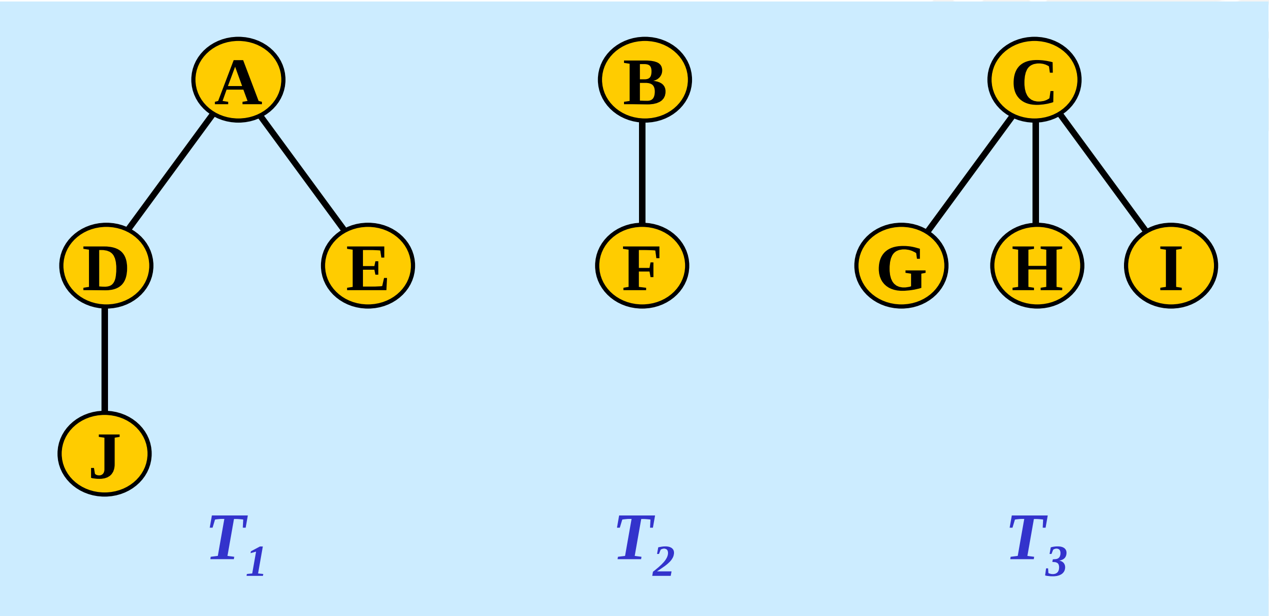
CSE2010 자료구조론

Week 8: Forest, Binary Search Tree

ICT융합학부 조용우

포리스트(Forest)?

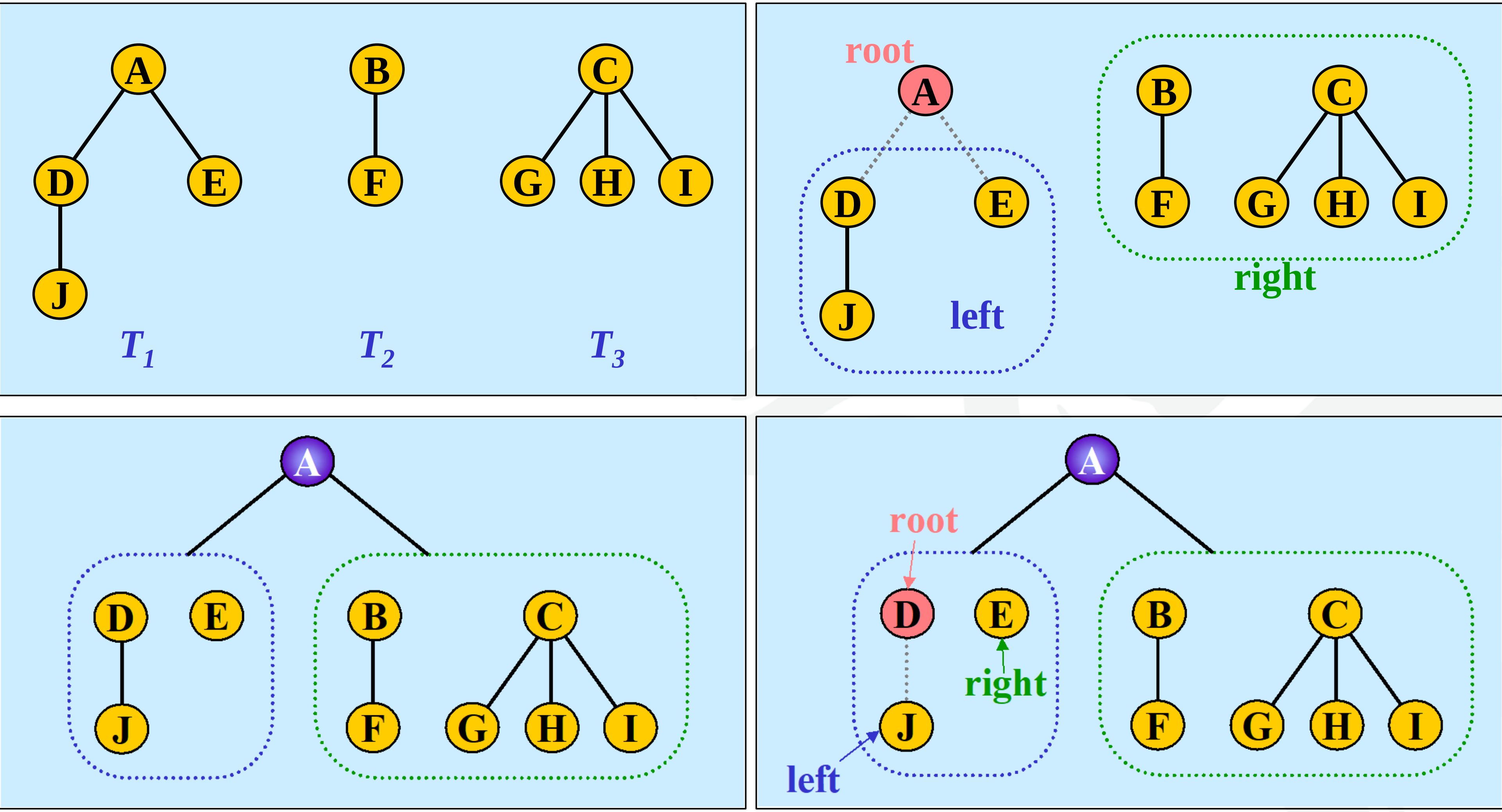
- 포리스트(forest) : m ($m \geq 0$)개의 트리로 구성된 집합



