5장 트리

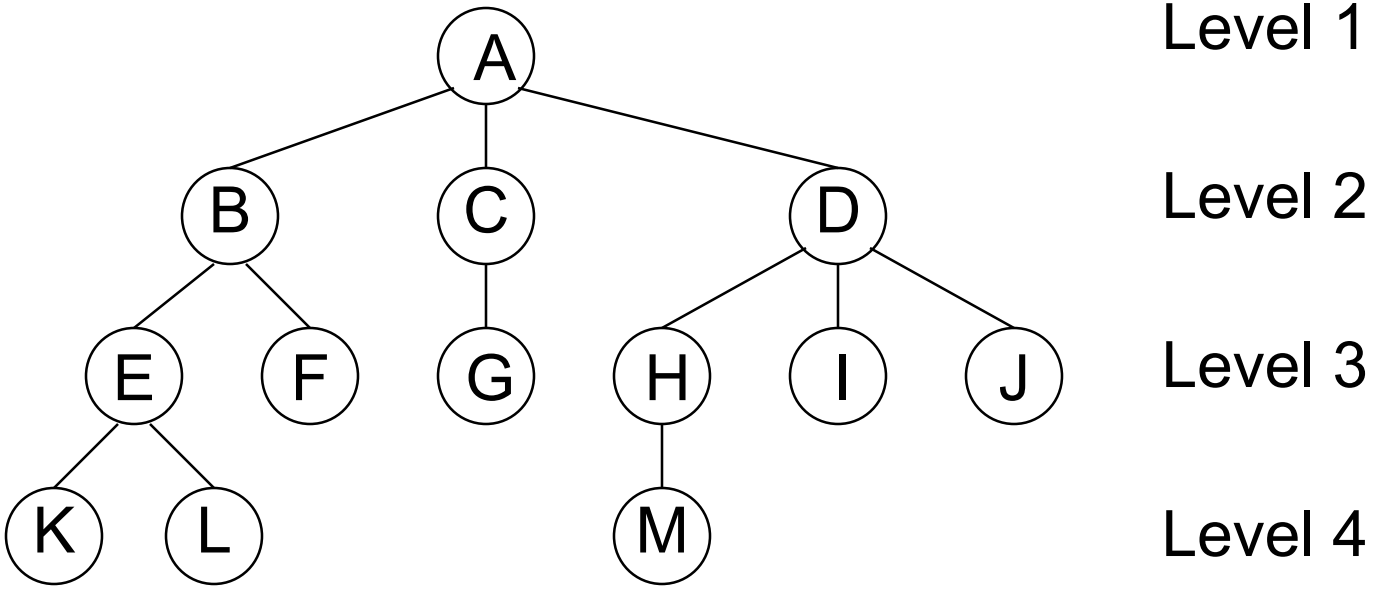
- 트리 (tree)
  - 정보의 항목들이 가지(branch)로 연결될 수 있게 데이터가 조직되는 것, 예) 그림 5.1

- 정의: 트리는 1개 이상의 노드로 이루어진 유한집합으로서

  ① root node

  ②나머지 노드들은 n(≥0)개의 분리 집합(disjoint set) T_1, T_2, … , T_n으로 분리, T_i 는 각각 트리로서 subtree

  - 노드: 정보항목 + 다른 노드로 뻗어진 가지



- 트리 관련 용어
  - 노드의 차수 (A: 3), 트리의 차수 (3)
  - Leaf 또는 단말노드 (K, L, F, G, M, I, J)
  - Parent (E: B), Children (B: E & F), Siblings (E & F)
  - Ancestor (M: H, D, A), Descendants (B: E, F, K, L)
  - Level (Root: 1), Height or Depth (4)

5.2 이진 트리

- 정의: 이진 트리는 공집합이거나 루트와 왼쪽 서브트리, 오른쪽 서브트리라고 부르는 분리된 이진 트리로 구성된 노드의 유한집합이다.

- 이진 트리에 대한 ADT

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

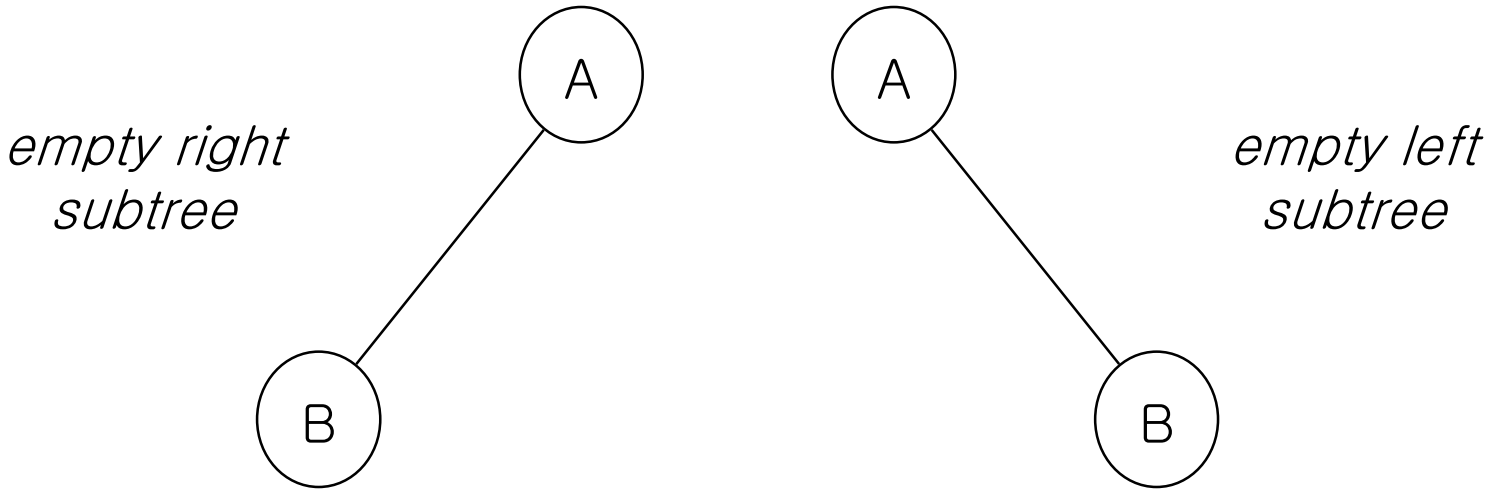
element Data(bt) ::= if (IsEmpty(bt)) return 에러
else bt의 루트에 있는 데이타를 반환

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

이진 트리와 트리의 차이점

- 공백 이진 트리 존재
- 서브트리의 순서 중요

- 서로 다른 두 이진 트리



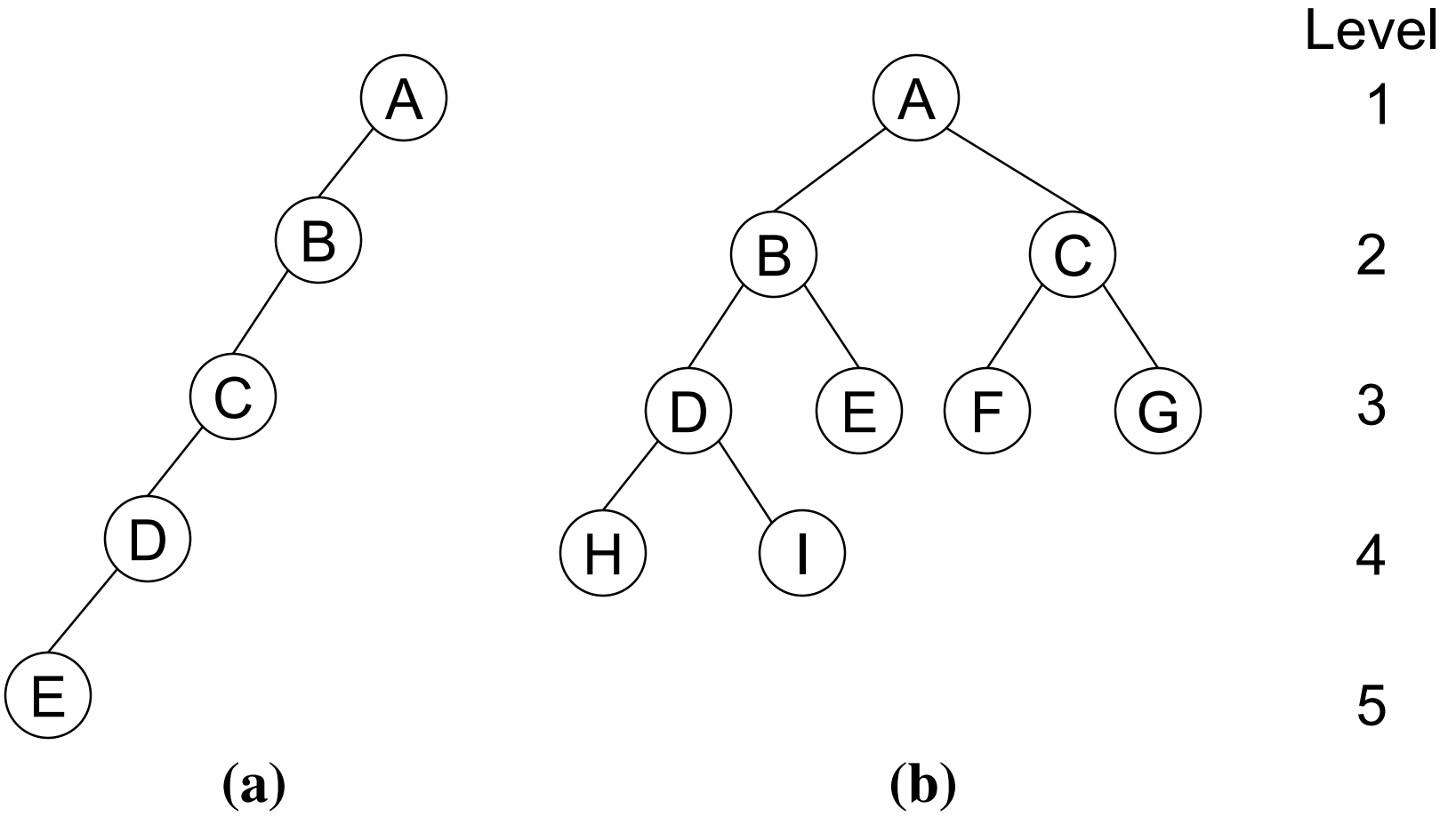


Figure 5.9: 경사 트리와 완전 이진 트리

이진 트리의 특성

- 보조정리 5.1 [최대 노드 수]
  - (1) 레벨 i에서의 최대 노드수 : $2^{i-1}(i \geq 1)$
  - (2) 깊이가 k인 이진 트리가 가질 수 있는 최대 노드 수 = 레벨 1,2,..k의 노드의 합 = $2^k - 1(k \geq 1)$

- 보조정리 5.2[단말노드수와 차수가 2인 노드수와의 상관관계]
  - n0 : 단말 노드수, n2 : 차수가 2인 노드 수 일 때
  - $n0 = n2 + 1$

pf) n : 전체 노드 수, B :전체 간선 수(n_1+2n_2)

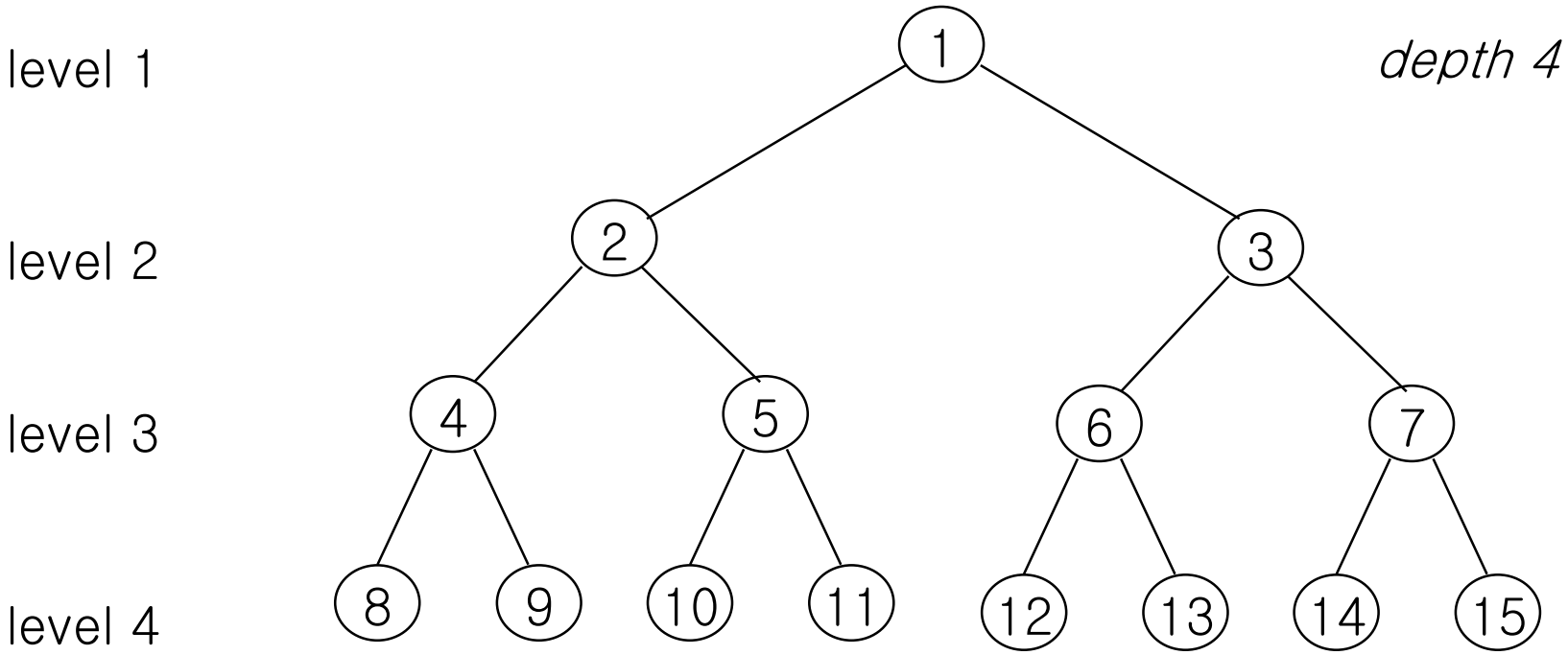
n_1: 차수가 1인 노드 수

$n=n_0+ n_1+ n_2$

$n=B+1=n_1 + 2n_2 +1$

$n_0 = n_2 +1$

- 포화 이진 트리(full binary tree)
  - 깊이가 K이고, 노드수가 2^K-1 인 이진 트리
  - 1,...,2^K-1 까지 순차적 노드 번호를 붙인 깊이 4의 포화 이진 트리



