



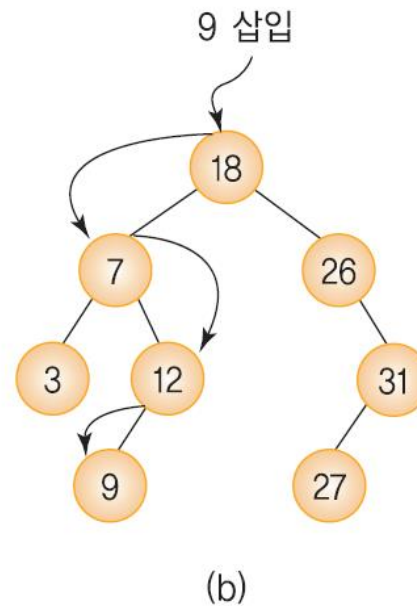
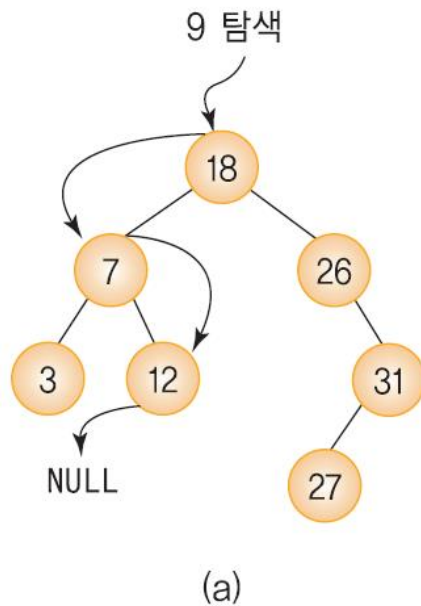
CSE2010 자료구조론