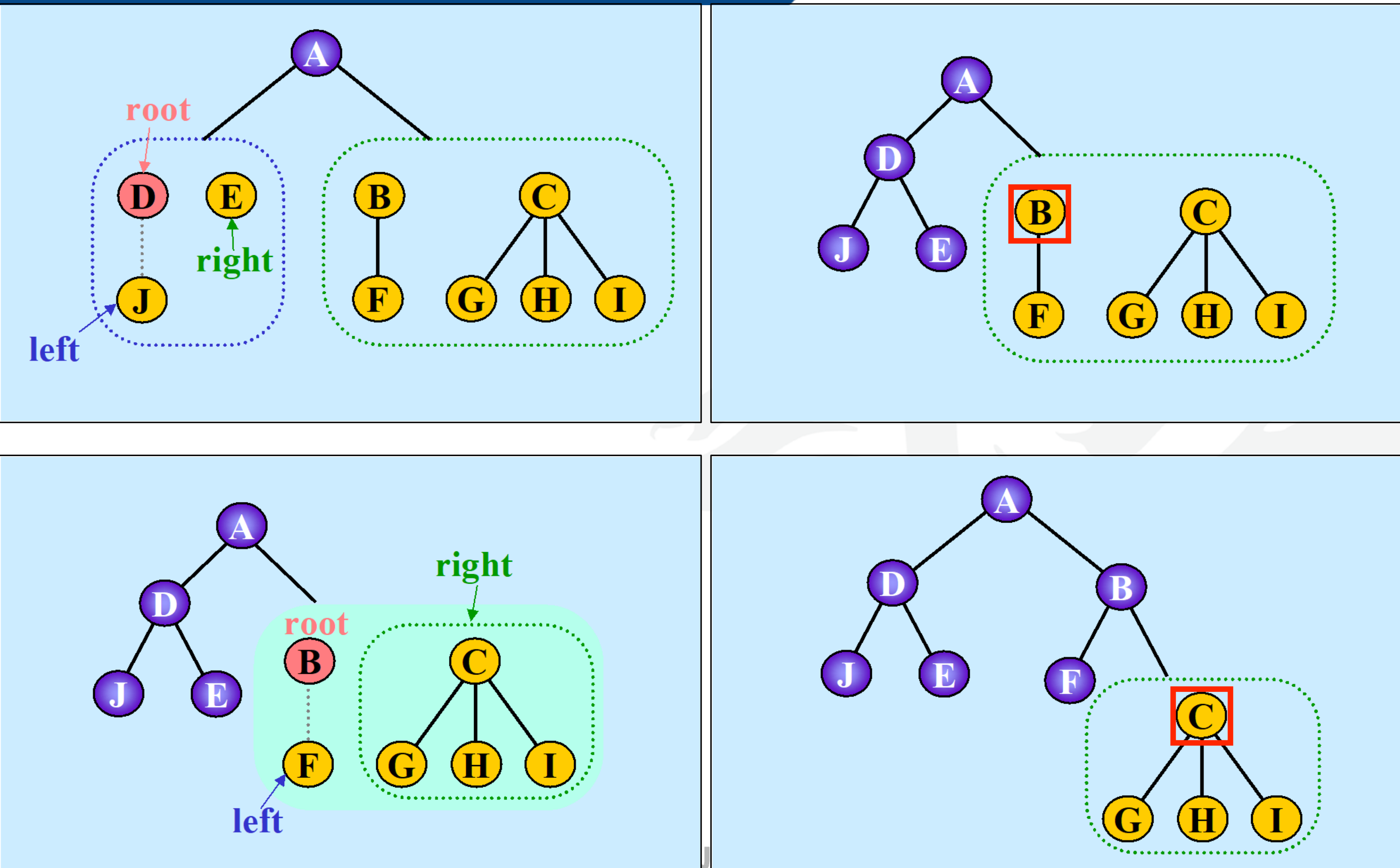
포리스트 -> 이진트리

- 포리스트 $F = \{T_1, T_2, \ldots, T_m\}$을 이진트리 $BT(\{T_1, T_2, \ldots, T_m\})$로 변환하는 방법
  - ① $m = 0$인 경우, 즉 $F = \emptyset$ 이면, 대응되는 $\mathrm{BT}(\emptyset) = \emptyset$ 이다
  - ② $m > 0$인 경우,
    - 루트 : T_1의 루트 $Root(T_1)$
    - 왼쪽 부분트리 : $Root(T_1)$의 부분트리 $BT(\{T_{11}, T_{12}, \ldots, T_{1r}\})$
    - 오른쪽 부분트리 : $BT(\{T_2, T_3, \ldots, T_m\})$

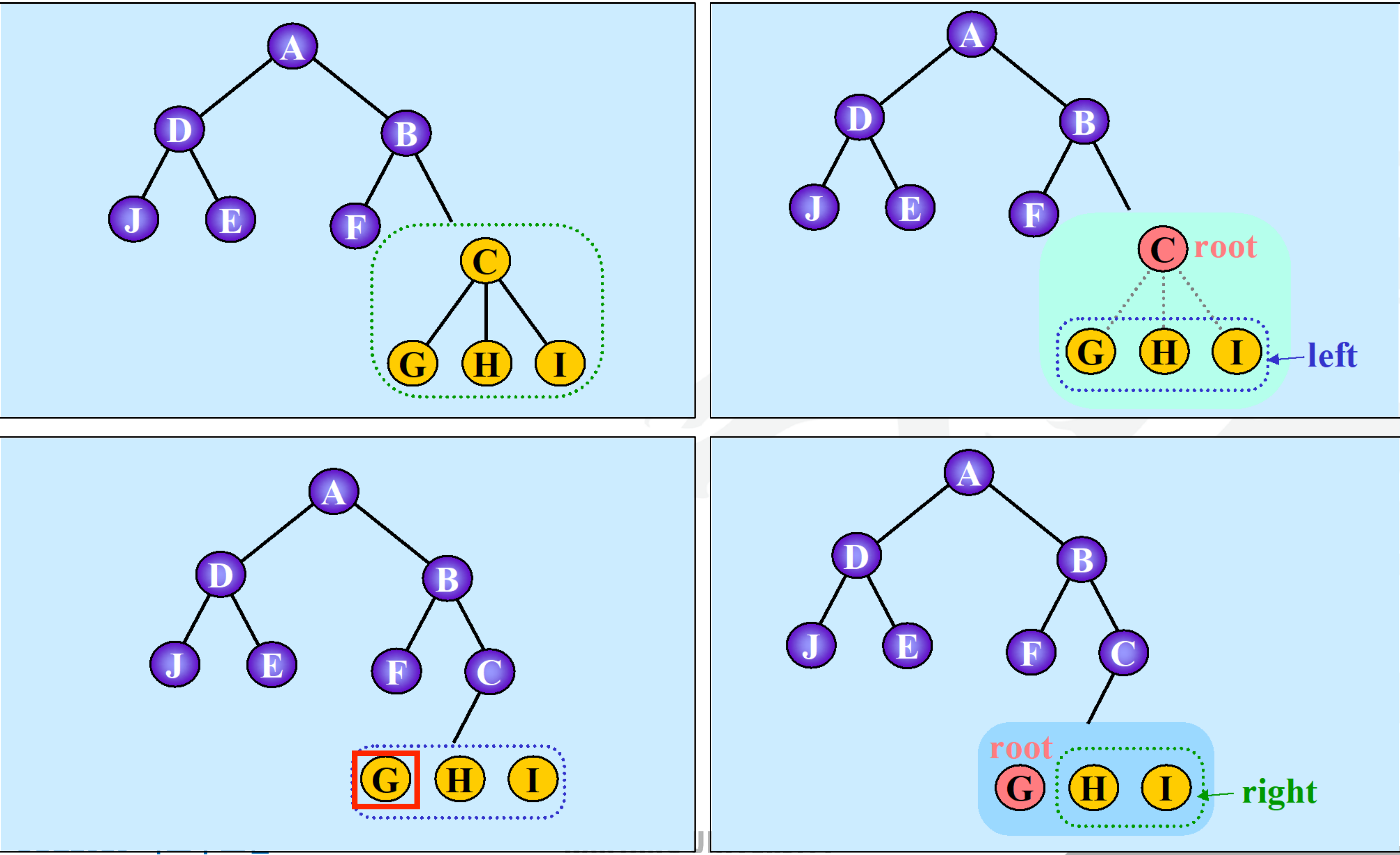
포리스트 -> 이진트리 예


A
D
E
J
T_1
B
F
T_2
C
G
H
I
T_3
root
A
D
E
J
left
B
F
C
G
H
I
right
A
D
E
J
B
F
C
G
H
I
A
root
D
E
right
J
left
B
F
C
G
H
I