- 정의: 깊이가 K이고 노드수가 n인 이진트리가 만일 이 트리의 각 노드들이 깊이 k인 포화 이진 트리에서 1부터 n까지 번호를 붙인 노드들과 일대일로 일치할 때 완전 이진 트리라 함, 예) 그림 5.9의 (b)

- 이진 트리의 표현
  - 배열 표현, 링크 표현

- 이진트리의 배열표현
  - 그림 5.10처럼, 노드에 1부터 n까지 번호가 할당되므로, 일차원 배열에 각 노드를 저장
  - 보조정리 5.3을 사용하여 자식과 부모의 위치 결정

- 보조정리 5.3: n개의 노드를 가진 완전 이진트리

  ① PARENT(i) : $\lfloor i/2 \rfloor$ if $i \neq 1$

  ② LCHILD(i) : 2i if $2i \leq n$

  otherwise no left child

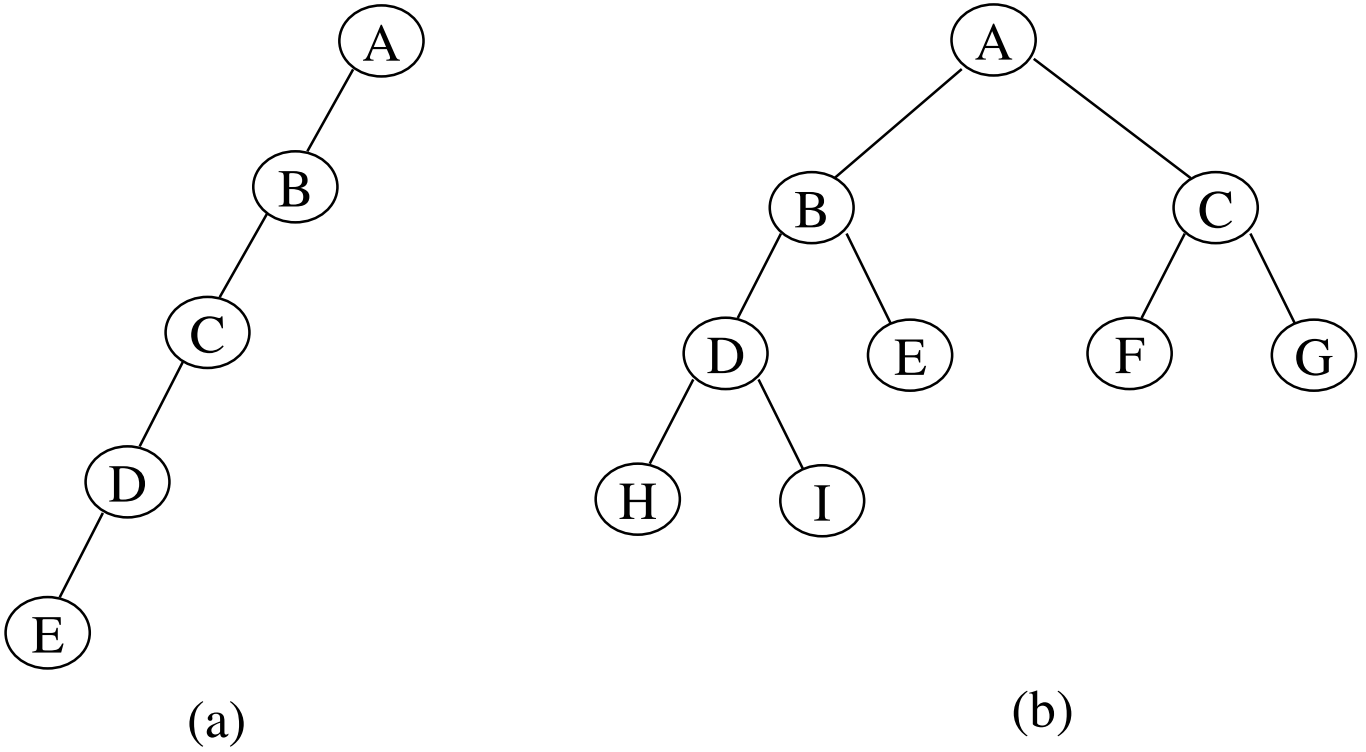
  ③ RCHILD(i) : 2i+1 if $2i+1 \leq n$

  otherwise no right child

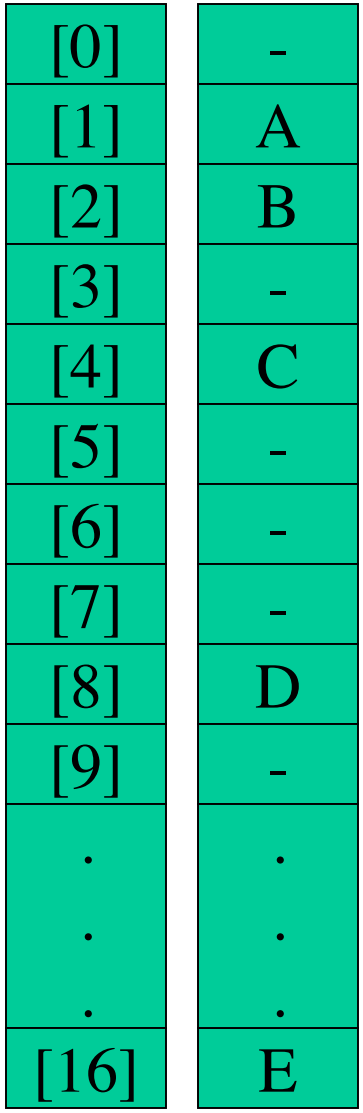
이진 트리의 배열 표현

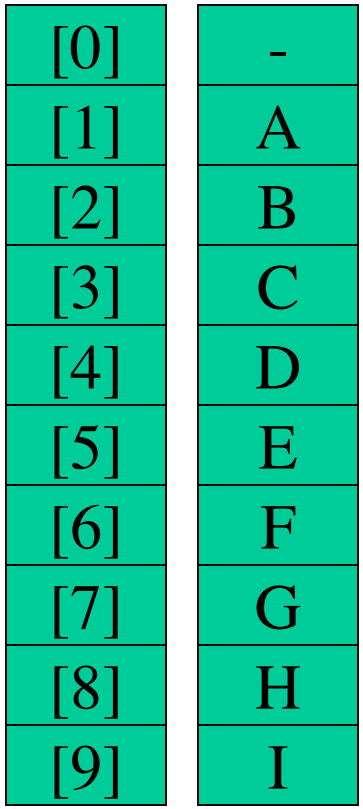
① 완전 이진 트리는 배열 표현이 공간 낭비 없으므로 좋은 표현

② 편향 이진 트리(깊이 k) : 2^k-1중 k만 사용 (최악의 경우)



(a) (b)

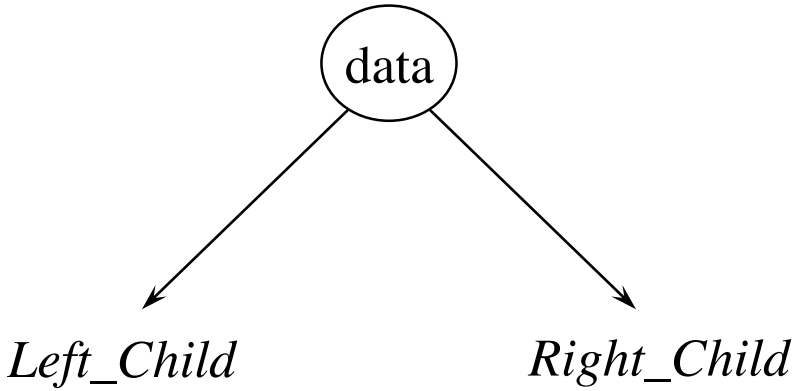

[0]
-
[1]
A
[2]
B
[3]
-
[4]
C
[5]
-
[6]
-
[7]
-
[8]
D
[9]
-
.
.
.
.
.
.
[16]
E


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

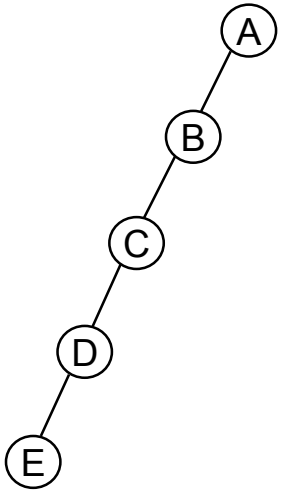
링크 표현

- 배열표현은 complete binary tree가 아닐 때는 메모리낭비가 있고, 삽입/삭제 시 많은 노드이동 필요

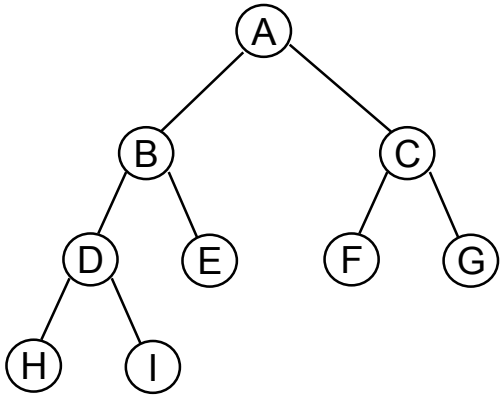
- linked 표현 사용



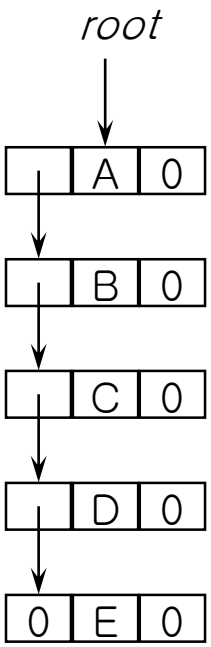

```
typedef struct node *tree_pointer;
typedef struct node {
            int data;
            tree_pointer left_child, right_child;
            } ;
```

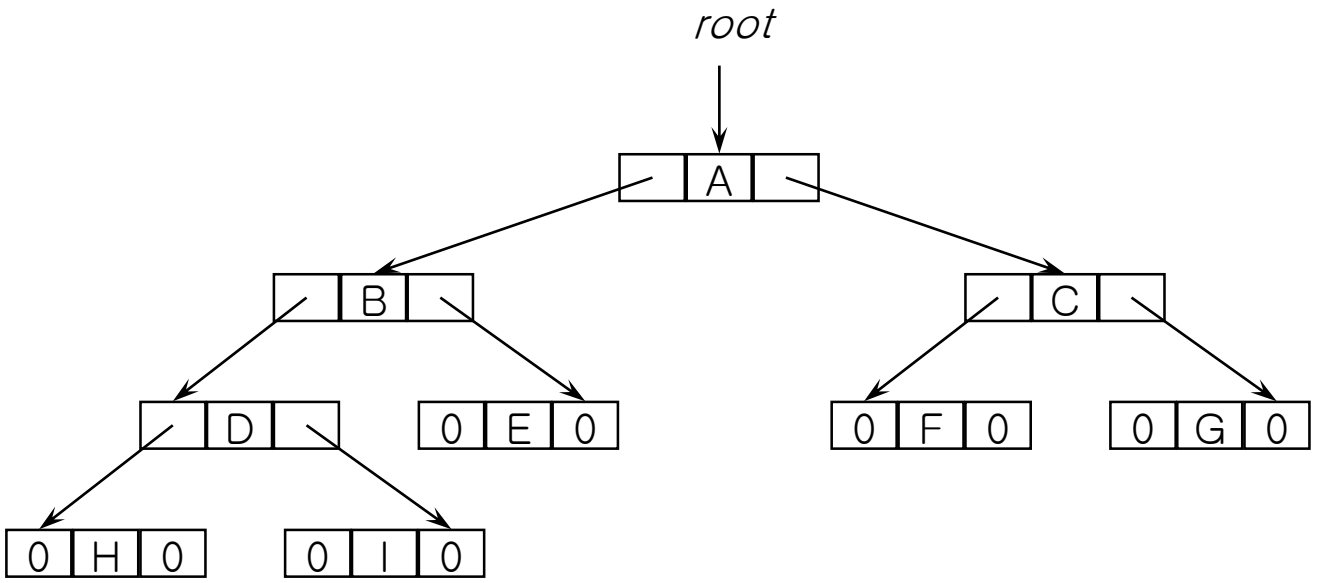

A
B
C
D
E

(a)