Week 7: Binary Search Tree 2

ICT융합학부 한진영

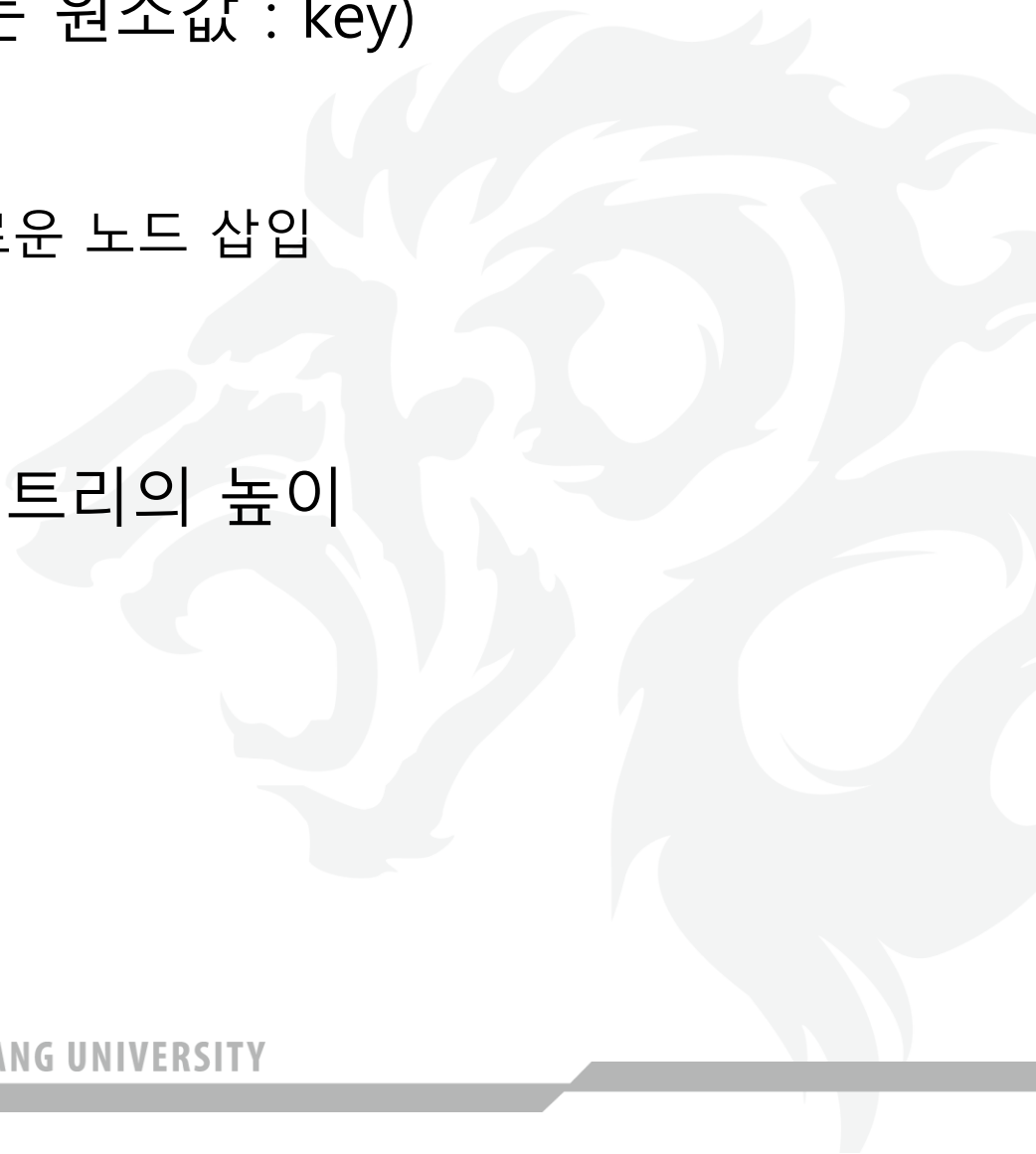
이진 탐색 트리의 삽입 연산(1)

