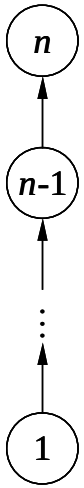
```
int find1(int i)
{
    for(; parent[i] >= 0; i = parent[i]) ;
    return i;
}

void union1(int i, int j)
{
    parent[i] = j; /* or parent[j] = i; */
}
```

– 두 알고리즘은 기술이 용이한 반면 성능은 매우 좋지 않다.

✓ 예

- $S_i = \{ i \}, 1 \le i \le p$
  - 트리들의 처음 상태는 포리스트
  - $parent(i) = 0, 1 \le i \le p$
- 연산 union-find의 수행
  - union(0, 1), find(0)
    union(1, 2), find (0)
    …
    union(n – 2, n – 1), find(0)

✧ **페이지 245, 그림 5.43 : 변질 트리**



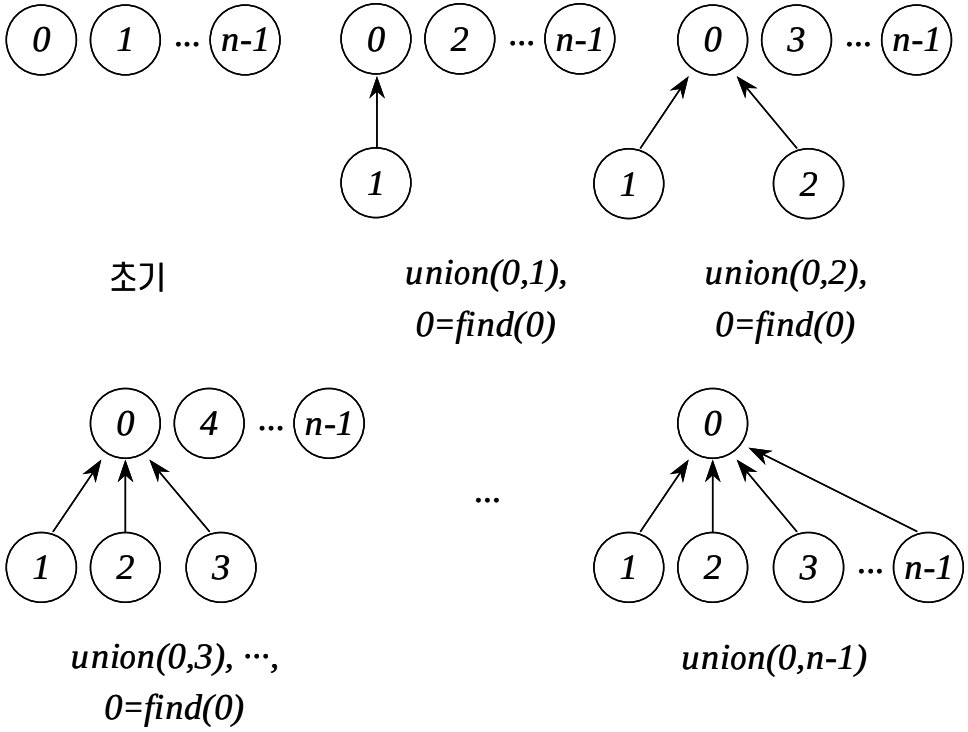
□ 연산 시간

- union
  - 합집합 연산을 위한 소요시간은 상수
  - n – 1개 합집합 연산 수행시간 O(n)
- find
  - i 레벨에 있는 원소에 대한 find 수행시간은 O(i)
  - n – 2번의 find 수행시간은 $O\left(\sum_{1}^{n-2} i\right) = O\left(n^2\right)$