A
B
C
D
E
F
G
H
I

(b)


root
A 0
B 0
C 0
D 0
0 E 0

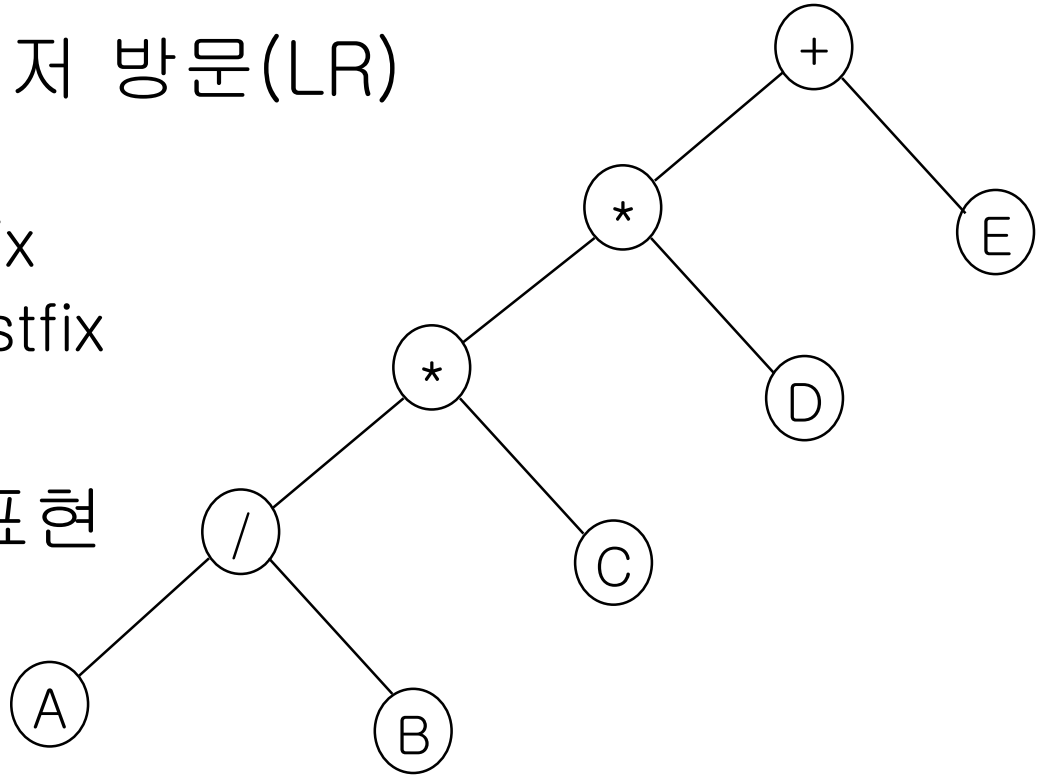

root
A
B
C
D
0 E 0
0 F 0
0 G 0
0 H 0
0 I 0

5.3 이진 트리 순회

- 트리의 노드들을 방문하는 연산
- 순회 방법 : LVR, LRV, VLR, VRL, RVL, RLV
  - L: 왼쪽 이동, V: 노드방문(데이타 출력), R: 오른쪽 이동

- 왼쪽을 오른쪽보다 먼저 방문(LR)
  - LVR (inorder) : infix
  - VLR (preorder) : prefix
  - LRV (postorder) : postfix

- 산술식의 이진 트리 표현
  - operators, variables

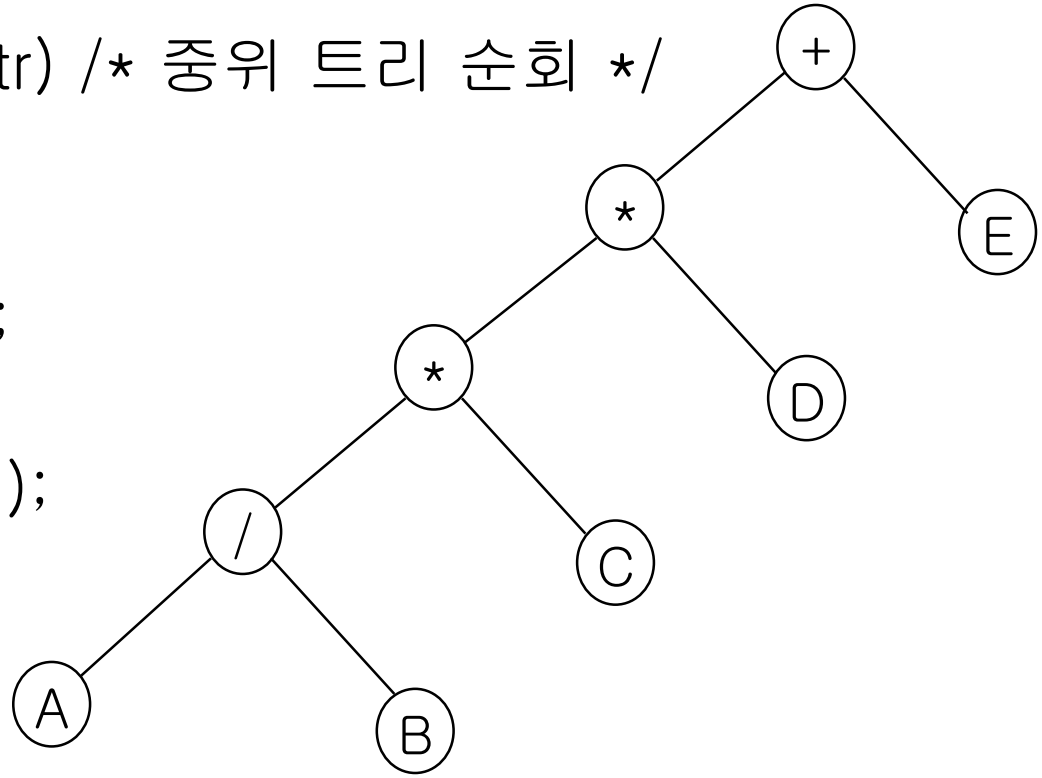


중위 순회(inorder traversal)

1. 널 노드에 도달할 때까지 왼쪽 방향으로 이동
2. 널 노드에 도착하면 널 노드의 부모 방문
3. 순회는 오른쪽 방향으로 계속
4. 오른쪽으로 이동할 수 없을 때에는 하나 더 뒤의 노드로 이동

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

output A / B * C * D + E

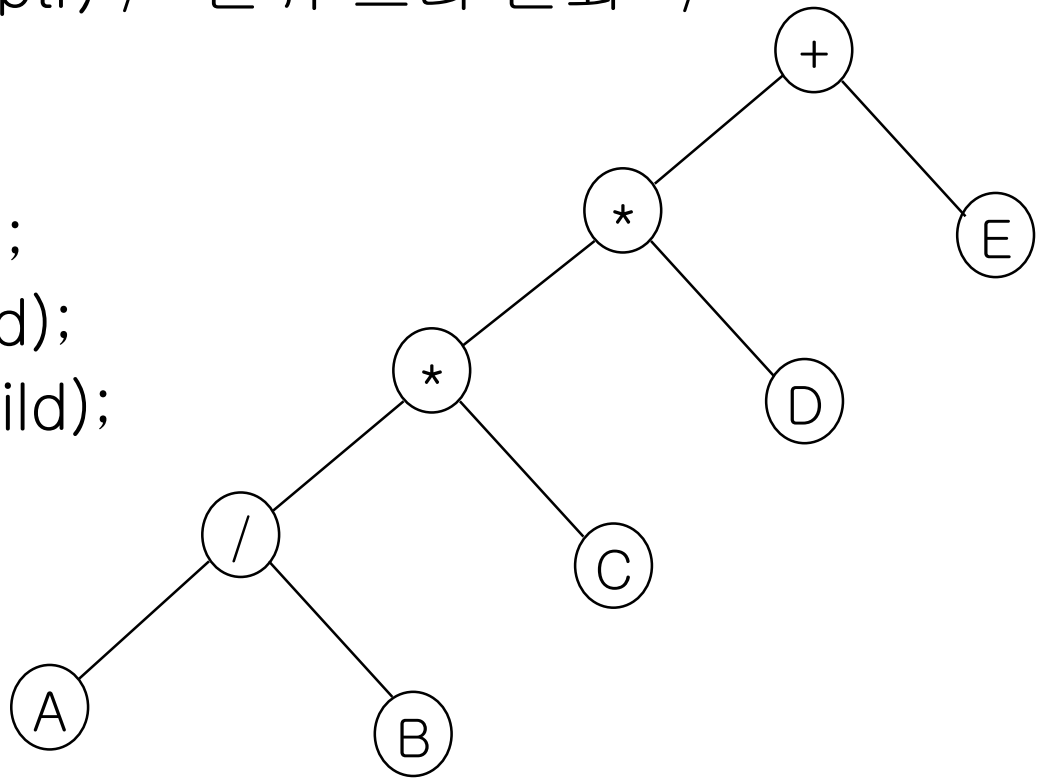


전위 순회(preorder traversal)

1. 노드를 먼저 방문
2. 왼쪽 가지의 모든 노드 방문
3. 널 노드에 도달하면 오른쪽 자식을 가진 가장 가까운 조상으로 돌아가서 오른쪽 자식에서 순회를 계속

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

output + * * / A B C D E

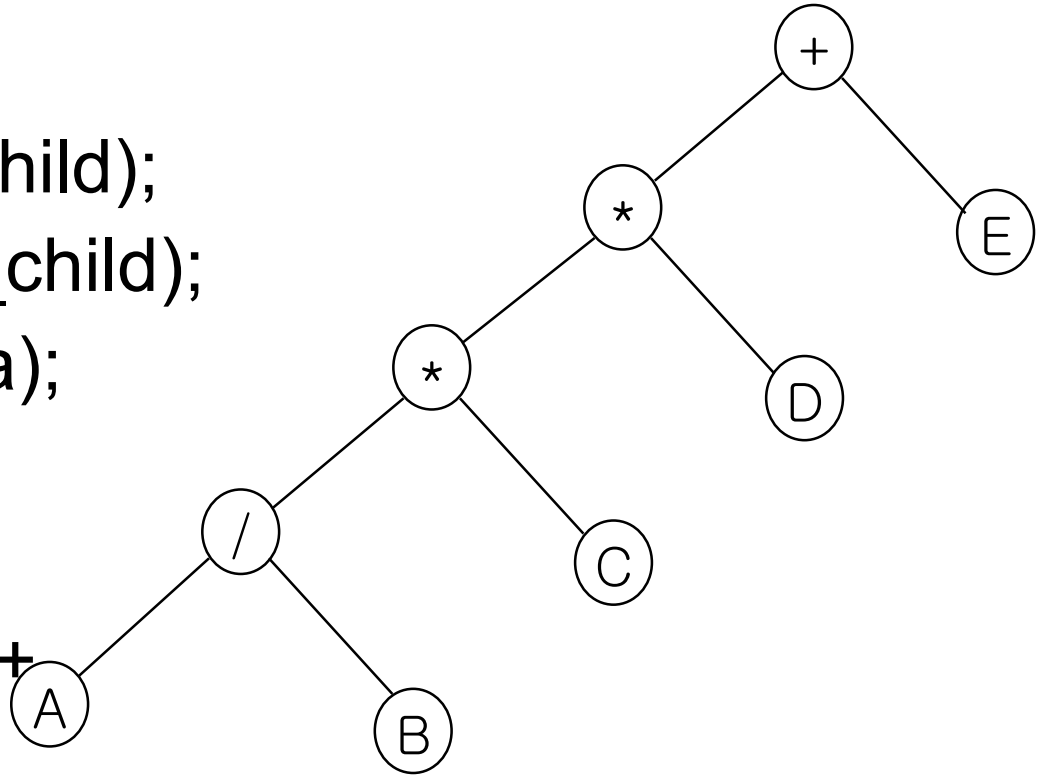


후위 순회(postorder traversal)

1. 노드를 방문하기 전에 두 자식을 먼저 방문
   노드 보다 그 노드의 자식이 먼저 출력

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

output A B / C * D * E +



반복적 중위 순회
   순환을 시뮬레이트 : 스택 필요

```
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(level order traversal): 큐를 사용하는 순회
1. 루트 먼저 방문, 2. 루트의 왼쪽 자식, 3. 루트의 오른쪽 자식

```
void level_order(tree_pointer ptr) /* 레벨 순서 트리 순회 */
{
   int front = 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;
   }
}// 그림 5.15,  output  + * E * D / C A B
```

이진 트리의 추가 연산

- 이진 트리의 복사 : 트리 순회를 이용하여 하나의 노드씩 복사

