

색인 Index

색인(찾아보기)

데이터의 홍수 시대

매일 250경(2.5×10^{18}) bytes 이상의 데이터 생성

1 Tera byte disk 250만개 분량

색인

책의 뒷부분 찾아보기,
웹 사이트, 파일, 레코드, ...



레코드, 키, 검색 트리

레코드 **Record**

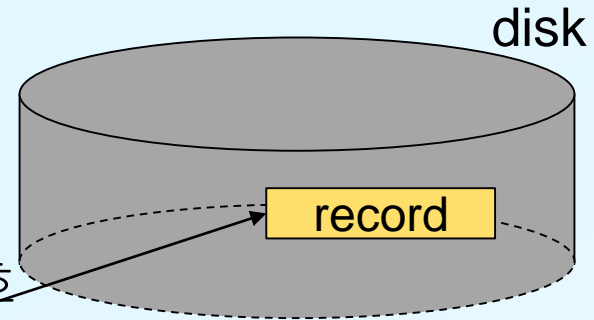
- 개체에 대한 모든 정보 포함
- 예: 사람 레코드
 - 주민번호, 이름, 사번, 집주소, 집전화번호, 직장전화번호, 핸드폰번호, 학력사항, 연소득, 부, 모, 형제자매, ...
 - 이들 각각을 필드 **Field**라 한다

색인의 목적은 개체의 레코드를 검색하는 것

주민번호, 이름, 사번, 집주소, 집전화번호, 직장전화번호,
핸드폰번호, 학력사항, 연소득, 부, 모, 형제자매, ...

색인은 각 레코드를 대표할 수 있는 필드로 만든다

- A field that uniquely identifies a record
- 이런 필드를 키^{Key}라 한다
- 사람의 레코드에서
 - 이름은 key가 될 수 없다
 - key가 될 수 있는 것
 - 학번, 주민번호, 핸드폰번호
 - “생년월일+본적지 주소”(어색하지만 이론상 가능? 쌍둥이는?)



색인의 구성

- (key, page#)
- key + 해당 key를 key로 하는 레코드의 page(=block) 번호

ADT Index

ADT Index

키 x 를 삽입한다

키 x 를 검색한다

키 x 를 삭제한다

색인을 위한 자료구조

수업에서 배울 것

Binary search tree

Balanced binary search tree

- AVL tree, Red-Black tree

B-tree

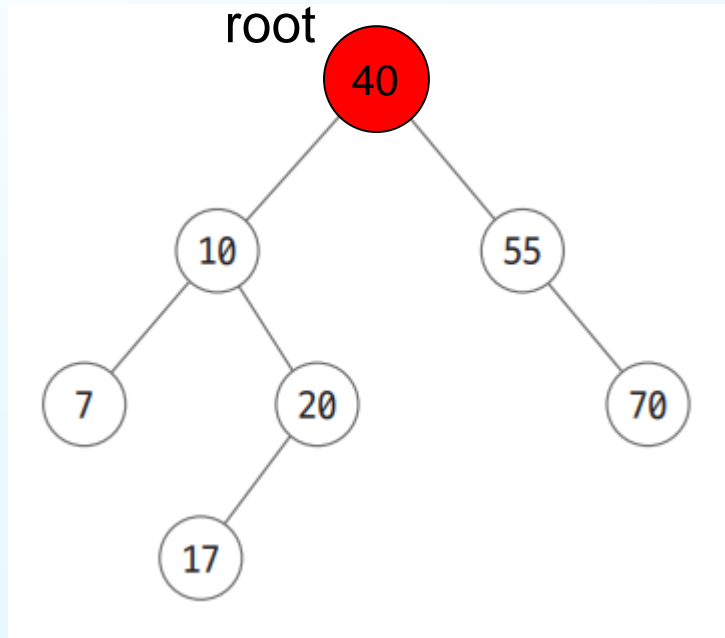
Hash table

- ✓ 배열도 색인으로 쓸 수는 있으나 매우 비효율적,
실용적 매력 없다

(a, p_1)	(b, p_2)			...					(x, p_n)
------------	------------	--	--	-----	--	--	--	--	------------