□ 개선된 알고리즘 - 가중 법칙의 사용

◆ *i* 트리에 있는 노드수가 *j* 트리의 노드수보다 적으면 *j*를 *i*의 부모로 아니면 반대로 한다

✧ **페이지 246, 그림 5.44 : 가중법칙을 적용하여 구한 트리**



✧ **페이지 246, 프로그램 5.19 : union 함수**

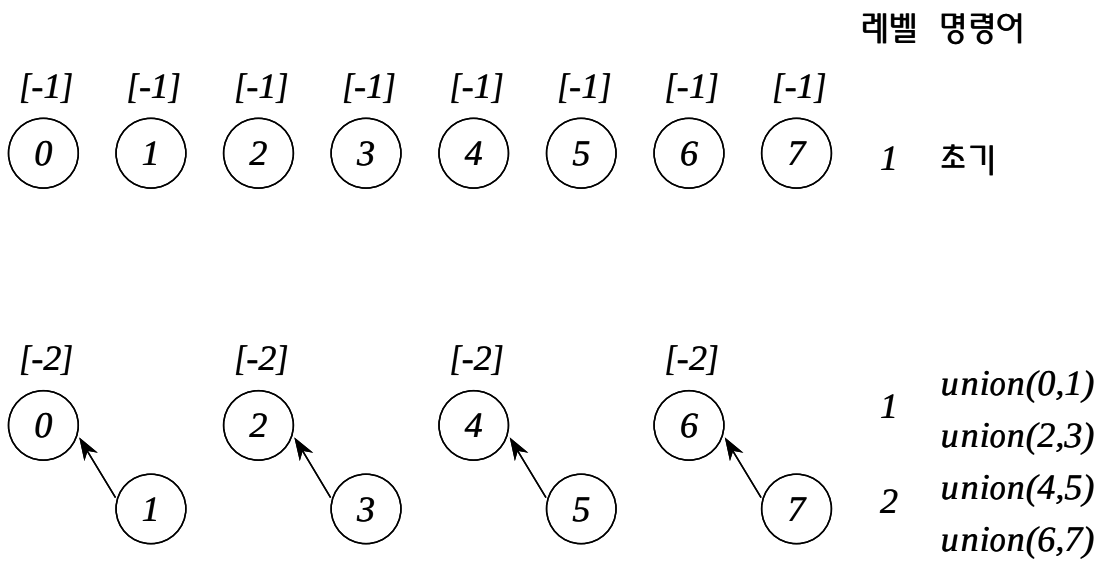
```
void union2(int i, int j)
/* 가중 법칙을 이용하여 루트가 i와 j(i != j)인 집합을 합집합함.
   parent[i] = -count[i]이며 parent[j] = -count[j]  */
{
    int temp;
    temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {
        parent[i] = j; /* j를 새로운 루트로 만듦*/
        parent[j] = temp;
    }
    else {
        parent[j] = i; /* i를 새로운 루트로 만듦 */
        parent[i] = temp;
    }
}
```