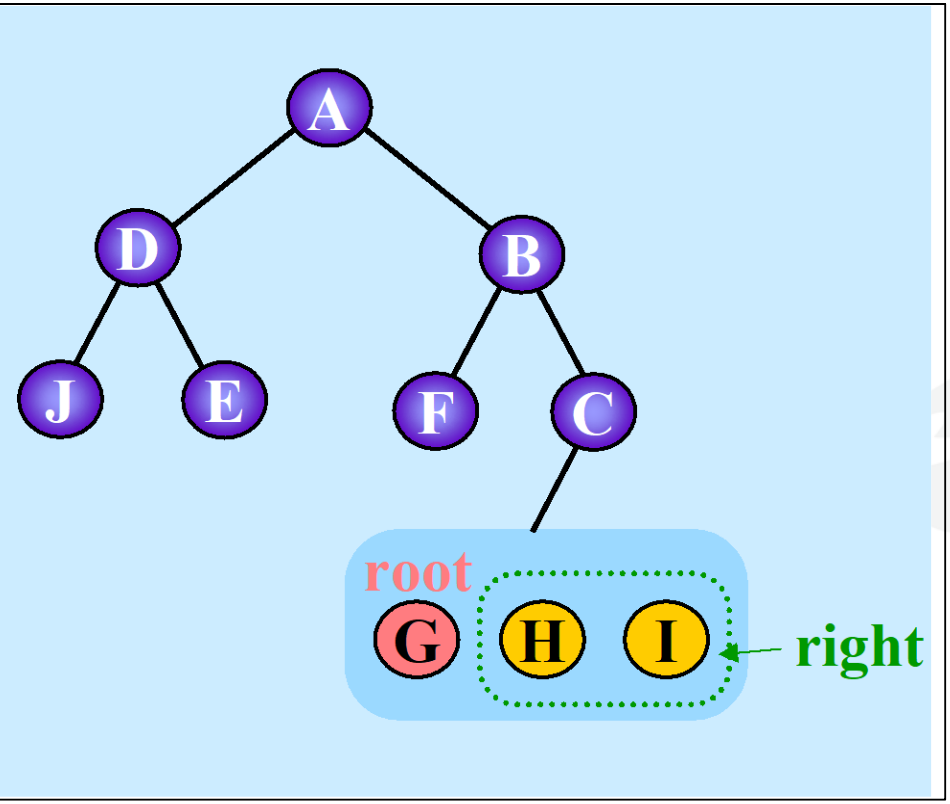
포리스트 -> 이진트리 예

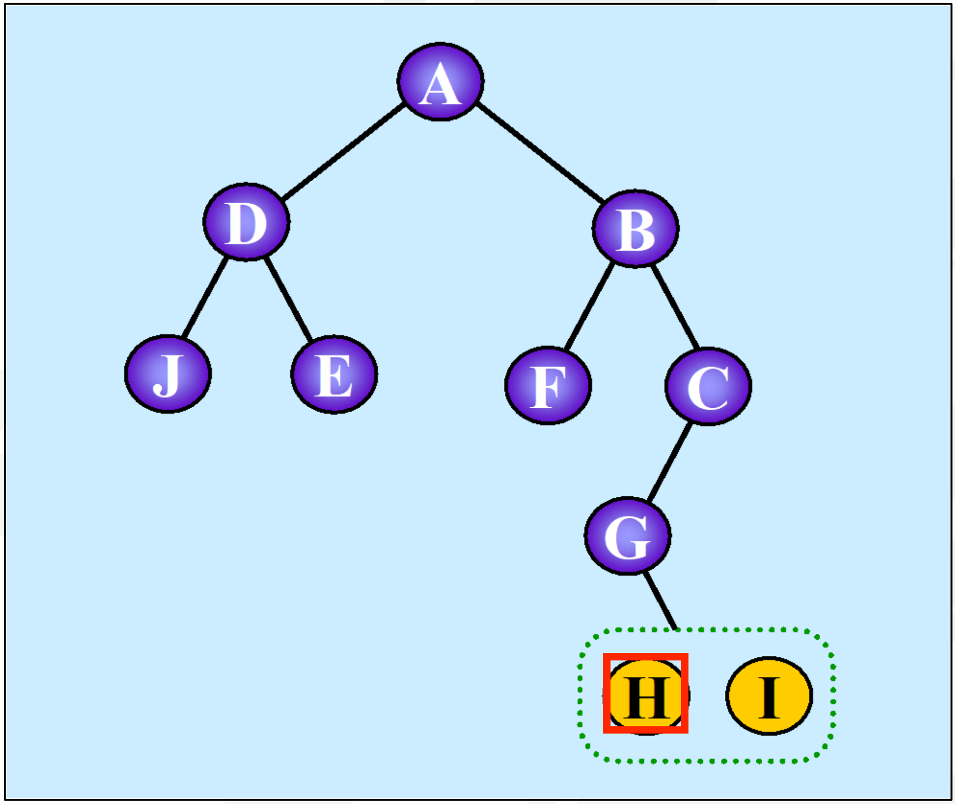

A
root
D
E
right
J
left
B
C
F
G
H
I
A
D
J
E
B
C
F
G
H
I
A
D
J
E
right
root
B
C
F
G
H
I
left
A
D
B
J
E
F
C
G
H
I