```
tree_pointer copy(tree_pointer original) //후위 순회 변형
{  /* 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");
         exit(1);
      }
      temp->left_child = copy(original->left_child);
      temp->right_child = copy(original->right_child);
      temp->data = original->data;
      return temp;
   }
   return NULL;
}
```

- 이진 트리의 동일성 검사
  - 동일한 구조와 정보?

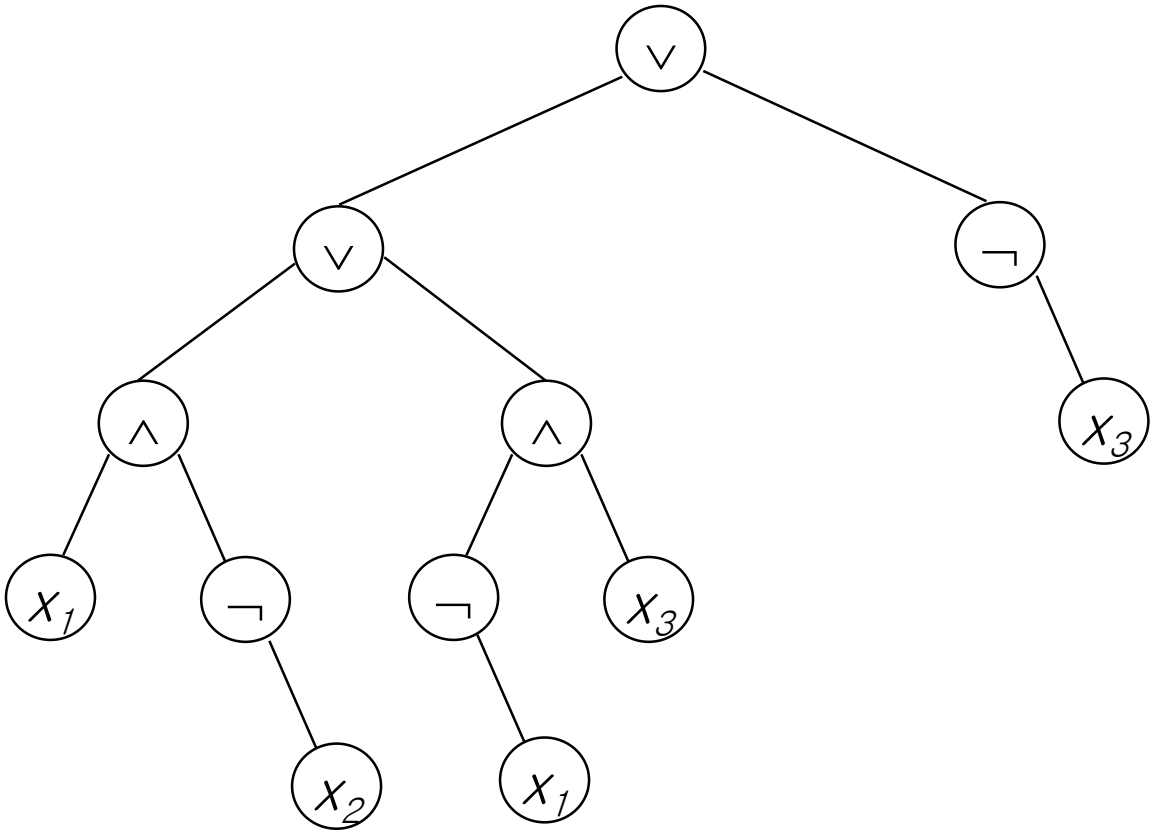
```
int equal(tree_pointer first, tree_pointer secon)
{ /* 두 이진 트리가 동일하면 TRUE, 그렇지 않으면 FALSE를 반
   환한다. */
   return ((!first && !second) || (first && second &&
           (first->data == second->data) &&
           equal(first->left_child, second->left_child) &&
           equal(first->right_child,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

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

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

- 가능한 모든 참과 거짓의 조합에 대해서 그 결과를 조사
  - 위의 경우에는 8가지의 조합이 가능함
  - 식을 계산하기 위해서는 두 subtree를 먼저 계산: 후위순회

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



```
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;
        } ;
```

- 리프 노드의 node->data는 이 노드가 나타내는 변수의 현재 값을 갖는다고 가정: x_1, x_2, x_3는 true 또는 false 가짐
- N 개의 변수를 갖는 명제식의 트리를 root가 가리킨다고 가정

만족성 알고리즘

```
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");
```

- 트리를 계산하는 함수는 후위 순회를 수정하여 작성

```
void post_order_eval (tree_pointer node)
{
   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;
                  break;
         case false: node->value = FALSE;
      }
   }
}
```

5.5 스레드 이진 트리

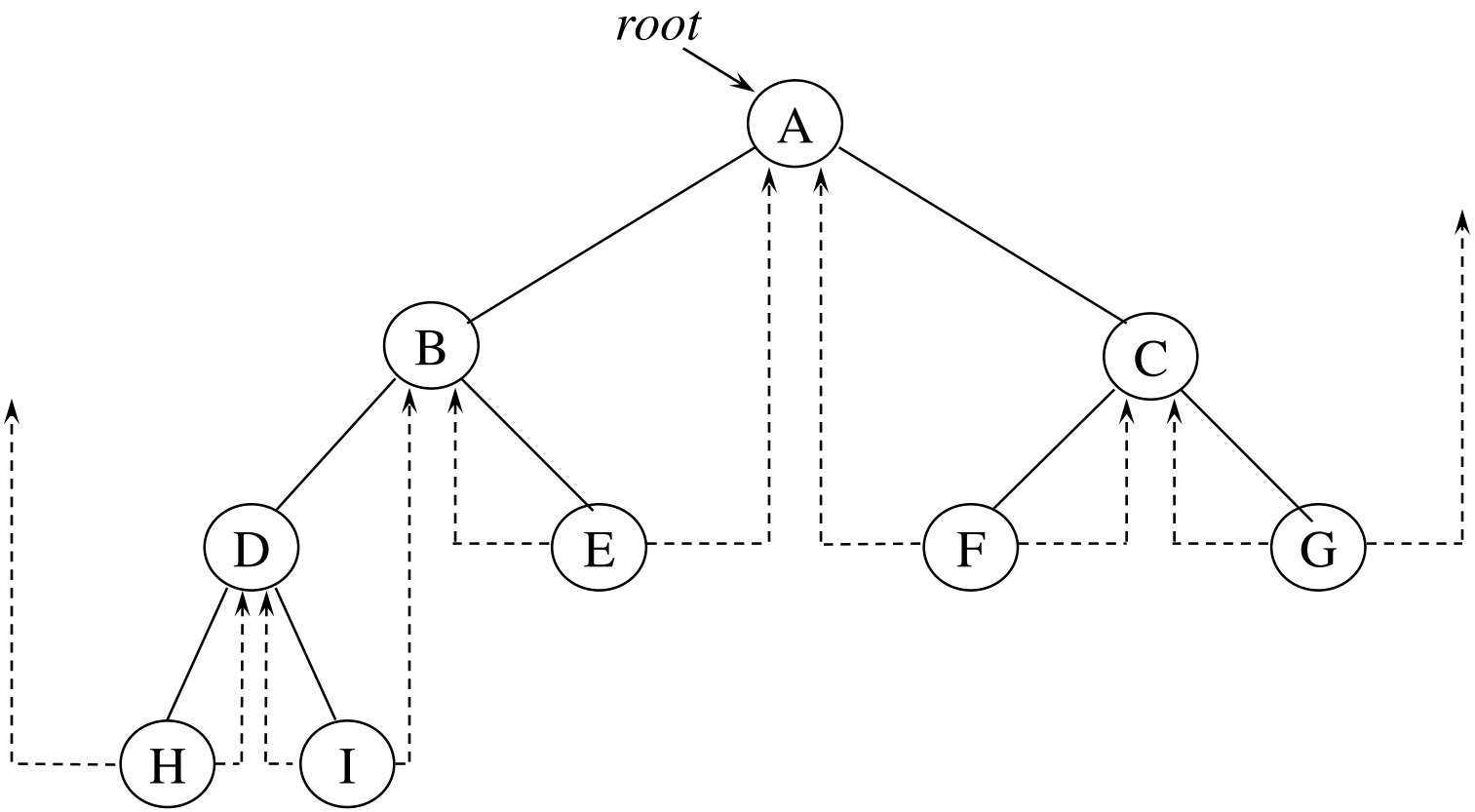
- 이진트리에는 2n개의 링크중 n+1개의 널링크
  - pf) 2 n_0 + n_1 = n + 1 ← p200

- 스레드(Thread : 실)
  - 널링크를 다른 노드를 가리키는 포인터로 사용

  ① ptr->left_child 가 NULL 이면 중위 순회시 ptr 앞에 방문하는 노드에 대한 포인터

  ② ptr->right_child 가 NULL 이면 중위 순회시 ptr 다음에 방문하는 노드에 대한 포인터

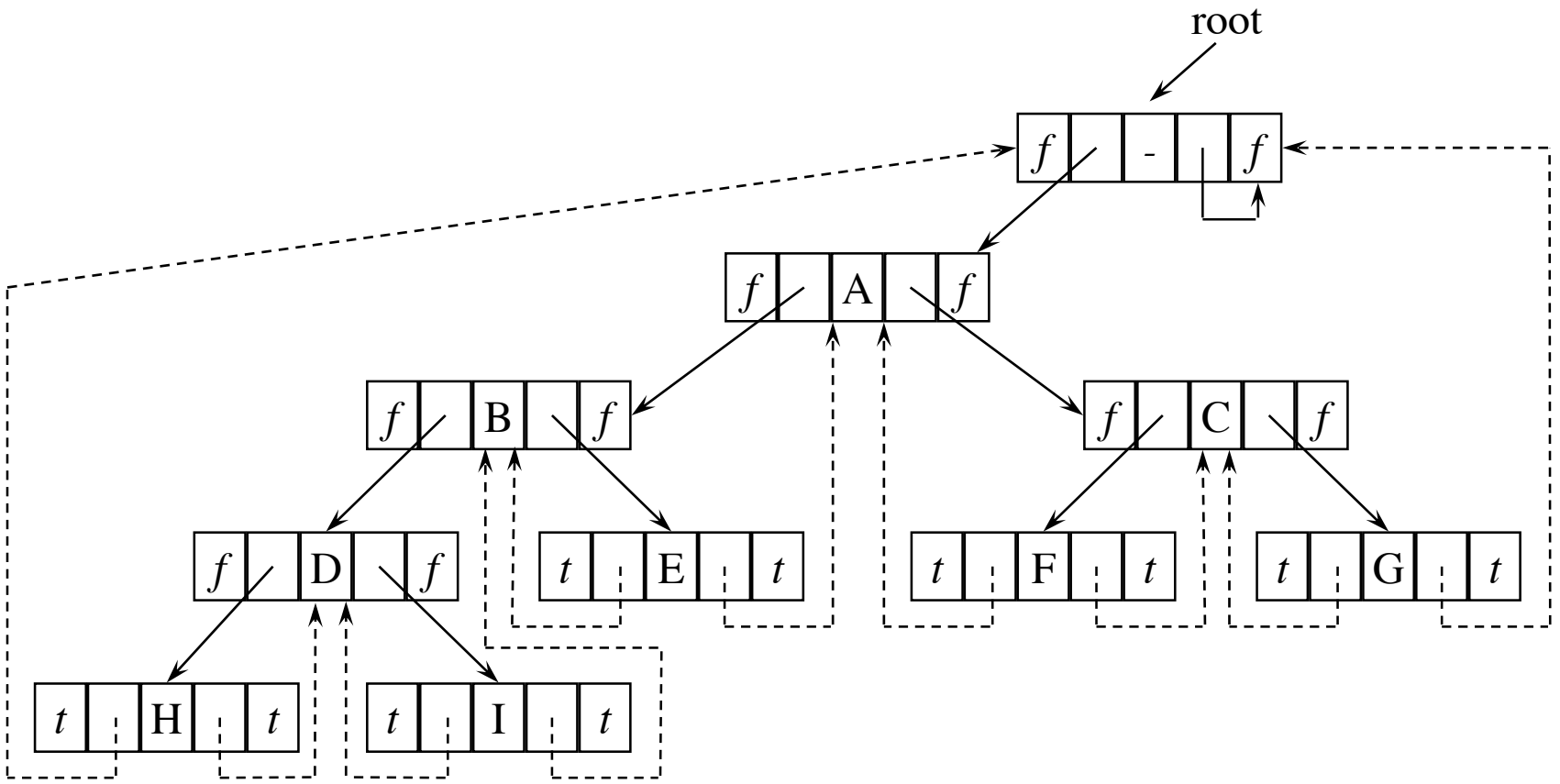
- 스레드 트리 예
  - 중위 순회 : H D I B E A F C G
  - 그림 5.14 (a) 스레드 이진트리는?