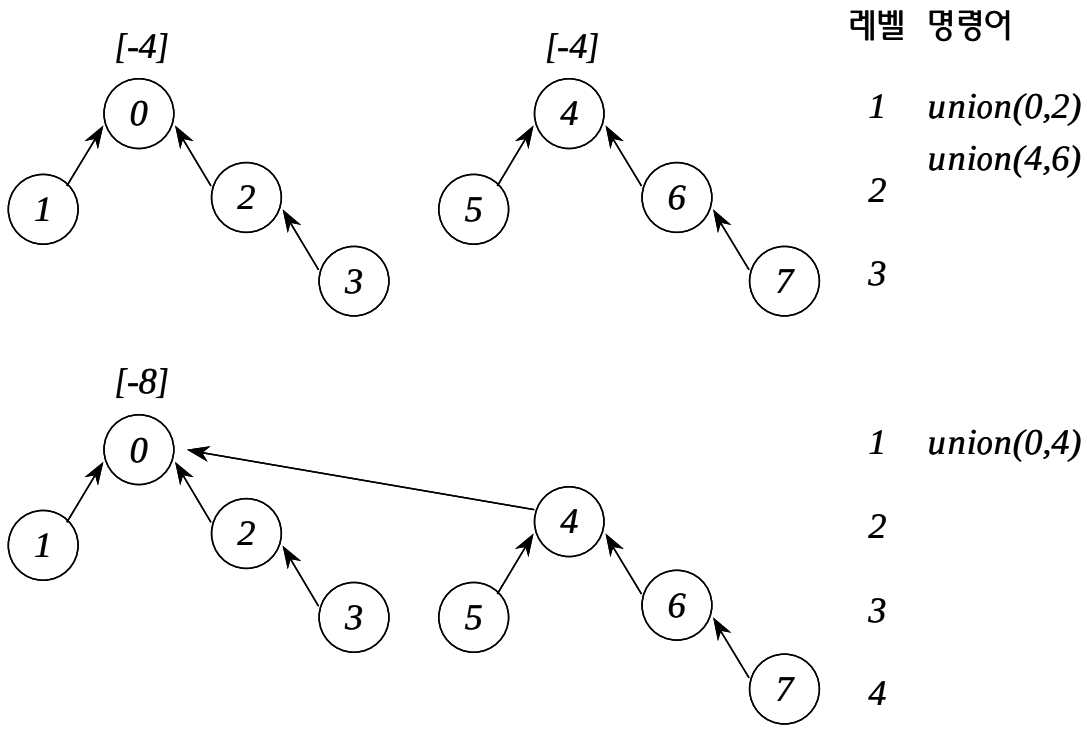
□ n개의 노드를 가진 트리 T가 union2 알고리즘으로 만들어진 트리라면 T에 있는 어떤 노드도 $\lceil \log_2 n \rceil + 1$보다 큰 레벨을 가질 수 없다

✓ 예

- 초기 상태 : parent(i) = -count(i) = -1, $1 \le i \le n = 2^3$인 집합
- union(1, 2), union(3, 4), union(5, 6), union(7, 8), union(1, 3), union(5, 7), union(1, 5)

✧ **페이지 248, 그림 5.45 : 최악의 경우의 트리**





레벨	명령어
1	*union(0,2)*
2	*union(4,6)*
3	
1	*union(0,4)*
2	
3	
4	

- 알고리즘 분석
  - find의 *n*개의 원소를 가진 트리에서 한번 수행시간은 O(log*n*)가 최대
  - *n* – 1개의 union과 *m*개의 find의 수행시간은 최악의 경우 O(*n* + *m*log*n*)

⇒ 붕괴 법칙

- 만일 *j*가 *i*에서 루트로 가는 경로 위에 있다면 *j*를 루트의 자식으로 만든다

✧ **페이지 248, 프로그램 5.20 : find 함수**

```
int find2(int i)
/* 원소 i를 포함하는 루트를 찾음. 붕괴 법칙을 사용하여 i로부터 루트로 가는 모든 노드를
   붕괴시킴 */
{
    int root, trail, lead;
    for (root = i; parent[root] >= 0; root = parent[root]) ;
    for (trail = i; trail != root; trail=lead) {
        lead = parent[trail];
        parent[trail] = root;
    }
    return root;
}
```

- 개별적 find 수행시는 2배의 시간의 필요하지만 연속적인 수행시 최악의 경우를 피할 수 있다

✓ *find*(7), *find*(7), ⋯, *find*(7) : 총 8개의 find 수행

- find1 함수 : $8 \times 3 = 24$ 회의 이동 필요
- find2 함수 : $3 + 3 + 1 \times 7 = 13$ 회의 이동