포리스트 -> 이진트리 예

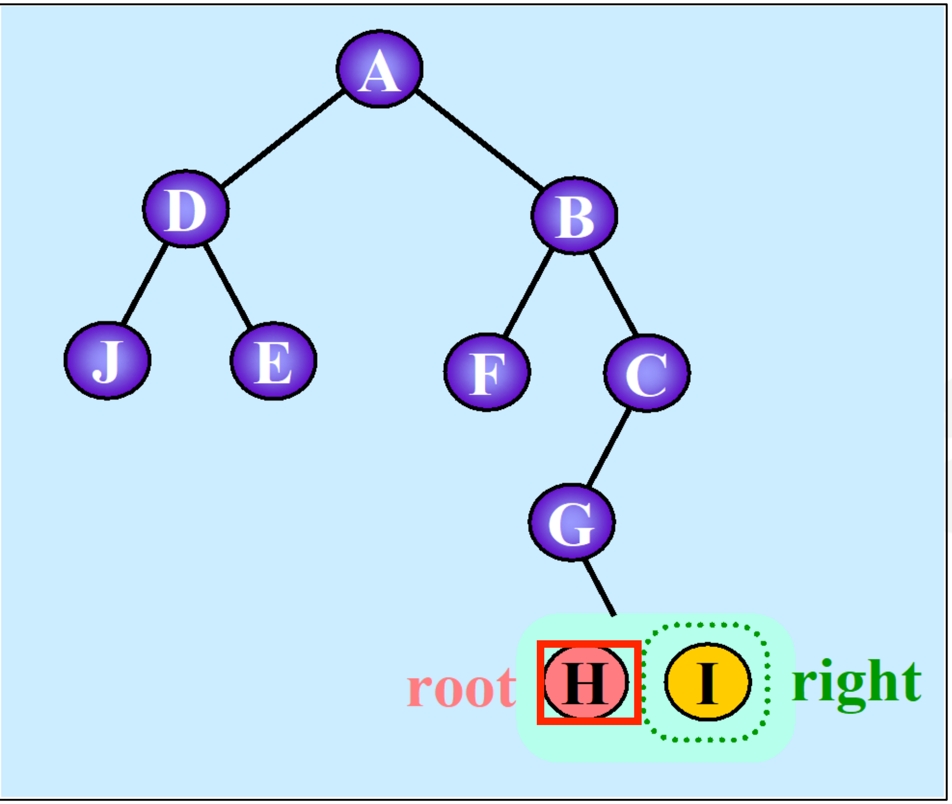


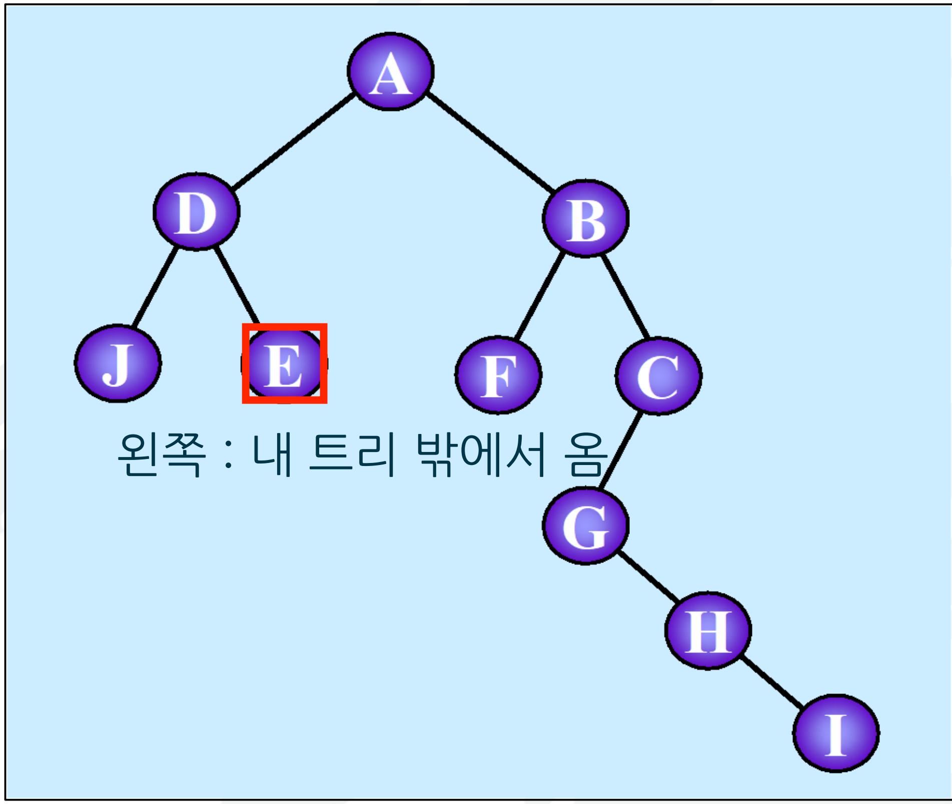
포리스트 -> 이진트리 예





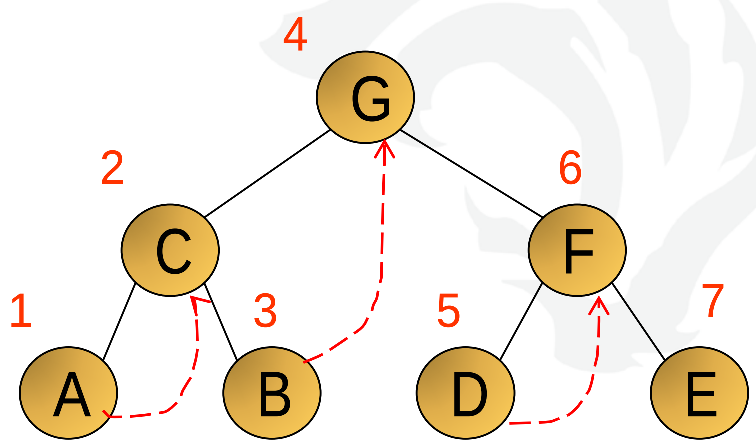
포리스트 -> 이진트리 예





스레드 이진 트리(1)

- 이진트리의 NULL 링크를 이용하여 순환 호출 없이도 트리의 노드들을 순회하는 것이 가능할 수 있음
- 스레드 이진 트리(threaded binary tree)
  - NULL 링크에 중위 순회시에 선행 노드인 중위 선행자(inorder predecessor)나 후속 노드인 중위 후속자(inorder successor)를 저장시켜 놓은 트리



스레드 이진 트리(2)

- 이진 트리의 2n개 링크 중 (n+1)개는 NULL
  - 노드 수: n, 각 노드별 링크 수: 2
  - 전체 링크 수 : n * 2
  - 노드 n 개를 잇는 링크 수: n-1
  - NULL 링크 수: 2n– (n-1) = n+1
- NULL 링크를 스레드(thread, 실) 포인터로 사용하여 스택 없이 중순위 운행이 가능함

스레드 이진 트리 구현(1)

- 스레드 이진 트리를 위한 노드 구조

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

- NULL 링크에 스레드가 저장되면 링크에 자식을 가리키는 포인터가 저장되어 있는지 스레드가 저장되어있는 지 구분 필요
  - is_thread 필드 필요

스레드 이진 트리 구현(2)

- 중위 후속자를 찾는 함수

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

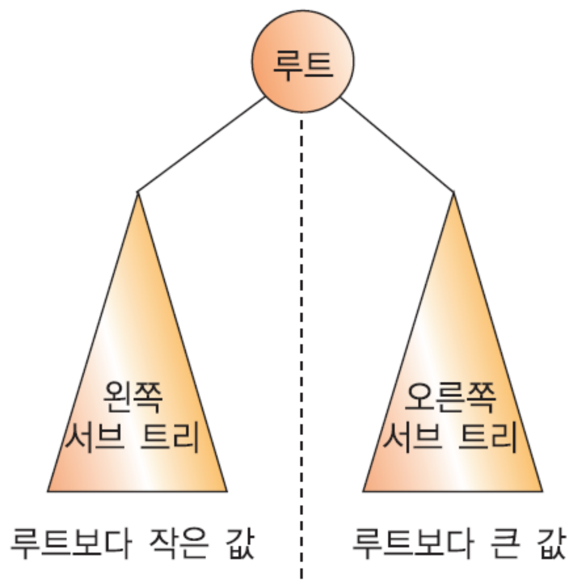
스레드 이진 트리 구현(3)

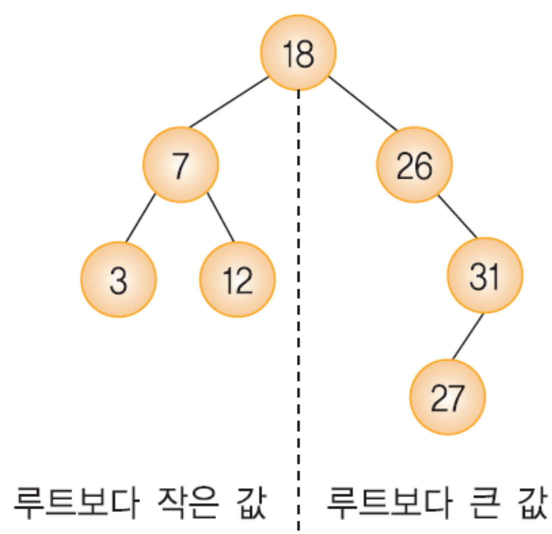
- 스레드 버전 중위 순회 함수
  - 중위 순회는 가장 왼쪽 노드부터 시작하기 때문에, 왼쪽 자식이 NULL이 될때 까지 왼쪽 링크를 타고 이동함
  - 데이터를 출력함
  - 중위 후속자를 찾는 함수를 호출하여 후속자가 NULL이 아니면 계속 루프를 돔

```
void thread_inorder(TreeNode *t)
{
	TreeNode *q;
	q=t;
	while (q->left != NULL) q = q->left;// 가장 왼쪽 노드로 간다.
	do
	{
		printf("%c ", q->data);// 데이터 출력
		q = find_successor(q); // 후속자 함수 호출
	} while(q!=NULL);			// NULL이 아니면
}
```

이진 탐색 트리(Binary Search Tree)

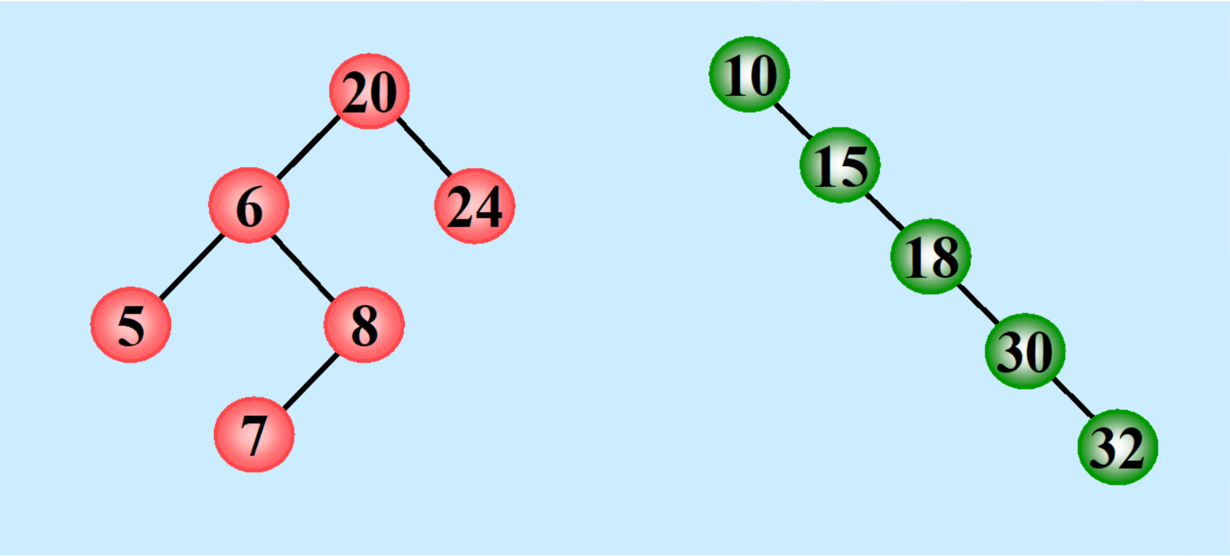
- 탐색작업을 효율적으로 하기 위한 자료구조
- key(왼쪽서브트리)≤key(루트노드)≤key(오른쪽서브트리)
  - 모든 노드의 Key는 유일함
- 이진탐색를 중위순회하면 오름차순으로 정렬된 값을 얻을 수 있음