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



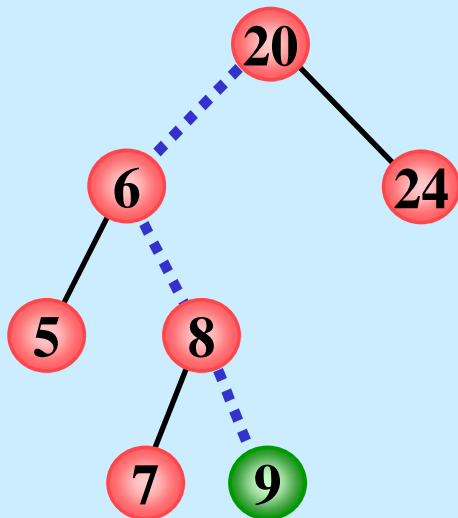
이진 탐색 트리의 삽입 연산(2)

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

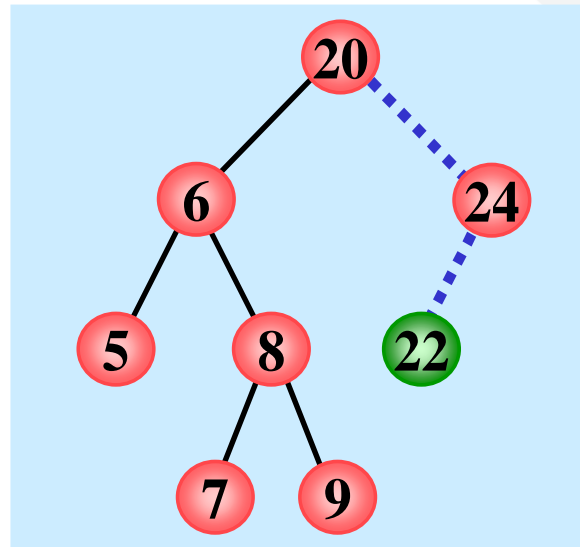


이진 탐색 트리의 삽입 연산 예

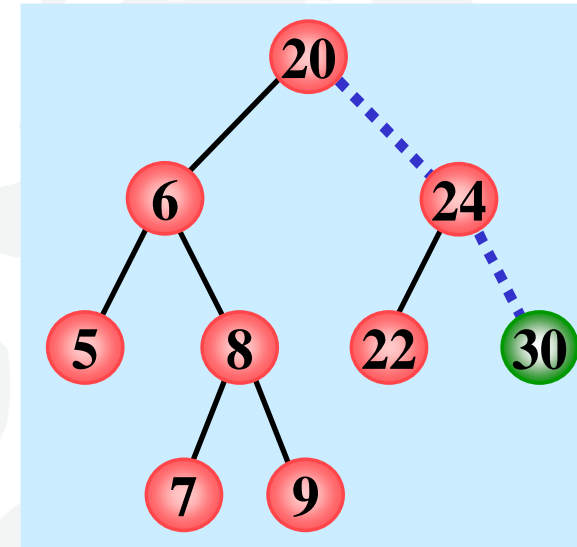
9 삽입