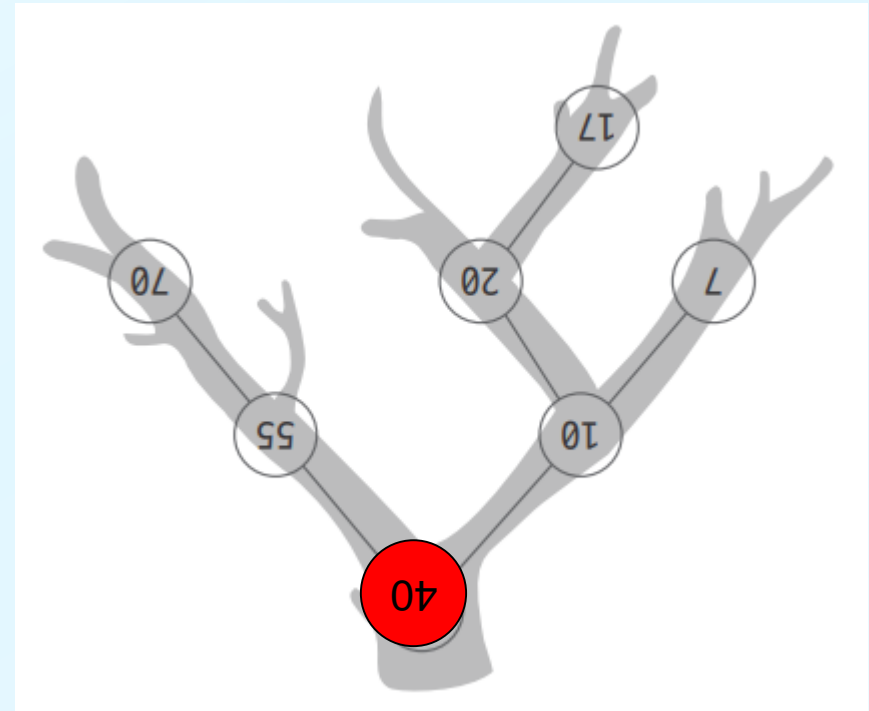
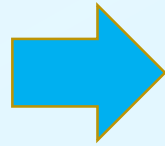
이진검색트리 Binary Search Tree