- 스레드 이진 트리의 표현
  - left_thread와 right_thread 라는 필드 첨가
  - left_thread가 true이면 left_child ← thread
    false이면 left_child ← pointer
  - right_thread가 true이면 right_child ← thread
    false이면 right_child ← pointer

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

- 분실 스레드
  - 첫번째 방문하는 H의 왼쪽 자식, 마지막 방문하는 G 오른쪽 자식
- 분실 스레드 해결 방법
  - 헤드노드 사용: root→left_child가 실제 트리의 첫번째 노드를 가리킴



f = FALSE; t = TRUE

스레드 이진 트리의 중위 순회

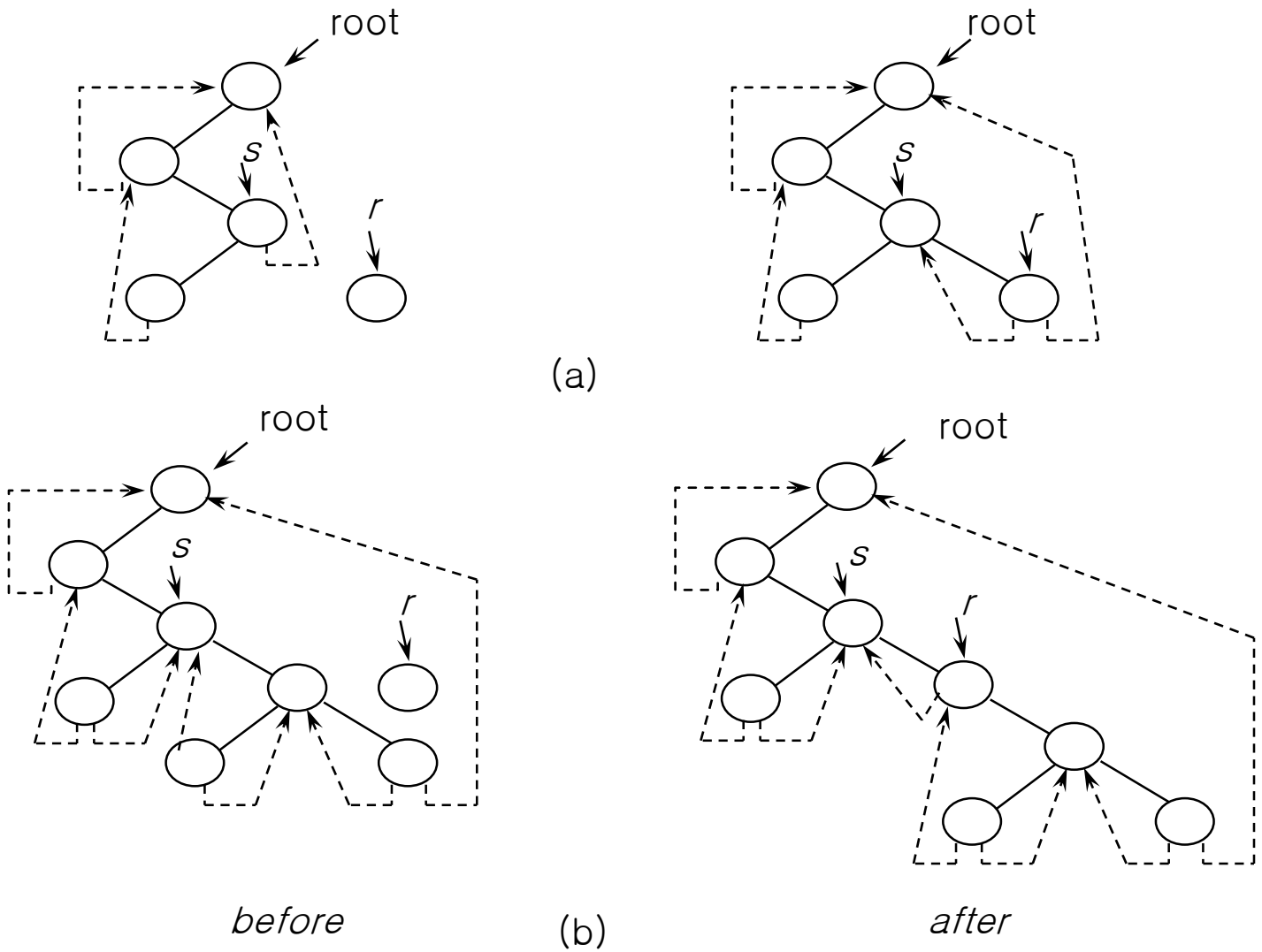
- 중위 후속자 찾기 : 다음 방문할 노드 찾기, 스택의 사용 없음

```
threaded_pointer insucc (threaded_pointer tree)
{
        threaded_pointer temp ;
        temp = tree→right_child ;
        if (!tree→right_thread) //right_thread가 false이면
                while (!temp→left_thread)
                        temp = temp→left_child ;
        return temp ;
} //헤드노드의 왼쪽자식은 트리를 가리키고, 오른쪽 스레드는 FALSE라 가정

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

스레드 이진 트리에서의 노드 삽입

- s의 오른쪽 자식으로 r을 삽입

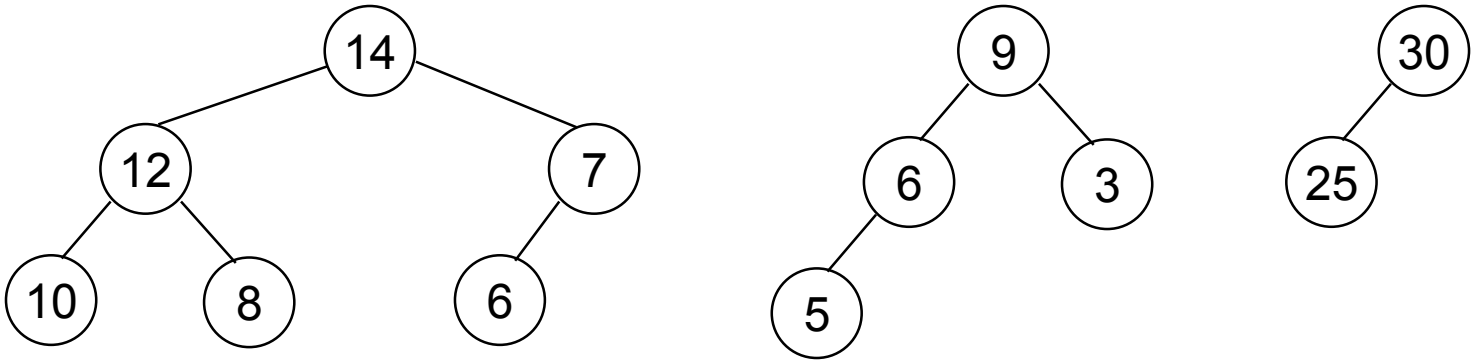


```
void insert_right (threaded_pointer parent, threaded_pointer child)
{
    // 스레드 이진 트리에서 child를 parent의 오른쪽 자식으로 삽입
    threaded_pointer temp ;
    child→right_child = parent→right_child ;
    child→right_thread = parent→right_thread ;
    child→left_child = parent ;
    child→left_thread = TRUE ;
    parent→right_child  = child ;
    parent→right_thread = FALSE ;
    if (!child→right_thread) { //parent의 오른쪽 자식이 있었으면,
          //child는 parent의 전 중위 후속자였던 노드의 중위 선행자가 됨
          temp = insucc (child) ;
          temp→left_child = child ;
    }
}
```

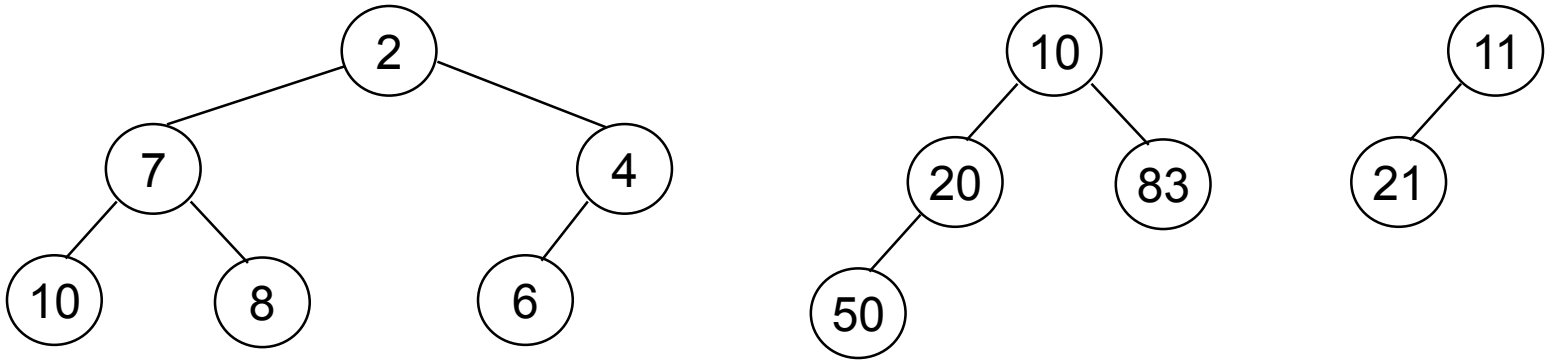
5.6 히프 (heap)

- 정의: 최대 트리는 각 노드의 키 값이 그 자식의 키 값보다 작지 않은 트리 이다. 최대 히프는 최대 트리이면서 완전 이진 트리이다.

- 정의: 최소 트리는 각 노드의 키 값이 그 자식의 키 값보다 크지 않은 트리 이다. 최소 히프는 최소 트리이면서 완전 이진 트리이다.



최대 히프



최소 히프

추상 데이타 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_size개의 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)) item을 히프에 삽입
하고 그 히프를 반환
else return 에러

Boolean HeapEmpty(heap, n) ::= if (n>0) return TRUE
else return FALSE

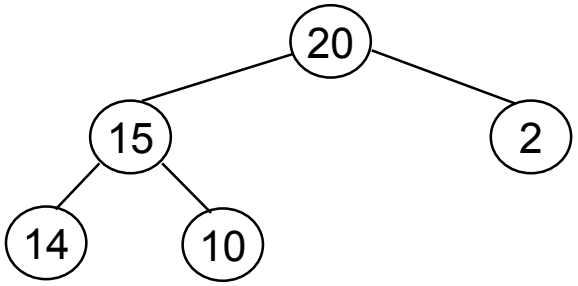
Element Delete(heap, n) ::= if (!HeapEmpty(heap, n)) 히프에서 가장 큰
원소를 제거하고 그 원소를 반환
else return 에러

- 우선 순위 큐(priority queue)
  - max(min) priority queue: 삭제시 highest(or lowest) priority를 가지는 것부터 삭제
  - Heap을 사용하여 구현 가능
  - 예) 운영체제 작업 스케쥴러

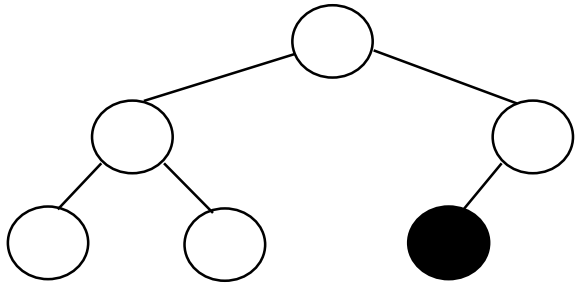
- 우선 순위 큐의 표현 방법

표현 방법	삽입	삭제
순서 없는 배열	Θ(1)	Θ(n)
순서 없는 연결 리스트	Θ(1)	Θ(n)
정렬된 배열	O(n)	Θ(1)
정렬된 연결 리스트	O(n)	Θ(1)
최대 히프	O(log_2 n)	O(log_2 n)

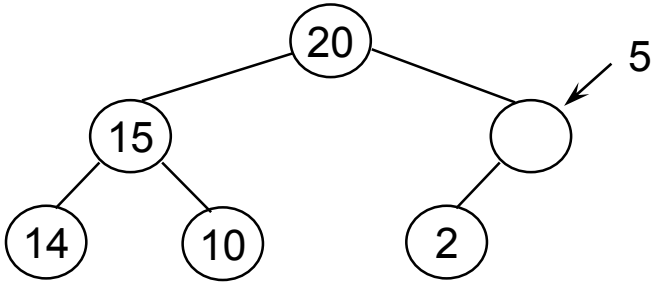
- 최대 히프에서의 삽입
  - 1, 5, 21을 삽입할 때의 예
  - p229 2. (a)



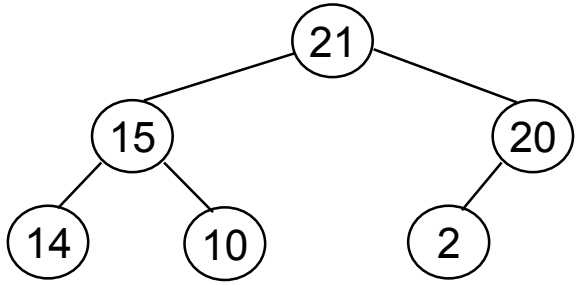
(a)삽입전 히프



(b) 새 노드의 초기 위치



©히프 (a)에 5를 삽입



(d) 히프 (a)에 21을 삽입

- 히프의 구현
  - 부모를 찾아갈 수 있어야 함
  - 완전 이진 트리이므로 배열을 사용하면 parent(i)는 $\lfloor \frac{i}{2} \rfloor$

```
#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;
```

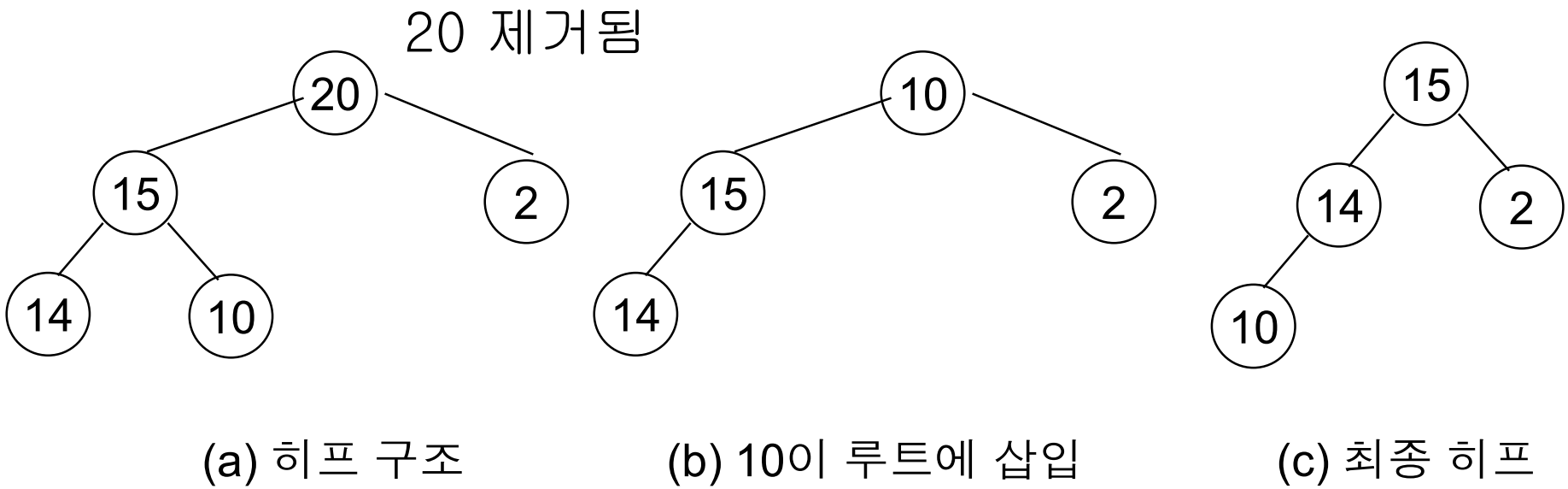
- 최대 히프에 삽입