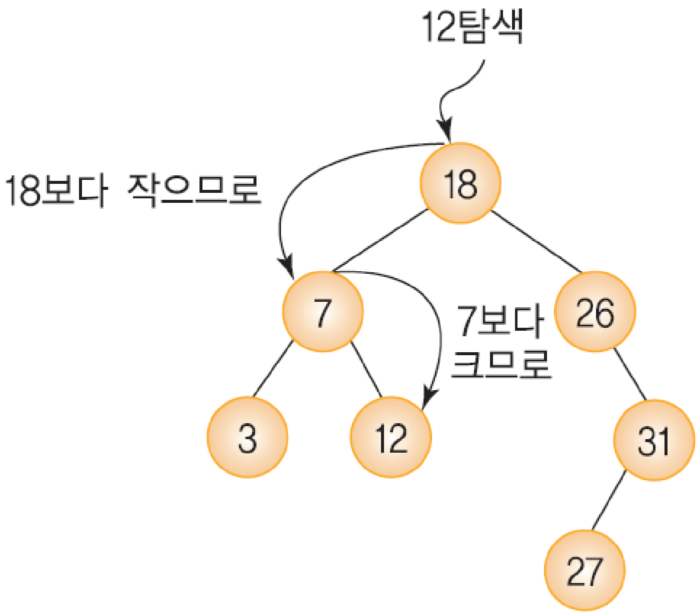
이진 탐색 트리 정의

- 이진 탐색 트리: 공집합이거나 다음을 만족하는 이진 트리
  - 공집합이 아닌 왼쪽 부분 트리의 모든 키값은 루트의 키값 보다 작음
  - 공집합이 아닌 오른쪽 부분 트리의 모든 키값은 루트의 키값보다 큼
  - 왼쪽 부분 트리와 오른쪽 부분 트리도 이진 탐색 트리



이진 탐색 트리 탐색 연산(1)

- 아이디어
  - 비교한 결과가 같으면 탐색이 성공적으로 끝남
  - 주어진 키 값이 루트 노드의 키값보다 작으면 탐색은 이 루트 노드의 왼쪽 자식을 기준으로 다시 시작
  - 주어진 키 값이 루트 노드의 키값보다 크면 탐색은 이 루트 노드의 오른쪽 자식을 기준으로 다시 시작



이진 탐색 트리 탐색 연산(2)

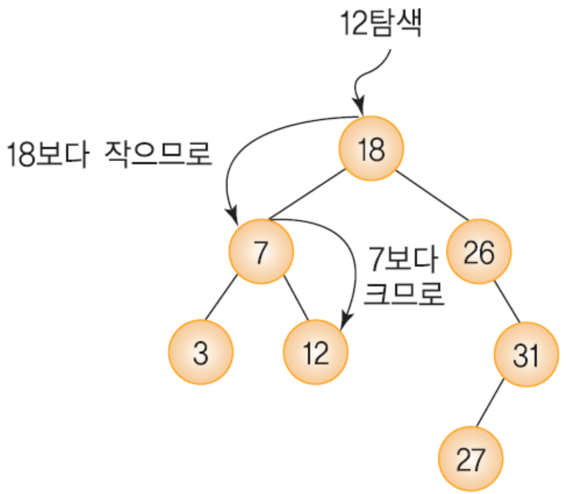
- 알고리즘: 재귀적인 탐색 방법 (찾고자 하는 값 : key)
  - 루트가 null 이면 탐색 실패
  - 루트의 원소값 = key 이면 탐색 성공 & 종료
  - 루트의 원소값 > key 이면 왼쪽 부분 트리를 재귀적으로 탐색
  - 루트의 원소값 < key 이면 오른쪽 부분 트리를 재귀적으로 탐색

- 시간 복잡도 : $O(h)$, 단 h: 트리의 높이

이진 탐색 트리 탐색 연산 알고리즘

```
search(x, k)

if x==NULL
        then return NULL;
if k==x->key
        then return x;
else if k < x->key
     then return search(x->left, k);
     else return search(x->right, k);
```



이진 탐색 트리 탐색 연산 구현(1)

```
//순환적인 탐색 함수

TreeNode  *search(TreeNode  *node,  int  key)
{
    if ( node == NULL )  return NULL;
    if ( key == node->key ) return node;
    else if ( key < node->key )
        return  search(node->left, key);
    else
        return  search(node->right, key);
}
```

recursive

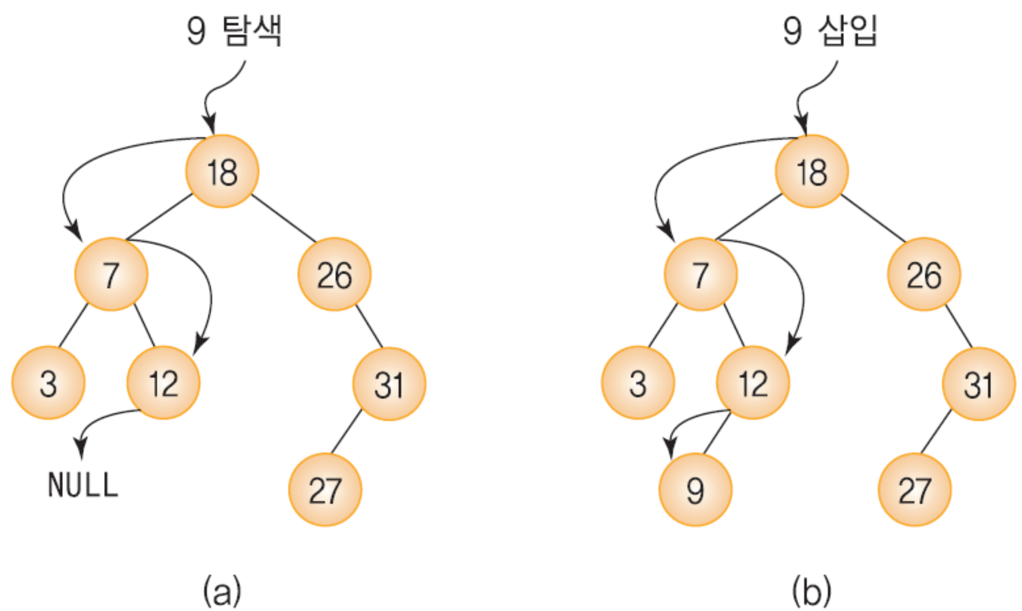
이진 탐색 트리 탐색 연산 구현(2)

```
// 반복적인 탐색 함수

TreeNode *search(TreeNode *node, int key)
{
	while(node != NULL){
		if( key == node->key ) return node;
		else if( key < node->key )
			node = node->left;
		else
			node = node->right;
	}
	return NULL;   // 탐색에 실패했을 경우 NULL 반환
}
```

이진 탐색 트리의 삽입 연산(1)

- 이진 탐색 트리에 원소를 삽입하기 위해서는 먼저 탐색을 수행하는 것이 필요
- 탐색에 실패한 위치가 바로 새로운 노드를 삽입하는 위치

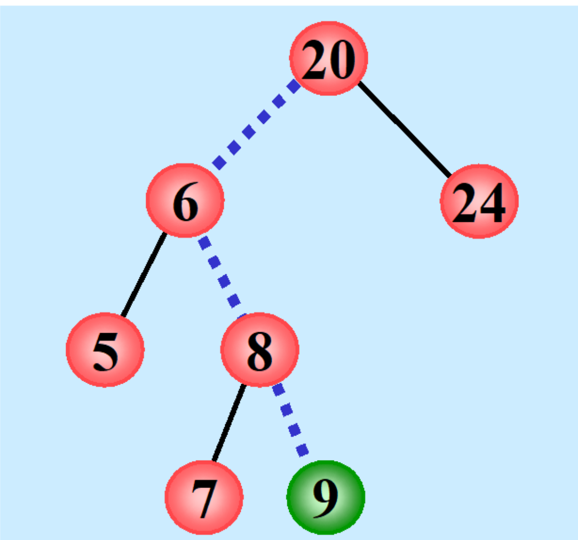


이진 탐색 트리의 삽입 연산(2)