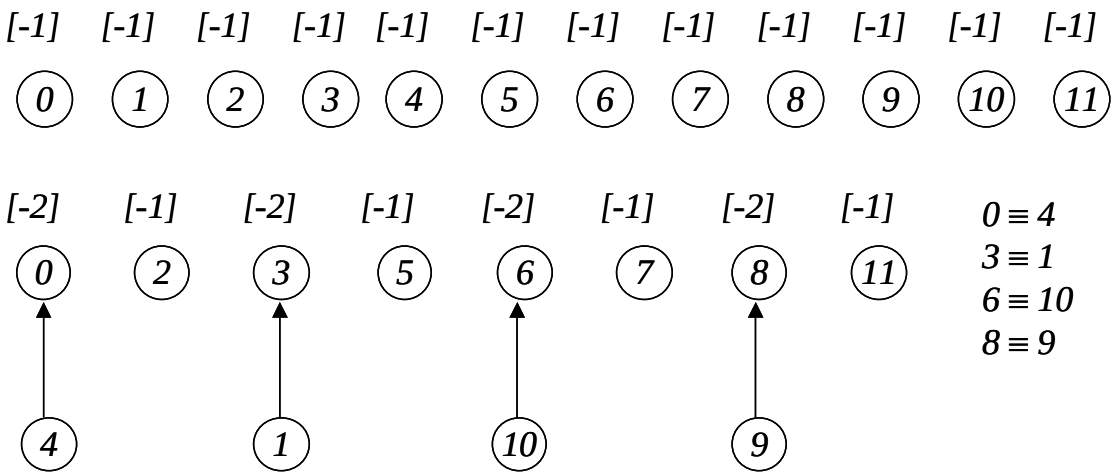
5.10.2 동치 부류

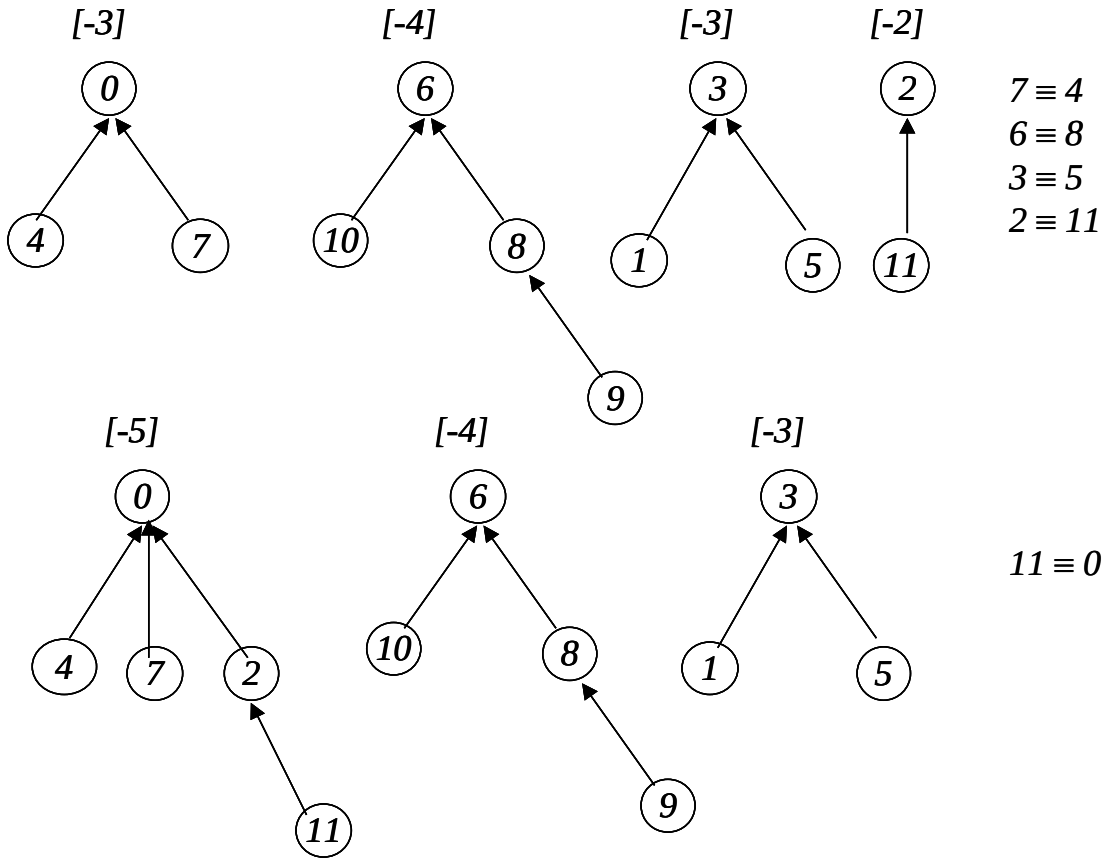
□ union과 find를 이용한 동치 부류 처리

- 하나의 동치 부류 처리에 두개의 find와 하나의 union이 필요

✧ **페이지 251, 그림 5.46 : 동치 부류를 위한 트리의 예**

초기 상태



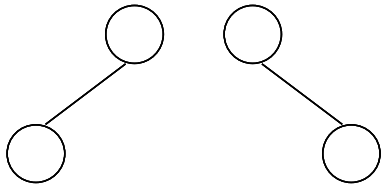

[-3]
[-4]
[-3]
[-2]
0
6
3
2
4
7
10
8
1
5
11
9
7 ≡ 4
6 ≡ 8
3 ≡ 5
2 ≡ 11
[-5]
[-4]
[-3]
0
6
3
4
7
2
10
8
1
5
11
9
11 ≡ 0