```
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. ");
      exit(1);
   }
   i = ++(*n);
   while ((i != 1)&&(item.key>heap[i/2].key)){ //부모보다 크면
      heap[i] = heap[i/2]; //부모를 i위치에 저장
      i /= 2;                    // item의 위치를 부모위치 i/2로
   }
   heap[i] = item;
}
```

- Insert_max_heap 알고리즘의 분석
  - While 루프, 새로운 리프에서 루트까지의 경로를 따라 루트까지 도달하던가, 부모 위치 I/2의 값이 삽입되어질 값보다 큰 위치 I를 선택
  - n개의 원소를 가진 완전 이진 트리의 높이 $\lceil \log_2 (n+1) \rceil$
  - 연산 시간은 O(log_2 n)

중간고사 범위 여기까지.

- 최대 히프에서의 삭제
  - 최대값 얻기 위해 루트에서 삭제
  - 나머지 트리를 완전 이진 트리로 재구성해야 함
  - 히프를 재구성하기 위해 부모 노드와 자식 노드를 비교하여 순서가 틀린 원소들을 교환



(a) 히프 구조 (b) 10이 루트에 삽입 (c) 최종 히프

```
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"); exit(1);}
   /* 가장 큰 키값을 저장 */
   item = heap[1];
    /* 히프를 재구성하기위해 마지막 원소를 이용 */
   temp = heap[(*n)--];
   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;
}
```

- delete_max_heap 알고리즘의 분석
  - 부모와 자식 노드의 비교 While 루프
  - n개의 원소를 가진 완전 이진 트리의 높이 $\lceil \log_2 (n+1) \rceil$
  - 연산 시간은 O(log_2 n)

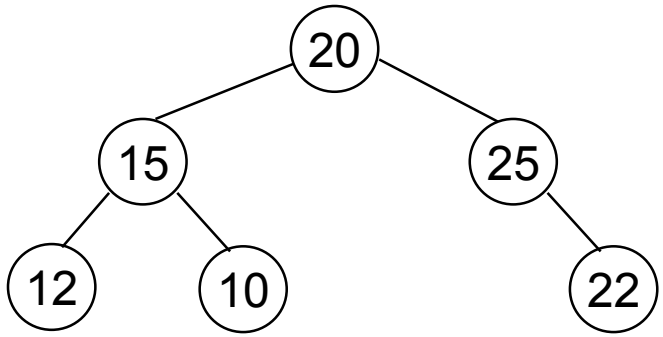
5.7 이진 탐색 트리

- 삽입, 삭제, 탐색 연산에 지금까지 자료구조중 좋은 성능
  - 히프는 임의의 원소의 삭제에 O(n)

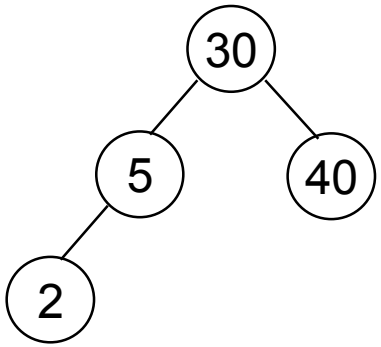
- 이진 탐색 트리는 이진트리로서 공백가능하고, 만약 공백이 아니라면
  1. 모든 원소는 서로 상이한 키를 갖는다.
  2. 왼쪽 서브트리의 키들은 그 루트의 키보다 작다.
  3. 오른쪽 서브트리의 키들은 그 루트의 키보다 크다
  4. 왼쪽과 오른쪽 서브트리도 이진 탐색 트리이다.

■ 이진 트리의 일종, 이전의 전위, 중위, 후위 순회를 수정 없이 사용할 수 있고, 이 절에서는 삽입, 삭제, 탐색 연산을 추가한다.

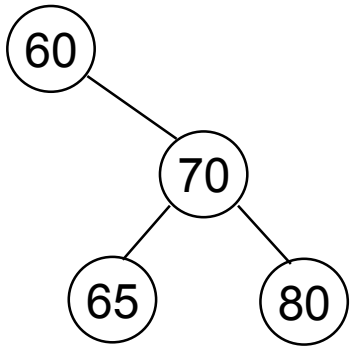
- 이진 탐색 트리 예 (그림 5.30)
  - 질문 : 이진 탐색 트리인가?



(a)



(b)



(c)

- 이진 탐색 트리의 탐색 : key 값을 찾는다.

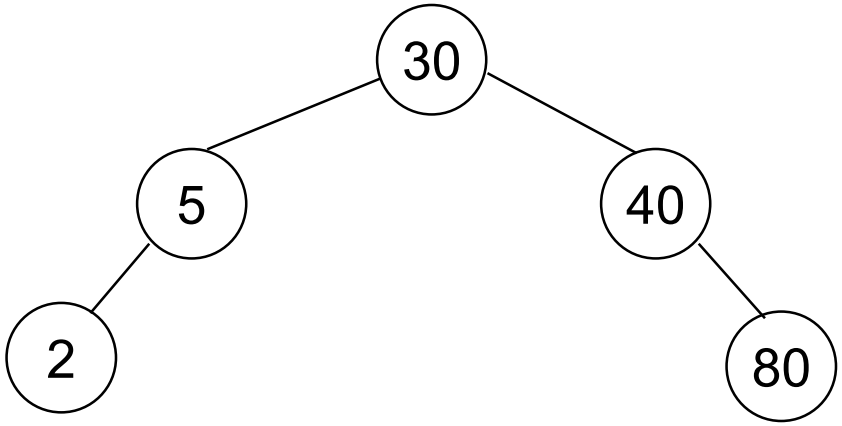
```
tree_pointer search(tree_pointer root,  int key) /* 순환 탐색 */
{  /* 키값이 key인 노드에 대한 포인터 반환. 그런 노드가 없는 경우에는 NULL 반환 */
   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
      return search(root->left_child, key);
   return search(root->right_child, key);
}
```

```
tree_pointer search2 (tree_pointer tree,  int key) /* 반복 탐색 */
{  /* 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;
}
```

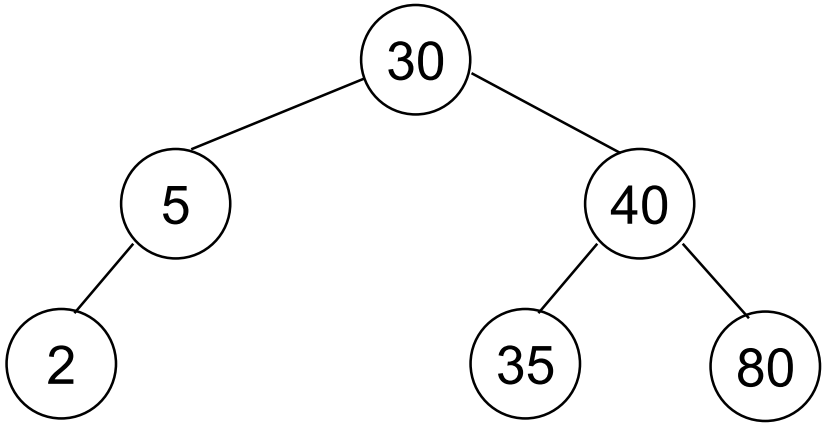
- search, serach2의 분석: 높이가 h인 이진 탐색 트리에 대해 O(h)

- 이진 탐색 트리에 대한 삽입
  - 동일한 key값을 가진 노드가 없어야 삽입가능
  - 탐색이 실패하면 탐색이 끝난 지점에 노드를 삽입
  - 그림 5.30(b)에 80 삽입 → 탐색이 끝난 40의 오른쪽 자식에 80을 삽입 → 5.31 (a)
  - (a)에 35 삽입 → (b)

- 그림 5.31



(a) 80을 삽입



(b) 35를 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)) { // temp가 널이 아니거나, 널이라도 *node가 널
        /* num이 트리내에 없음 */
       ptr =  (tree_pointer)malloc(sizeof(node)) ;
       if (IS_FULL(ptr)) {
          fprintf(stderr, "The memory is full");
          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:
    }
}
```

- modified_search
  - search2를 수정
  - 이진 탐색 트리 *node에서 키값 num을 탐색하는데 만약 트리가 공백이거나 num이 존재하면 NULL을 반환하고, 그렇지 않으면 탐색 도중에 마지막으로 검사한 노드에 대한 포인터를 반환한다.

- insert_node의 분석
  - 높인 h인 트리에서 num을 탐색하는데 O(h)
  - 나머지는 Θ(1)
  - 전체 시간은 O(h)

- 이진 탐색 트리에서의 삭제 : 세가지 경우가 존재

1. 단말 노드의 삭제 : 부모의 자식 필드에 NULL을 삽입
   - 그림 31(b)에서 35를 삭제하려면 그 부모 노드의 왼쪽 자식 필드를 NULL로 만들고 노드를 반환, 결과는 그림 31(a)
2. 하나의 자식을 가진 비단말 노드의 삭제
   - 삭제된 노드의 자식을 삭제된 노드의 자리에 대체
   - 그림 5.31(a)에서 40을 삭제하면 그림 5.32의 트리

3. 두개의 자식을 가진 비단말 노드의 삭제 :

   i. 왼쪽 서브트리에서 가장 큰 원소로 치환, 또는

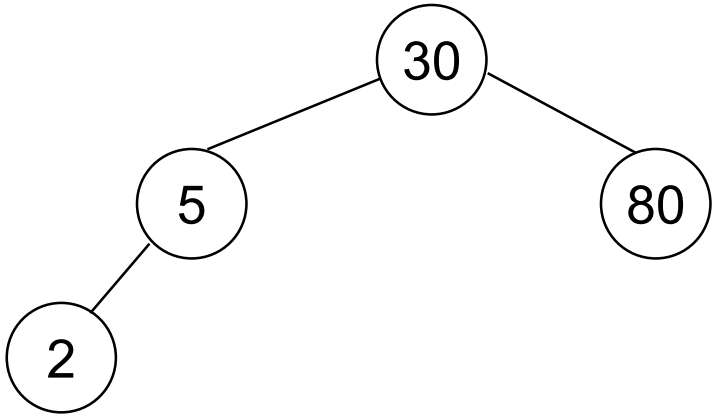
   ii. 오른쪽 서브트리에서 가장 작은 원소로 치환

대체하려고 선택된 노드를 A라 하면

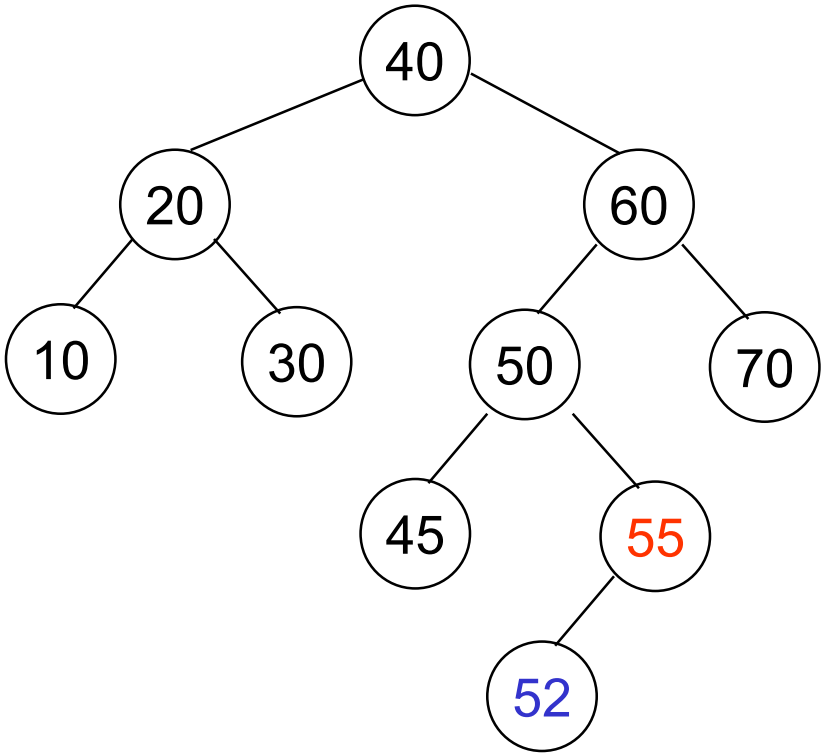
A의 자식을 A의 부모 노드의 자식으로 연결하고, A를 반환

(참고) A의 자식은 0이거나 1 (A는 가장 크거나 작으므로)