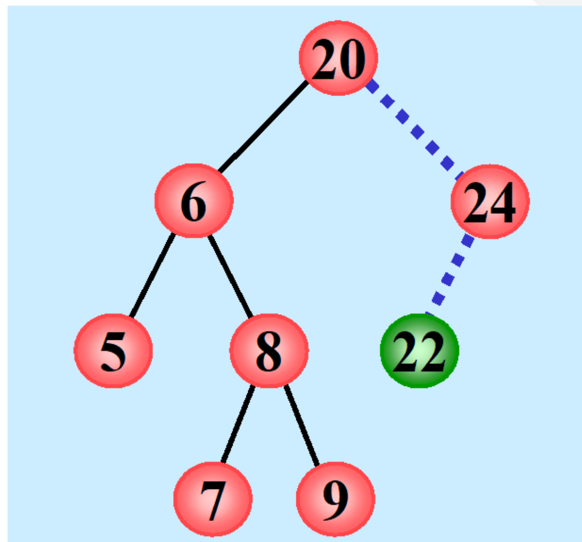
- 삽입(insert) 방법(삽입되는 원소값 : key)
  - key를 탐색
  - 탐색이 실패한 위치에 새로운 노드 삽입
- 시간 복잡도 : $O(h)$, 단 h : 트리의 높이

이진 탐색 트리의 삽입 연산 예

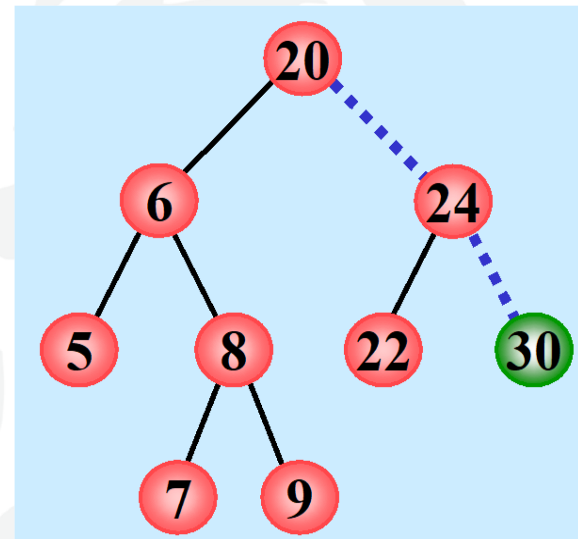
9 삽입



22 삽입



30 삽입



이진 탐색 트리의 삽입 연산 알고리즘(1)

- 경우 1
  - 비어 있는 트리에 노드를 삽입
- 경우 2
  - 삽입하고자 하는 key가 트리에 이미 존재
- 경우 3
  - 3-1: 새로운 노드를 기존 노드의 왼쪽 자식 링크에 삽입하는 경우
  - 3-2: 새로운 노드를 기존 노드의 오른쪽 자식 링크에 삽입하는 경우

이진 탐색 트리의 삽입 연산 알고리즘(2)

```
insert_node(T, key)

p←NULL;  //p: 부모노드 포인터
T←T;       //t: 탐색을 위한 포인터
While t≠NULL do   //탐색을 수행함
  p←t;  //현재 탐색 포인터 값을 부모 노드 포인터에 복사
  if    key < p->key
     then t←p->left;
     else t←p->right;
  z ← make_node(key);

// 삽입
if p=NULL
  then  T←z;                 // 트리가 비어있음
else if   key < p->key
  then p->left←z
  else p->right←z
```

이진 탐색 트리의 삽입 연산 구현(1)

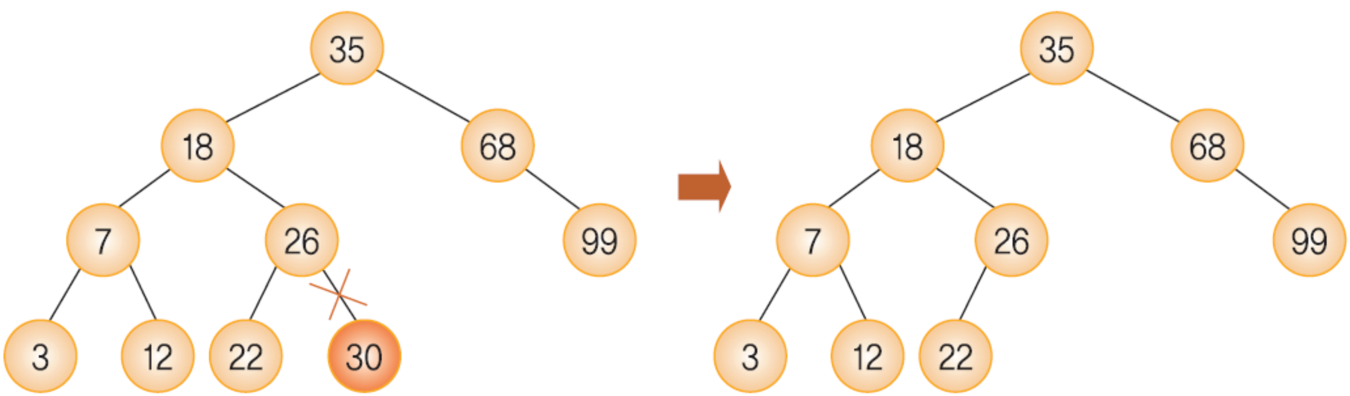
```
// key를 이진 탐색 트리 root에 삽입한다.
// key가 이미 root안에 있으면 삽입되지 않는다.
void insert_node(TreeNode **root, int key)
{
        TreeNode *p, *t; // p는 부모노드, t는 현재노드
        TreeNode *n;     // n은 새로운 노드
        t = *root;
        p = NULL;
        // 탐색을 먼저 수행
        while (t != NULL){
           if( key == t->key ) return;
           p = t;
           if( key < t->key ) t = p->left;
           else t = p->right;
        }
```

이진 탐색 트리의 삽입 연산 구현(2)

```
    // key가 트리 안에 없으므로 삽입 가능
    // 트리노드 구성
     n = (TreeNode *) malloc(sizeof(TreeNode));
     if( n == NULL ) return;
     // 데이터 복사
     n->key = key;
     n->left = n->right = NULL;
     // 부모 노드와 링크 연결
     if( p != NULL )
          if( key < p->key )
               p->left = n;
          else p->right = n;
     else *root = n;
}
```

이진 탐색 트리 삭제 연산(1)