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;
}
```

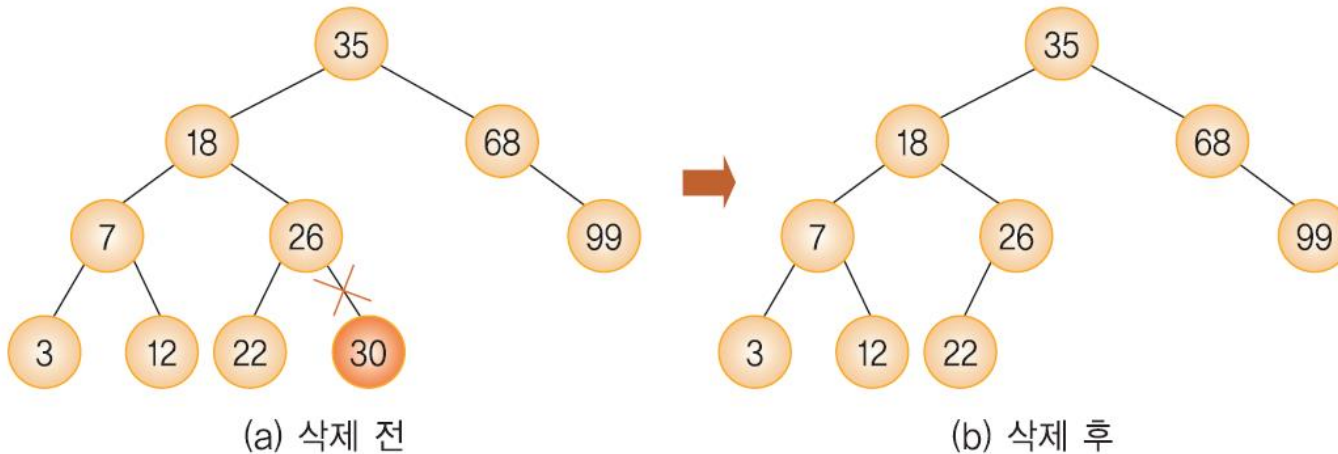

이진 탐색 트리 삭제 연산(1)

■ 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

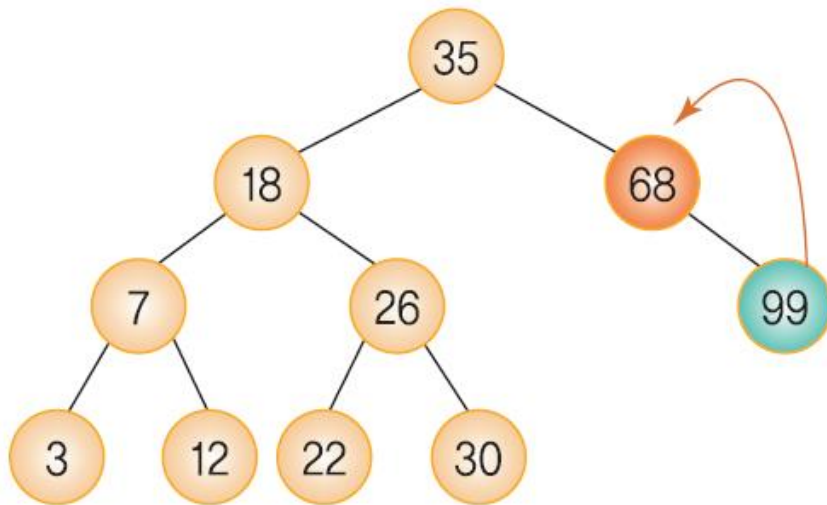
■ CASE 1: 삭제하려는 노드가 단말 노드일 경우

- 단말노드의 부모노드를 찾아서 연결을 끊음