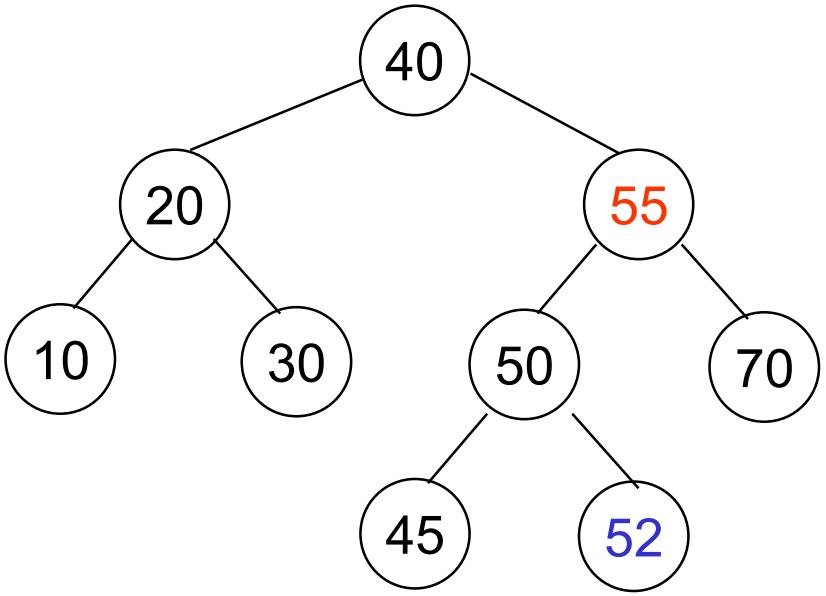
- 그림 5.32



- 그림 5.33 두 자식을 가진 노드의 삭제



(b) 60을 삭제하기 전의 트리



(b) 60을 삭제한 후의 트리

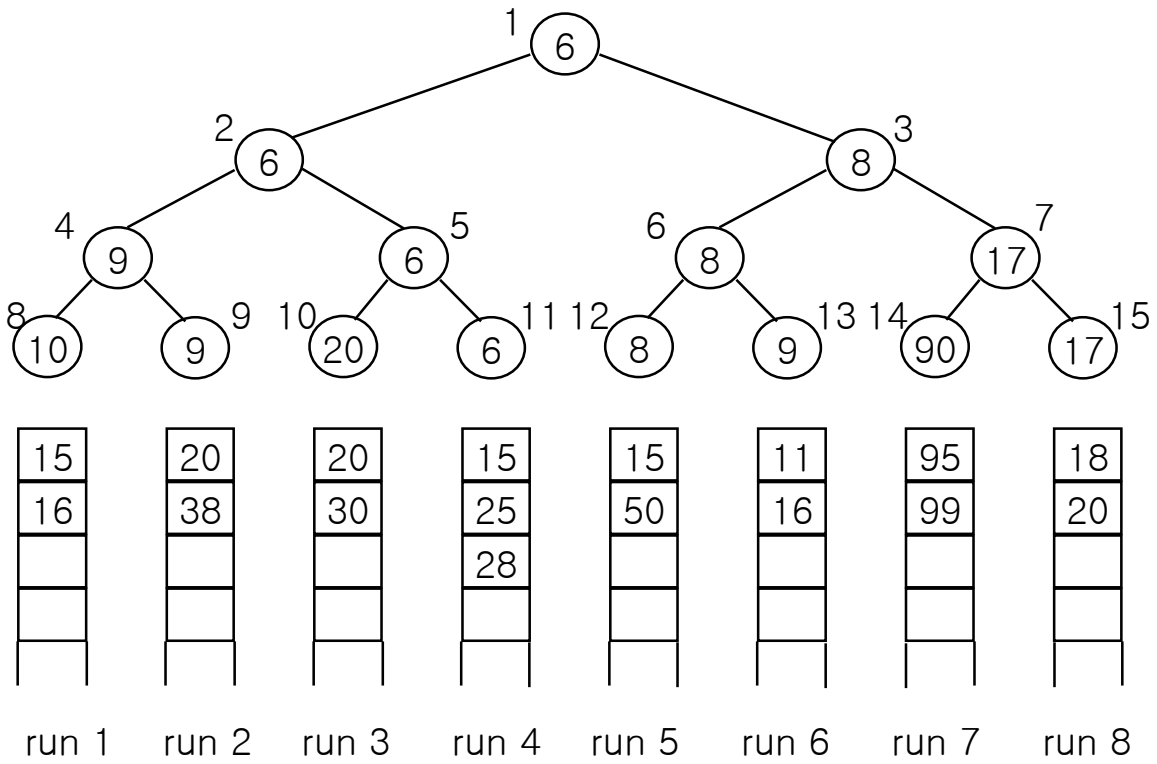
이진 탐색 트리(n)의 높이

- worst case : height = n
  - [1,2,3,...,n]으로 삽입할 때
- average : height = O(log n)

- 최악의 경우에도 height = O(log n)이 되는 트리→ 균형 탐색 트리(balanced search tree)
  - 탐색, 삽입, 삭제를 O(h) 시간에 할수 있다.
  - 10장: AVL, 2-3, 2-3-4, red-black, B 트리

5.8 선택 트리

- K 개의 런(run)에서 가장 작은 키의 레코드를 출력
  - 런 : key에 따라 비감소 순서로 정렬된 레코드로 구성
  - 직접적인 방법은 k-1번 비교
  - 선택트리 사용하여 k-1 보다 적은 비교로 가능

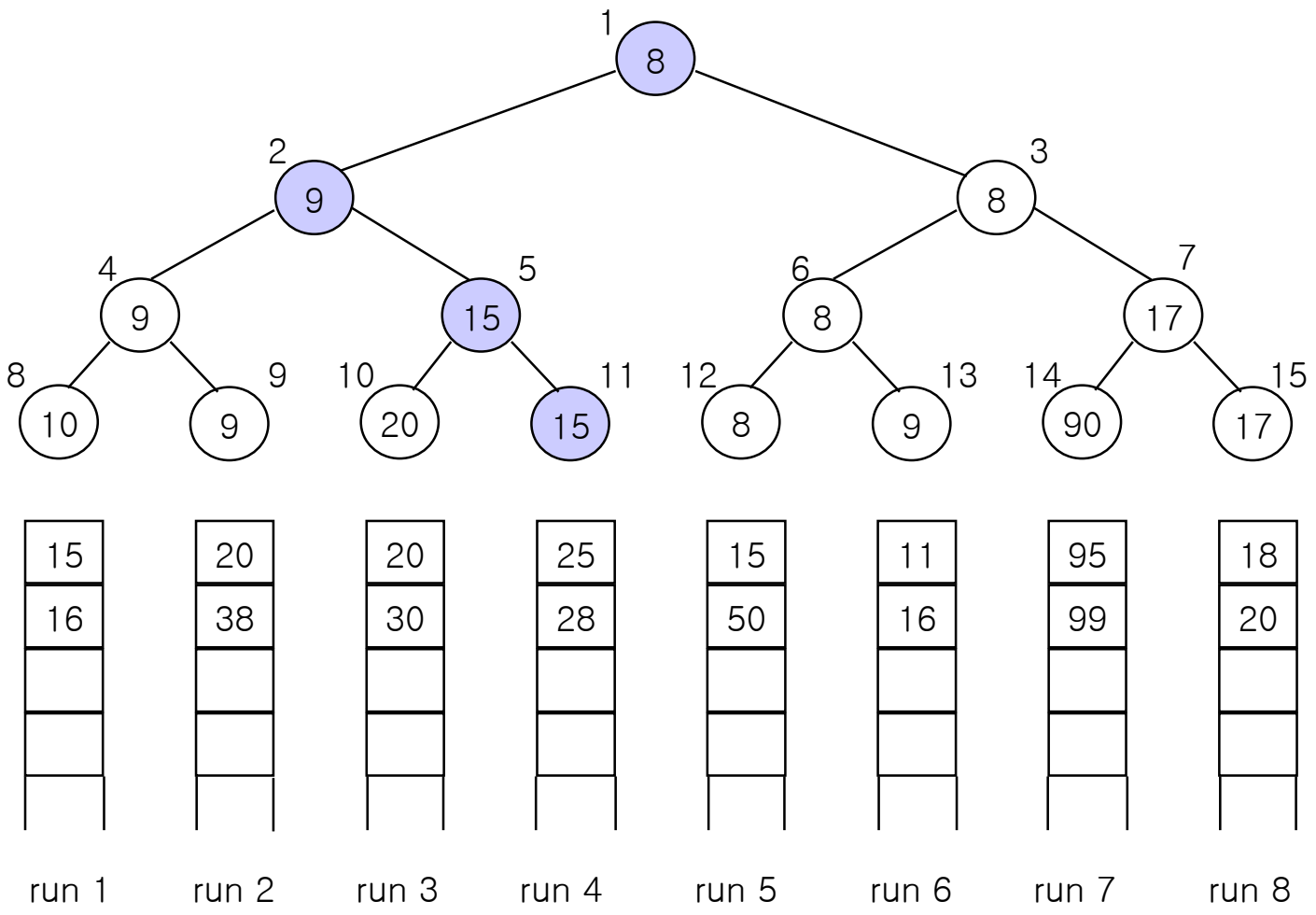


- 선택 트리의 구성
  - 단말 노드 : 해당 런의 첫번째 레코드
  - 각 비단말 노드 : 토너먼트의 승자
  - 루트 노드 : 전체 승자 또는 가장 작은 값

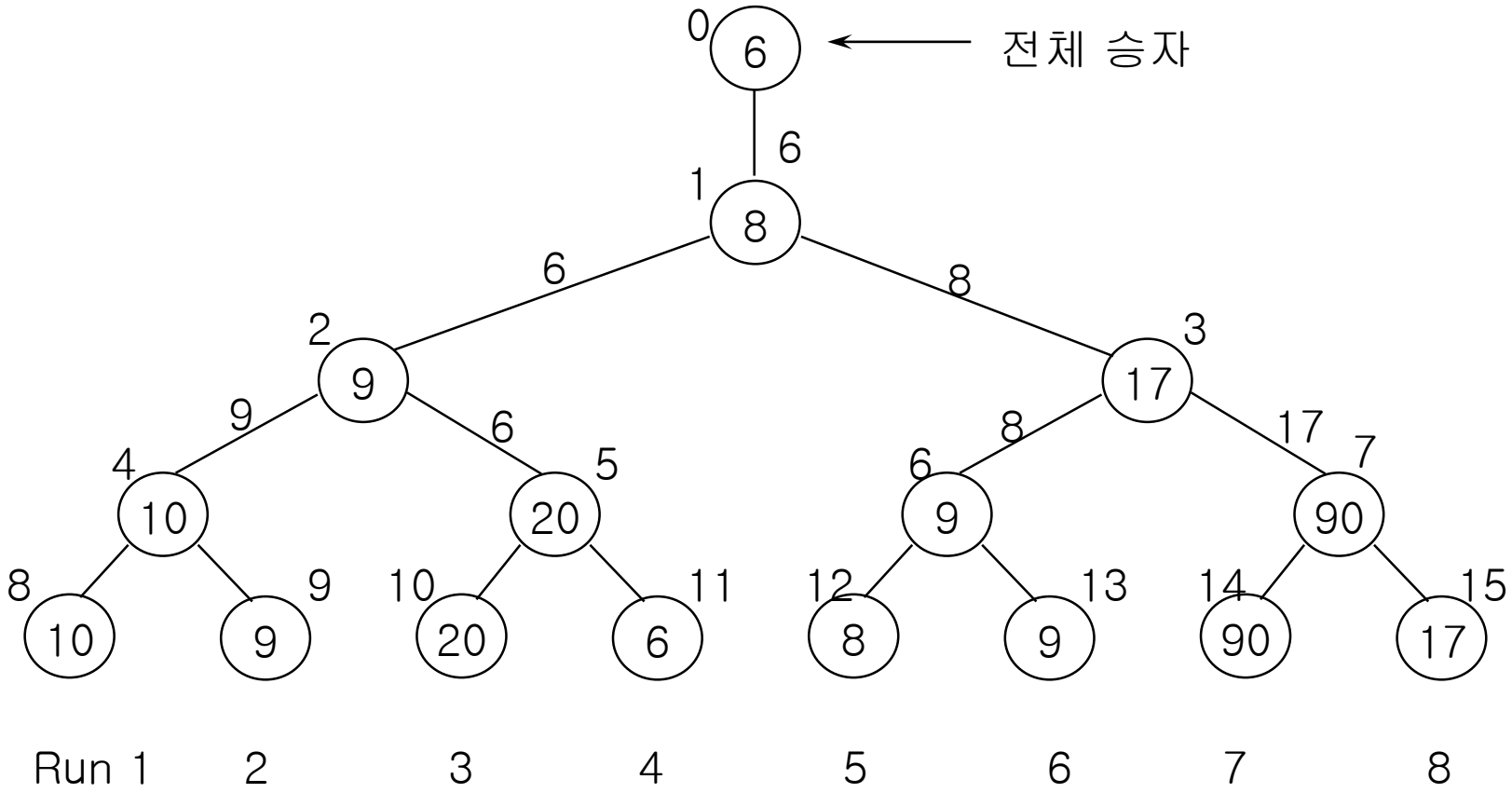
- 선택 트리 구현
  - 노드는 레코드에 대한 포인터만을 포함
  - 배열 표현 방식 사용 (why ?) : 노드 번호는 노드의 주소

- 시간 복잡도
  - 비교는 단말노드에서 루트로 가는 경로에서 발생
  - 트리의 레벨수는 $ceil\ (\log_2^{k+1})$ 따라서, 트리의 재구성 시간은 O(log k), n개의 모든 레코드를 모두 합병하는데 O(n log k)
  - 처음 선택 트리 설정 O(k)이므로 전체 시간은 O(n log k)