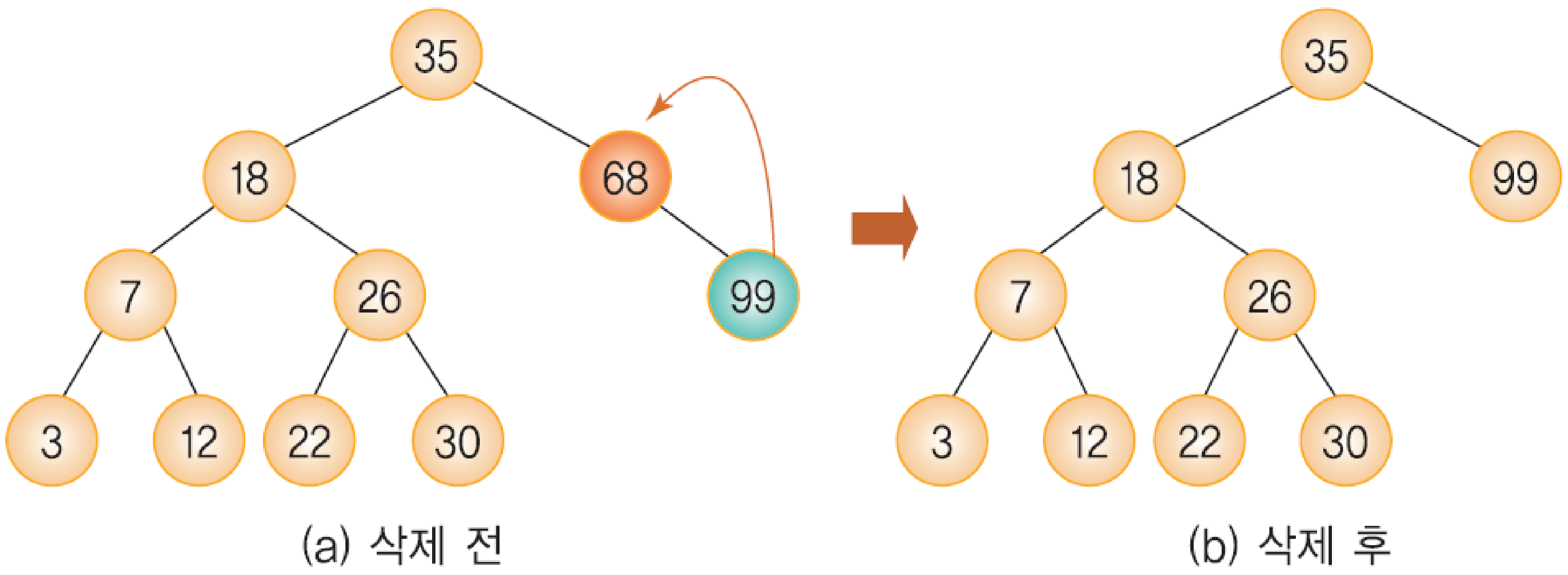
- 3가지의 경우
  - 1. 삭제하려는 노드가 단말 노드 일 경우
  - 2. 삭제하려는 노드가 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우
  - 3. 삭제하려는 노드가 두개의 서브 트리 모두 가지고 있는 경우
- CASE 1: 삭제하려는 노드가 단말 노드일 경우
  - 단말노드의 부모노드를 찾아서 연결을 끊음



(a) 삭제 전 (b) 삭제 후

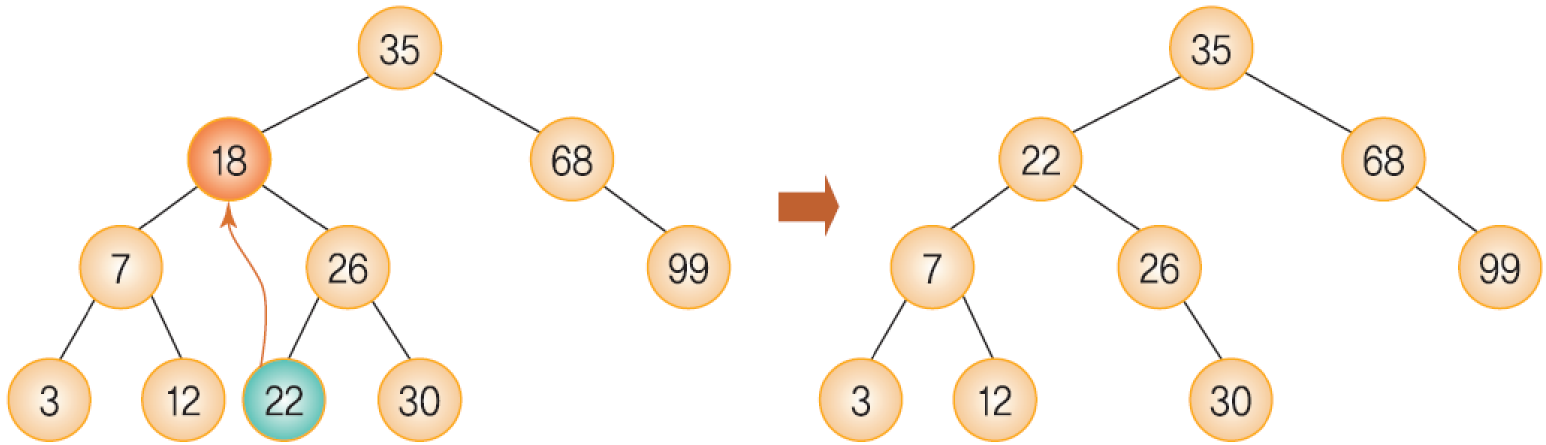
이진 탐색 트리 삭제 연산(2)

- CASE 2: 삭제하려는 노드가 하나의 서브 트리만 갖고 있는 경우
  - 삭제되는 노드가 왼쪽이나 오른쪽 서브 트리중 하나만 갖고 있을 때, 그 노드는 삭제하고 서브 트리는 부모 노드에 붙여줌



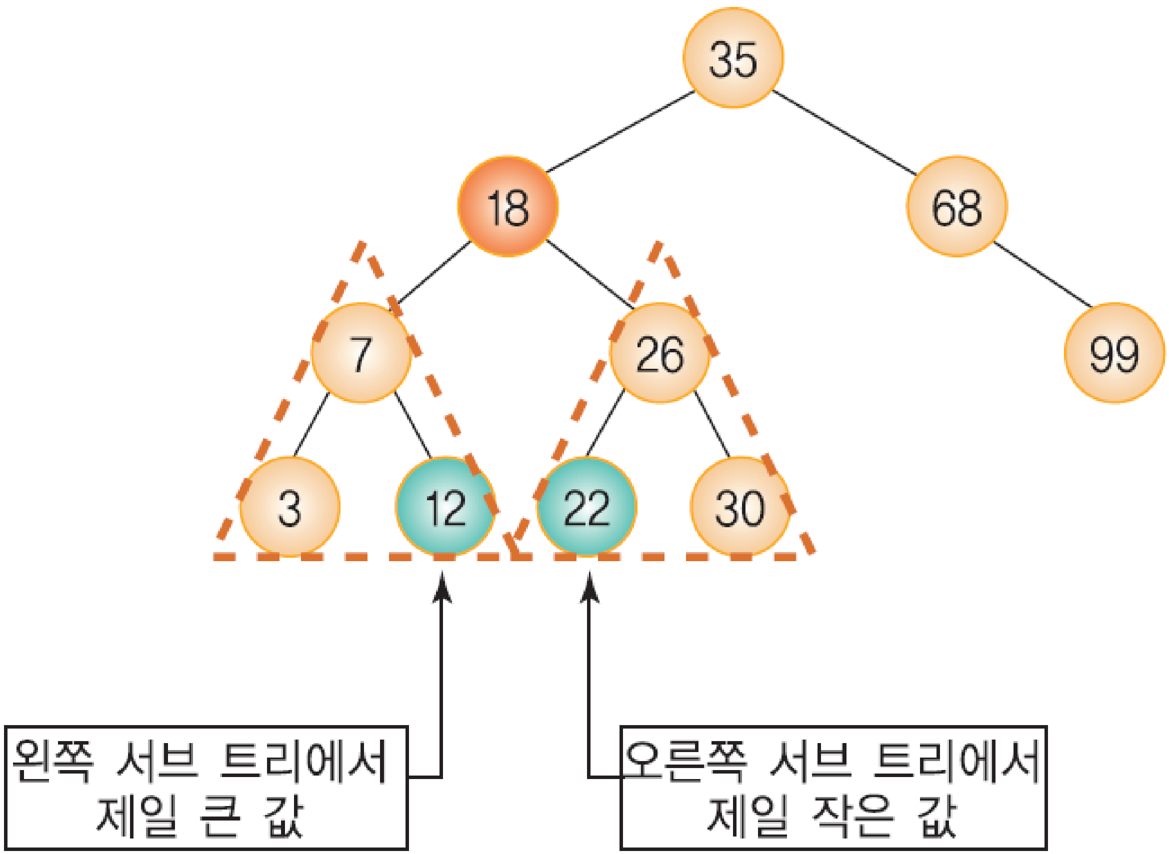
이진 탐색 트리 삭제 연산(3)

- CASE 3: 삭제하려는 노드가 두개의 서브트리를 갖고 있는 경우
  - 삭제노드와 가장 비슷한 값을 가진 노드를 삭제노드 위치로 가져옴



이진 탐색 트리 삭제 연산(4)

- Case 3에서 가장 비슷한 값은 어떻게 찾을까?