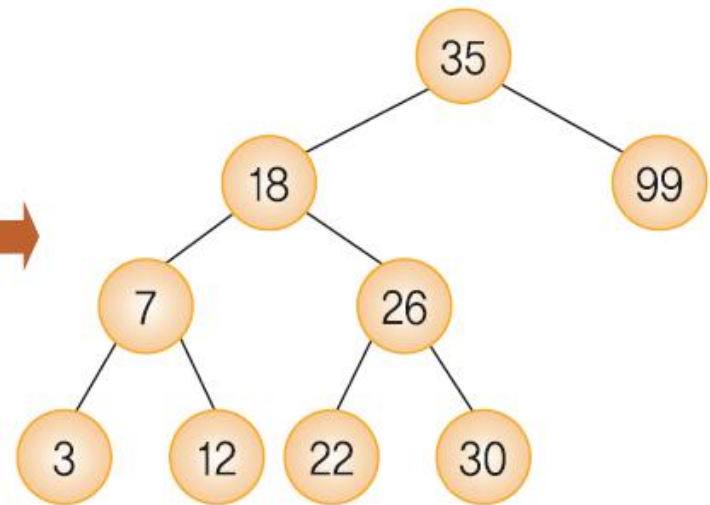


이진 탐색 트리 삭제 연산(2)

- 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



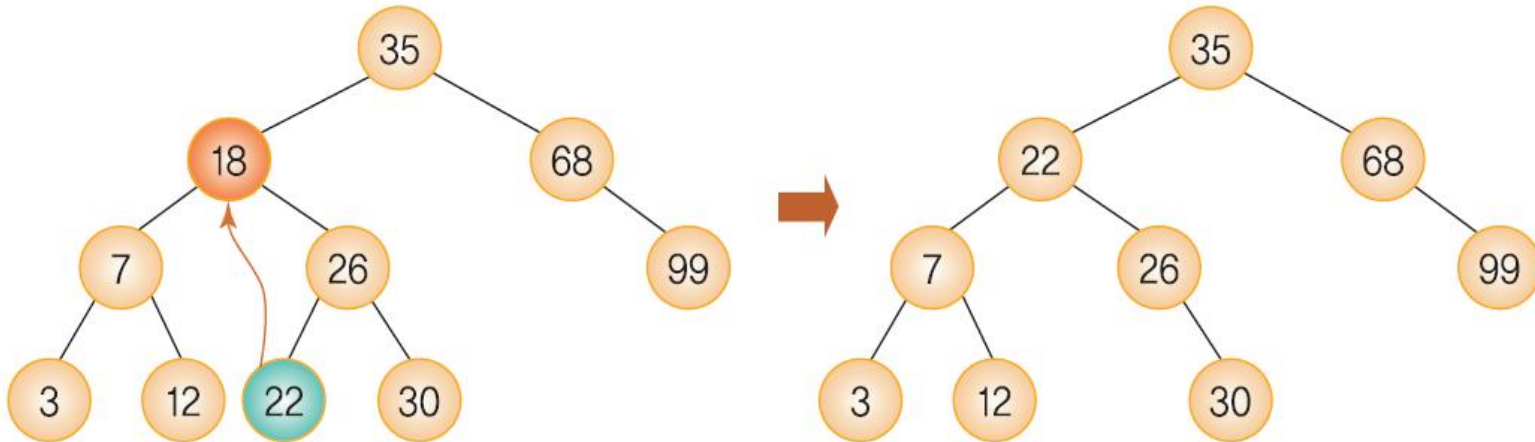
(a) 삭제 전



(b) 삭제 후

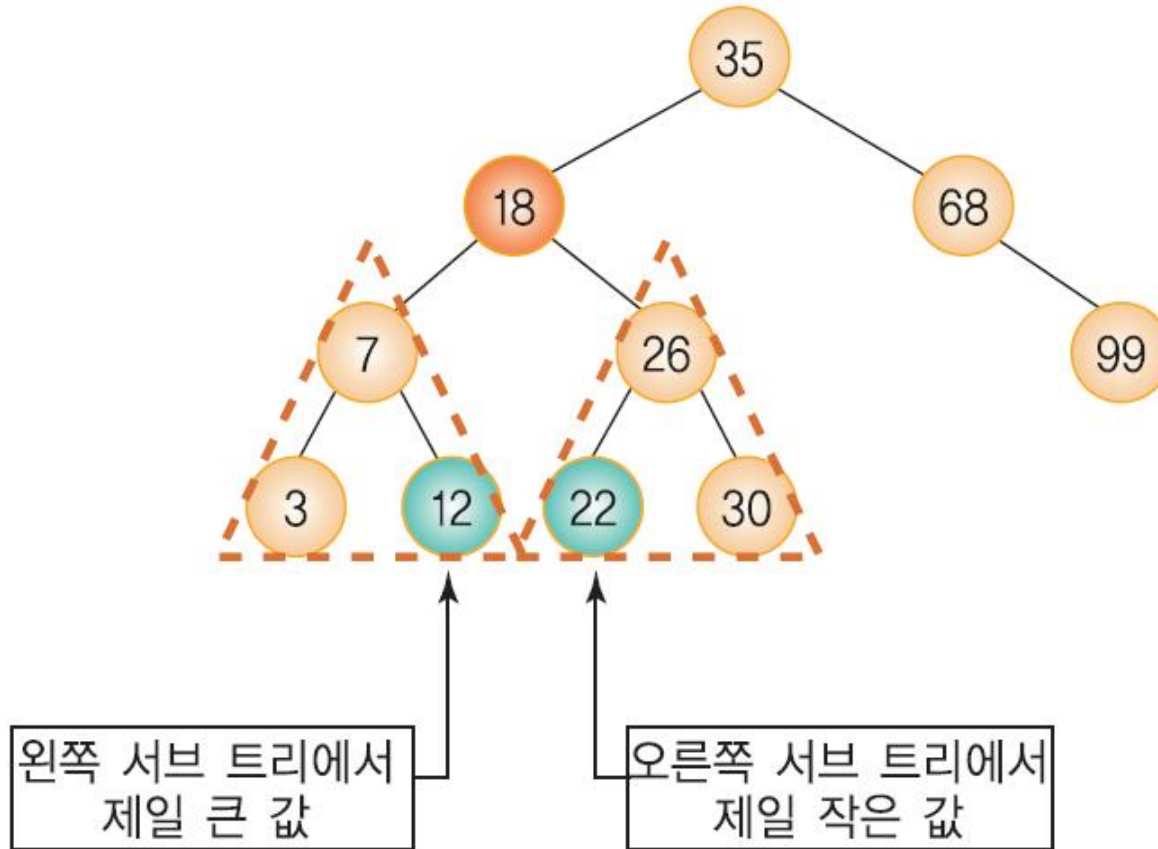
이진 탐색 트리 삭제 연산(3)

- 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)