5.11 이진 트리의 개수 계산

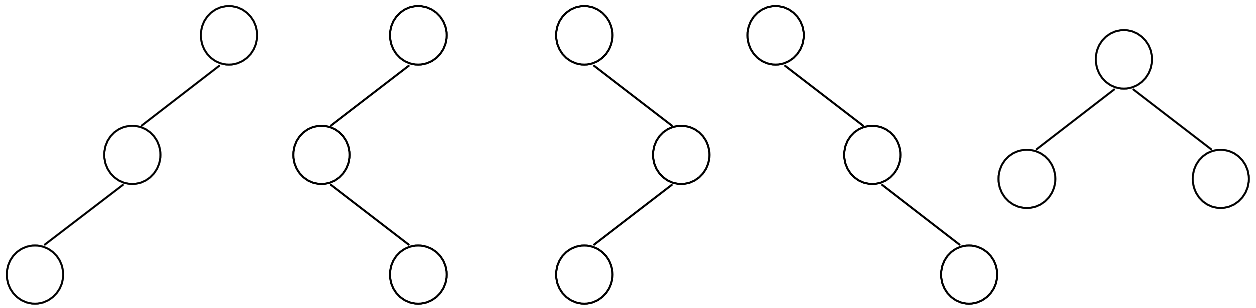
□ 상이한 이진 트리

✧ **페이지 252, 그림 5.47 : *n* = 2일 때의 서로 다른 이진 트리**

& 페이지 252, 그림 5.48 : *n* = 3일 때의 서로 다른 이진 트리



$n = 2$



$n = 3$

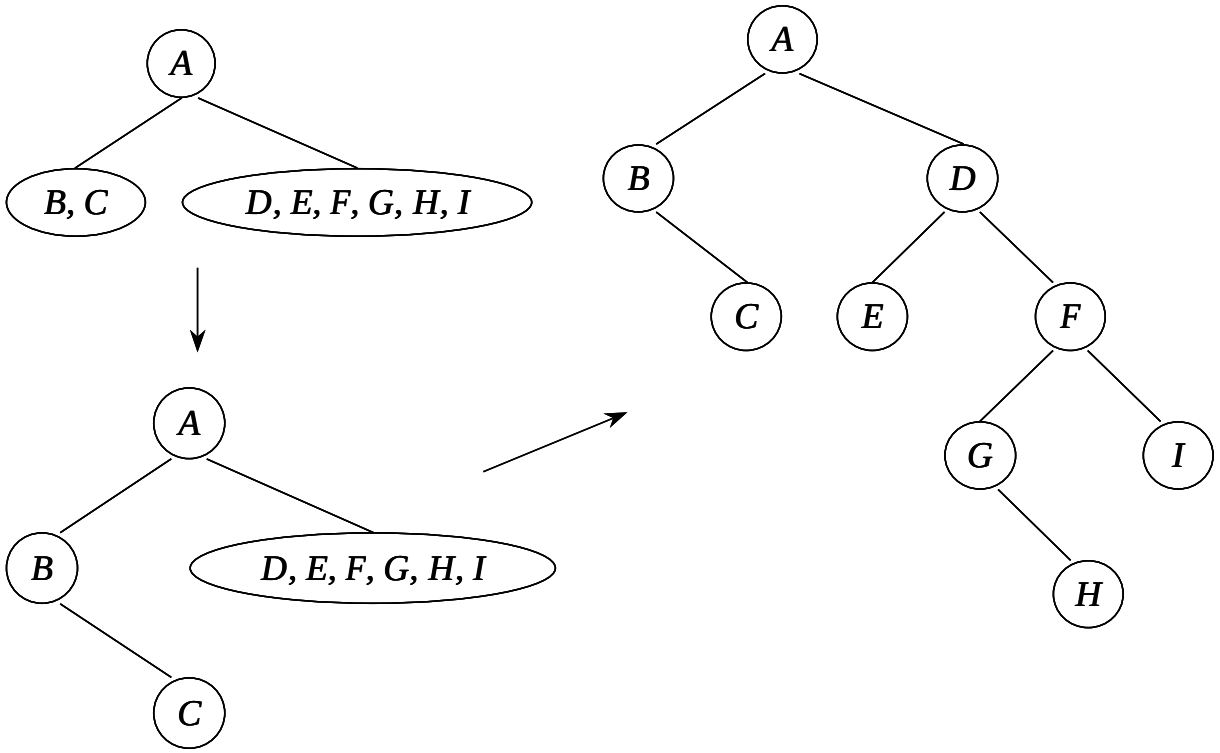
□ 스택 순열

- ◆ 모든 이진 트리는 유일한 전위-중위 순서 쌍을 가진다

✓ 예

- 전위 순서 : *A B C D E F G H I*
- 중위 순서 : *B C A E D G H F I*

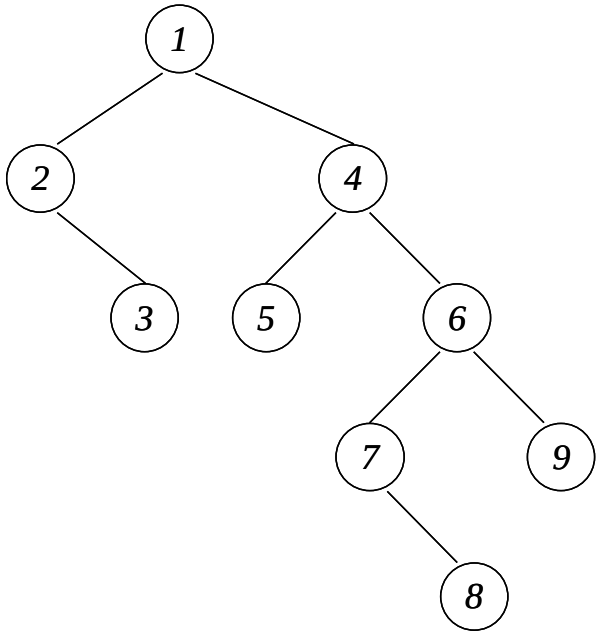
✧ **페이지 253, 그림 5.49 : 중위와 순열로부터 구성한 이진 트리**



□ 이진 트리의 갯수

- 이진 트리의 전위 순열이 1, 2, ..., 9일때 중위 순열은 2, 3, 1, 5, 4, 7, 8, 6, 9 이다.

✧ **페이지 253, 그림 5.50 : 노드에 번호를 부여한 이진 트리**



□ 서로 다른 이진 트리의 수는 $1, 2, \ldots, n$의 전위 순열을 가지는 이진트리로부터 얻을 수 있는 중위 순열의 수와 같다.

- 중위 순열 : 1부터 n까지의 수를 스택에 삽입, 삭제하는 모든 방법으로 만들 수 있는 이진 트리의 갯수와 같다.
  - n = 3 : (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 2, 1)

✧ **페이지 254, 그림 5.51 : 다섯 개의 순열에 대응하는 이진 트리**