이진 탐색 트리 삭제 연산 구현(1)

```
// 삭제 함수
void delete_node(TreeNode **root, int key)
{
TreeNode *p, *child, *succ, *succ_p, *t;
// key를 갖는 노드 t를 탐색, p는 t의 부모노드
p = NULL;
t = *root;
// key를 갖는 노드 t를 탐색한다.
while( t != NULL && t->key != key ){
        p = t;
        t = ( key < p->key ) ? p->left : p->right;
}
// 탐색이 종료된 시점에 t가 NULL이면 트리안에 key가 없음
if( t == NULL ) {    // 탐색트리에 없는 키
        printf("key is not in the tree");
        return;
}
```

이진 탐색 트리 삭제 연산 구현(2)

```
// 첫번째 경우: 단말노드인 경우
if( (t->left==NULL) && (t->right==NULL) ){
        if( p != NULL ){
                // 부모노드의 자식필드를 NULL로 만든다.
                if( p->left == t )
                        p->left = NULL;
                else p->right = NULL;
        }
        else    // 만약 부모노드가 NULL이면 삭제되는 노드가 루트
                *root = NULL;
}
```

이진 탐색 트리 삭제 연산 구현(3)

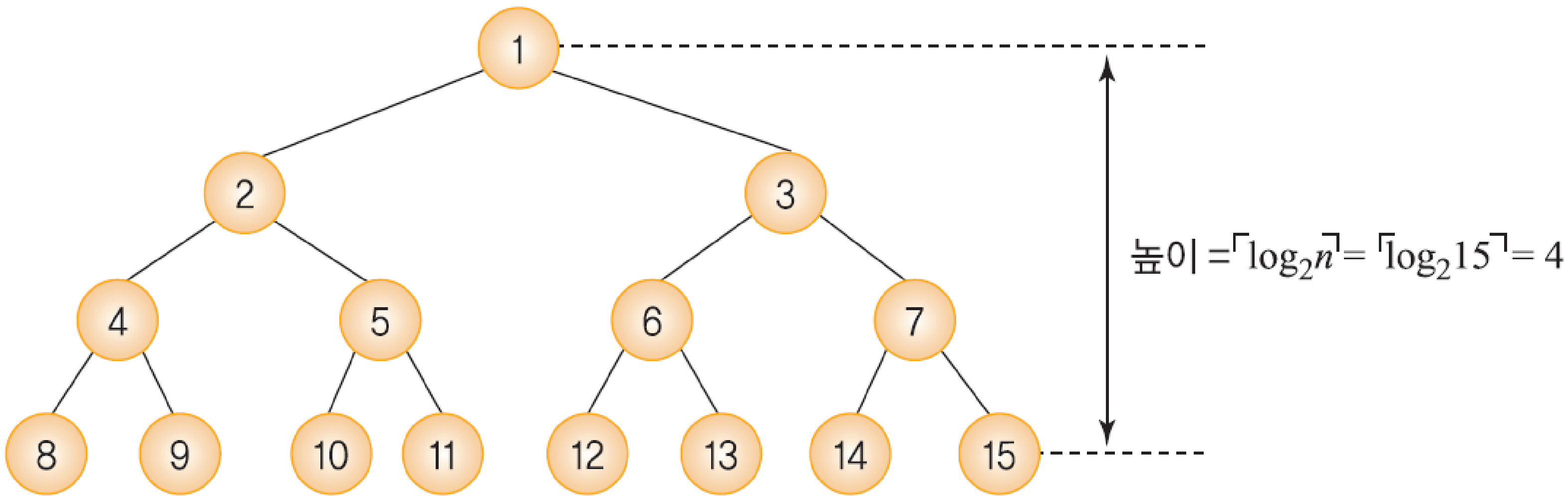
```
// 두번째 경우: 하나의 자식만 가지는 경우
else if((t->left==NULL)||(t->right==NULL)){
        child = (t->left != NULL) ? t->left : t->right;
        if( p != NULL ){
                if( p->left == t )        // 부모를 자식과 연결
                        p->left = child;
                else p->right = child;
        }
        else // 만약 부모노드가 NULL이면 삭제되는 노드가 루트
                *root = child;
}
```

이진 탐색 트리 삭제 연산 구현(4)

```
// 세번째 경우: 두개의 자식을 가지는 경우
else{
        // 오른쪽 서브트리에서 후계자를 찾는다.
        succ_p = t;
        succ = t->right;
        // 후계자를 찾아서 계속 왼쪽으로 이동한다.
        while(succ->left != NULL){
                succ_p = succ;
                succ = succ->left;
        }
        // 후속자의 부모와 자식을 연결
        if( succ_p->left == succ )
                succ_p->left = succ->right;
        else
                succ_p->right = succ->right;
        // 후속자가 가진 키값을 현재 노드에 복사
        t->key = succ->key;
        // 원래의 후속자 삭제
        t = succ;
}
free(t);
}
```

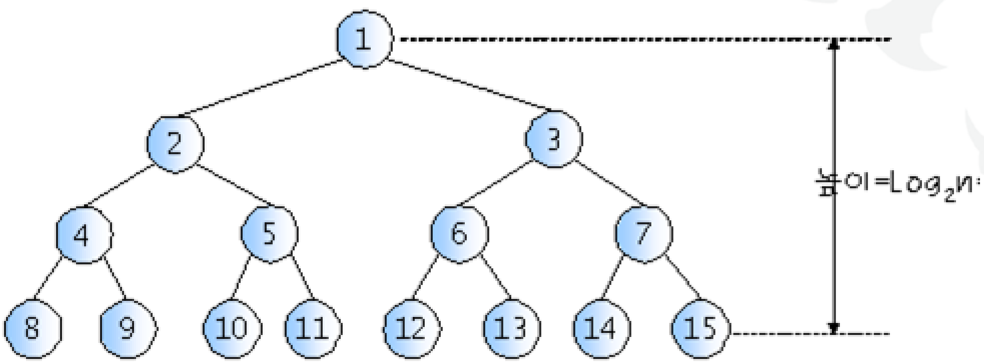
이진 탐색 트리 성능 분석(1)

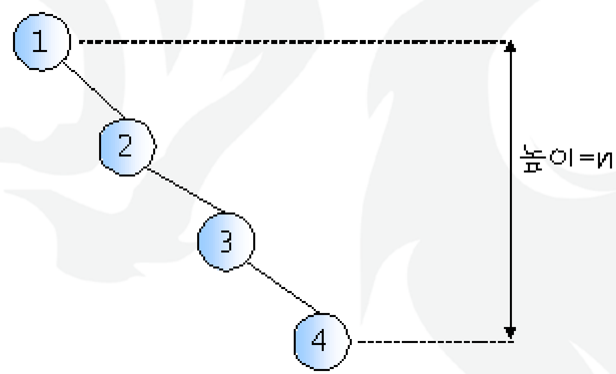
- 이진 탐색 트리에서의 탐색, 삽입, 삭제 연산의 시간 복잡도는 트리의 높이를 h라고 했을 때 O(h)
- 일반적인 이진 탐색 트리의 높이



이진 탐색 트리 성능 분석(2)

- 최선의 경우
  - 이진 트리가 균형적으로 생성되어 있는 경우
  - $h=\log_2 n$
- 최악의 경우
  - 한쪽으로 치우친 경사이진트리의 경우
  - $h=n$
  - 순차탐색과 시간복잡도가 같음





이진 탐색 트리 성능 분석(3)

- 이진 탐색 트리의 최대 높이
  - n개 노드에 대하여 최대 $n-1$ (루트의 레벨을 0으로 하는 경우)
  - 1,2,…,n 순으로 삽입되거나 n,$n-1$,…,2,1 순으로 삽입되어 사향트리를 생성하는 경우
  - 탐색, 삽입, 제거 연산의 최악의 시간 복잡도
    - $T(n) = O(h) \Rightarrow n$ 에 대한 식으로 표기하면, $T(n) = O(n)$
  - 효율성 측면에서 최악의 경우 $O(\log_2 n)$이 바람직 ⇒ AVL 트리, 2-3 트리
    - 균형 잡힌 트리

<< 한쪽으로 치우치지 않은 균형잡힌 트리를 만든다면 안정적인 퍼포먼스를 구현해 낼 수 있다.

- 이진 탐색 트리의 탐색, 삽입, 제거의 평균 시간은 $O(\log_2 n)$으로 증명되어 있음

Week 8: Forest, Binary Search Tree