- 승자 트리에서 한 레코드가 출력되고 나서 재구성된 모습(변경된 노드는 표시되어 있음)
  - 연습



- 패자 트리
  - 노드가 패자를 표현, 좀 더 빠른 알고리즘
  - 형제 노드들은 전에 시행되었던 토너먼트의 패자
  - 비단말 노드는 패자, 승자는 상위 노드로 진출
  - 각 노드는 레코드에 대한 포인터 저장, 그림에서는 편의상 레코드의 키 값 사용, 노드 0은 전체 토너먼트의 승자



5.9 포리스트

- 포리스트 : n >= 0 개의 분리 트리의 집합

- 포리스트의 이진 트리 변환
  - 변환된 이진 트리들을 루트의 오른쪽 서브트리로 연결
  - 정의 : T1,T2,...,Tn이 트리들로 이루어진 포리스트라 하면 이 포리스트에 대응하는 이진 트리, B(T1,T2,...,Tn)는

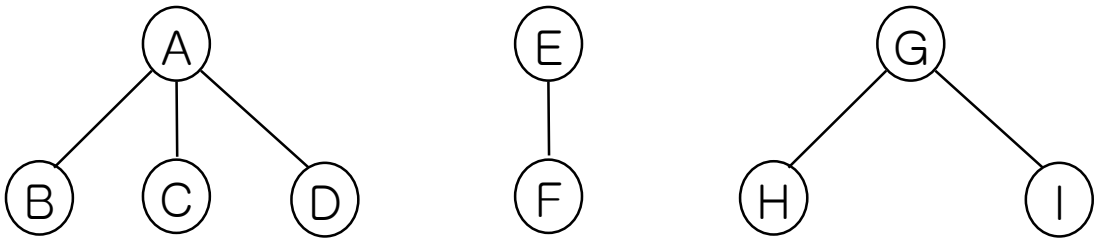
    (1) n=0이면 공백

    (2) B는 T1과 같은 루트를 가지며,

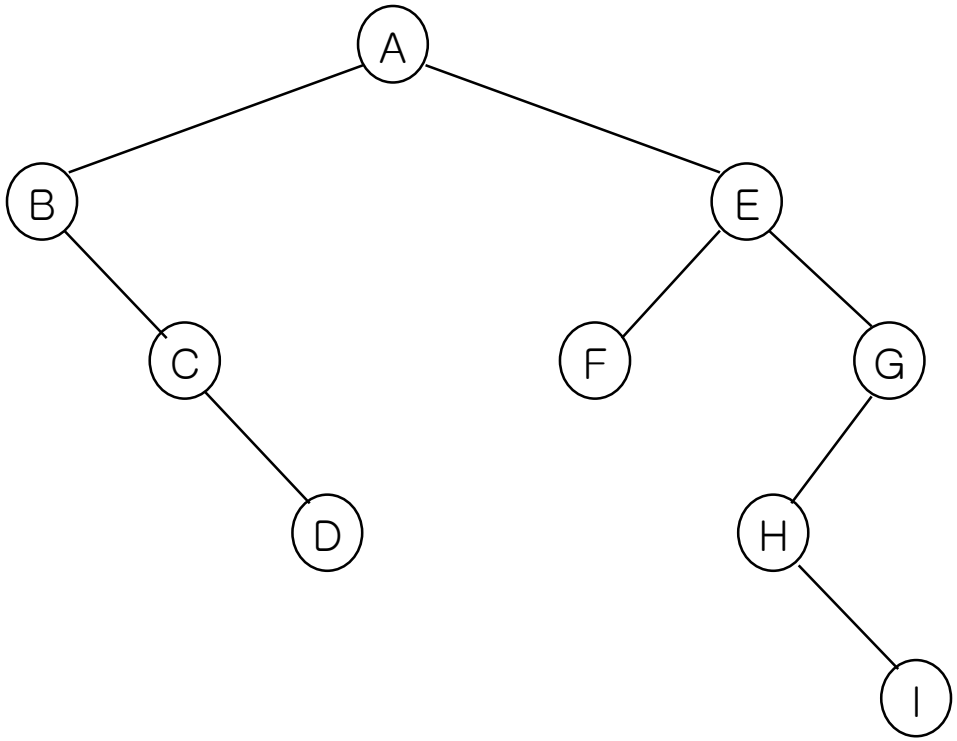
    왼쪽 서브트리로 B(T11,T12,...,T1m), 여기서, T11,T12, ..., T1m는 T1의 서브트리

    오른쪽 서브트리 = B(T2,...,Tn)

- 세 개의 트리로 구성된 포리스트



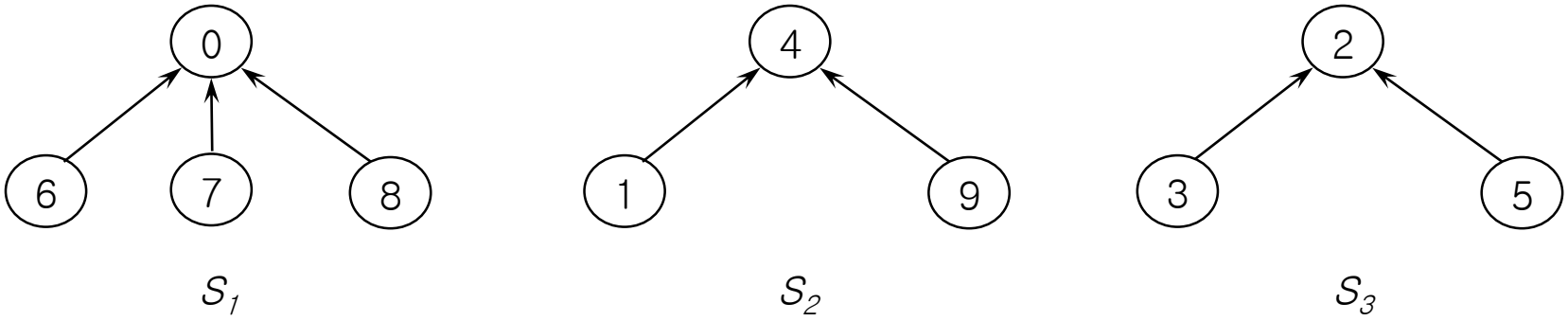
- 변환된 이진 트리 표현



- 포리스트 F에 대응하는 이진 트리 T의 전위, 중위 순회는 F의 순회와 대응관계
- 포리스트 전위(forest preorder)
  (1) F가 공백이면 귀환
  (2) F의 첫 트리의 루트를 방문
  (3) 첫 트리의 서브트리들을 포리스트 전위 순회
  (4) F의 나머지 트리들을 포리스트 전위로 순회
- 포리스트 중위(forest inorder)
  (1) F가 공백이면 귀환
  (2) F의 첫 트리의 서브트리를 포리스트 중위로 순회
  (3) 첫 트리의 루트를 방문
  (4) 나머지 트리들을 포리스트 중위로 순회
- 포리스트 후위(forest postorder)
  (1) F가 공백이면 귀환
  (2) F의 첫 트리의 서브트리를 포리스트 후위로 순회
  (3) 나머지 트리들을 포리스트 후위로 순회
  (4) F의 첫 트리의 루트를 방문

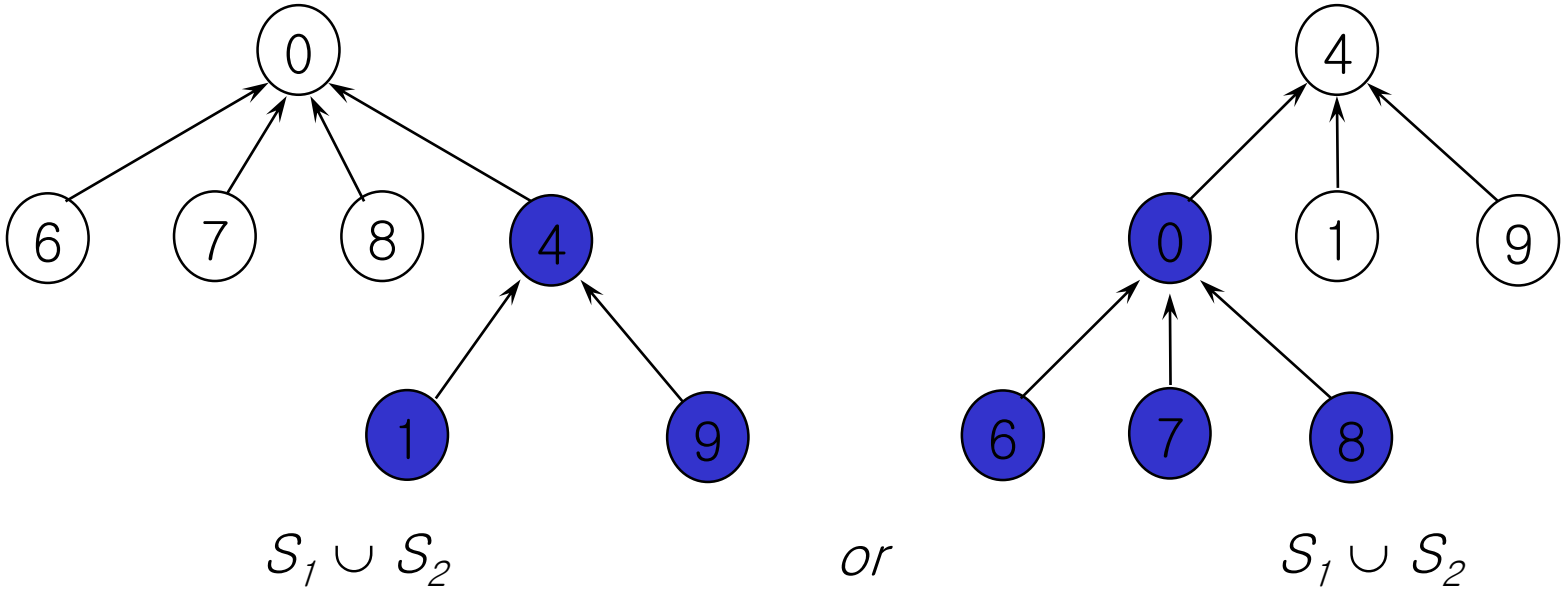
5.10 집합 표현

- 트리를 사용한 집합 표현
  - 그림 5.39: 자식에서 부모로 가는 링크로 연결



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

- 집합의 연산
  - Union : S_1 U S_2 = {0, 6, 7, 8} U {1, 4, 9} = {0, 6, 7, 8, 1, 4, 9}
  - Find (I): 원소 I를 포함하는 집합을 찾음, 3은 S_3에 8은 S_1에 있음
- Union(합집합) 연산
  - 두 트리 중의 하나를 다른 트리의 서브트리로 만듦
  - 한 루트의 부모 필드가 다른 트리의 루트를 가리키도록 함

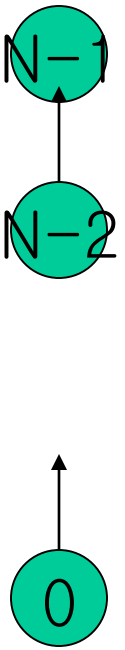


- find(i)
  - i에서 시작하여 음수값의 부모 인덱스를 따라 감
  - find(5)는 5→2→-1

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

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

- union1 과 find1의 분석
  - S_I = {I}인 집합, 다음의 연산을 수행



union(0, 1), union(1, 2), … , union(n-2, n-1),
find(0), find(1), … , find(n-1)

union 연산은 상수이므로 n-1개 수행 O(n)

find 는 레 벨 I 에 대 해 서 는 O(I) 이 므 로 find(0), find(1), ... ,find(n-1) = 1+2+…+n: O(n^2)

- union 연산의 효율적 구현
- union(I,j)를 위한 가중법칙
  - 트리 I의 노드수 < 트리 j의 노드수 → j가 I의 부모
  - 트리 I의 노드수 > 트리 j의 노드수 → i가 j의 부모

  - 가중 법칙을 적용하면 그림 5.44

  - 가중법칙을 적용하기 위해, 트리의 노드수를 루트노드의 parent 필드에 음수로 저장하면 됨

```
 void union2 (int I, int j)
{/* 가중법칙을 이용하여 루트가 I와 j인 집합을 합집합함.
   Parent[I] = -conunt[I], parent[j] = -count[j]

 int temp = parent[I] + parent[j] ;
 if (parent[I] > parent[j]) { // 음수이므로 i쪽이 노드수가 작다
   parent[I] = j; /*j를 새 루트로 만듦 */
  parent[j] = temp ;
 }
 else {
    parent[j] = i; /*i를 새 루트로 만듦 */
  parent[i] = temp ;
 }
}
```

- 보조 정리 5.4: n개의 노드를 가진 트리 T가 union2 함수로 만들어진 트리 라면, T에 있는 어떤 노드도 |_ log_2 n _| + 1 보다 큰 레벨을 가질 수 없다.

- 보조 정리 5.4에 따라 union2로 만들어진 트리에서 find 연산 시간 O(log_2 n)이므로 n-1개의 union과 m개의 find를 실행하는 시간은 O(n + mlog_2 n)

- 예제 5.1
  - 결과는 그림 5.45

- 예제 5.2
  - 그림 5.45에서 8개의 find(7)를 수행하면 각각 3개의 parent 링크 필드를 거쳐야 하므로, 24개의 링크를 이동

  - 붕괴법칙을 사용하여 find를 수정한 find2는 처음 find2는 세개의 링크를 거치고 세개의 링크를 변경, 나머지는 7개는 하나의 링크만 거치면 되므로 총 13번 이동

- 붕괴법칙: 만일 j가 I에서 루트로 가는 경로 위에 있다면 j를 루트의 자식으로 만든다.

```
 int find2(int I)
{ /* 원소 I를 포함하는 루트를 찾음. 붕괴법칙을 사용하여 I 로부터 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 ;
}
```