```
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;

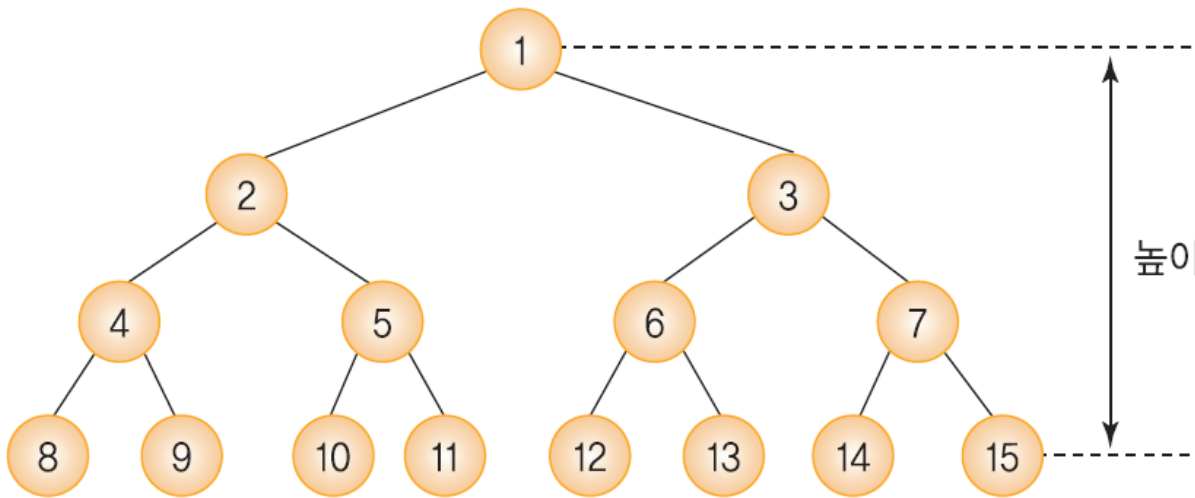
}

free(t);

}
```


이진 탐색 트리 성능 분석(1)

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



$$\text{높이} = \lceil \log_2 n \rceil = \lceil \log_2 15 \rceil = 4$$

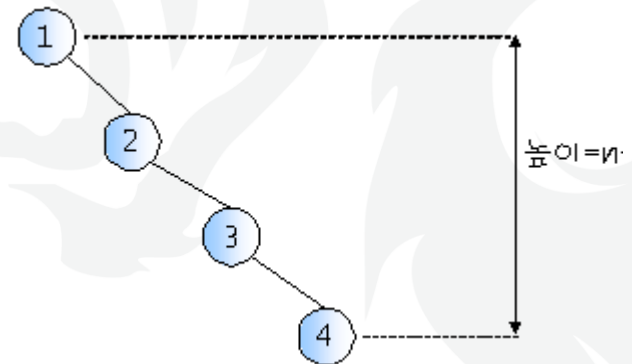
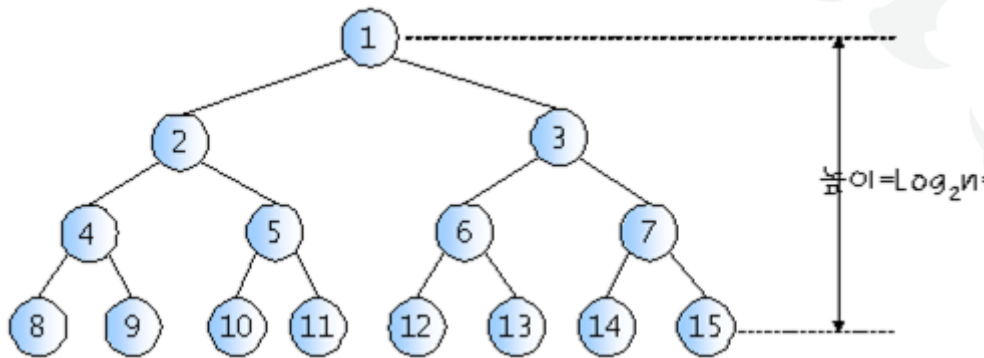
이진 탐색 트리 성능 분석(2)

■ 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
 - n 개 노드에 대하여 최대 $n-1$ (루트의 레벨을 0으로 하는 경우)
 - $1, 2, \dots, n$ 순으로 삽입되거나 $n, n-1, \dots, 2, 1$ 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 \Rightarrow AVL 트리, 2-3 트리
 - 균형 잡힌 트리
- 이진 탐색 트리의 탐색, 삽입, 제거의 평균 시간은 $O(\log_2 n)$ 으로 증명되어 있음

Week 7: Binary Search Tree 2