트리 Tree



트리의 예

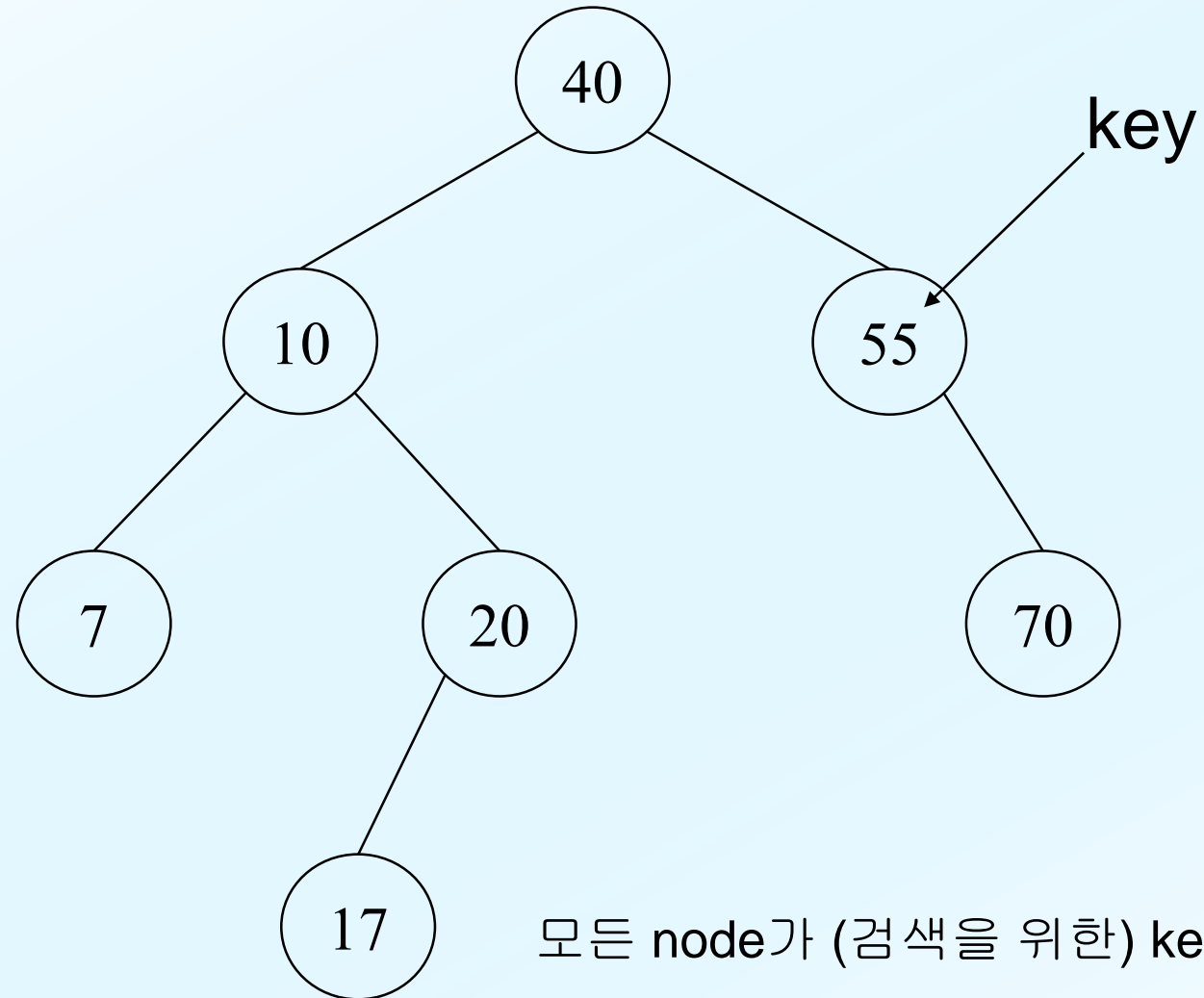
뒤집어 보면



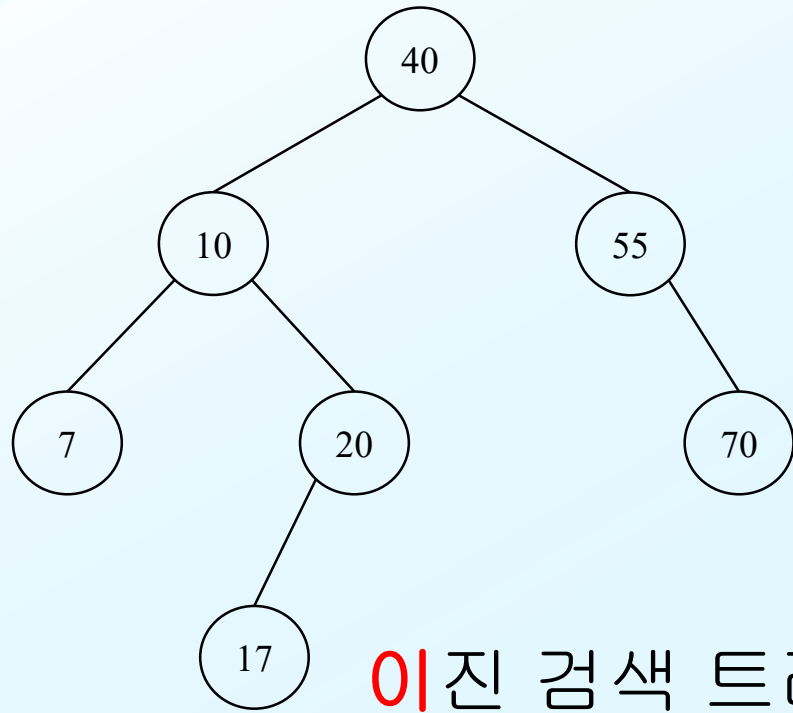
나무 Tree 처럼 보인다

뿌리 root

검색 트리 Search Tree

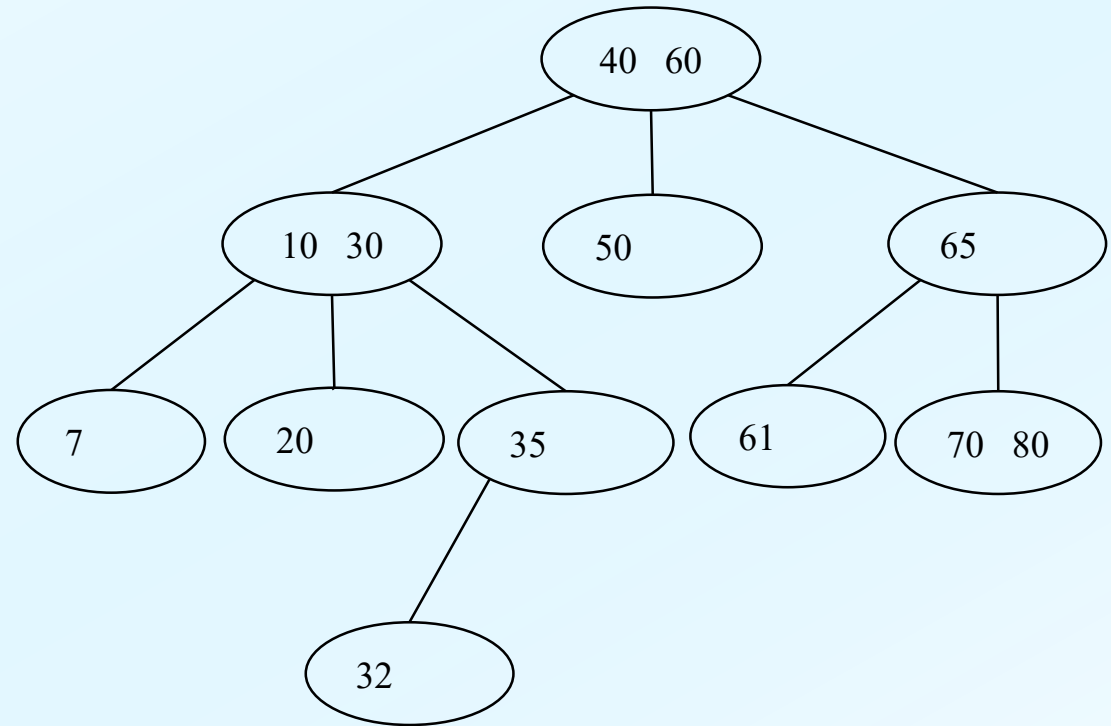


K-진 검색 트리



이진 검색 트리

K=2, 최대 2개 분기

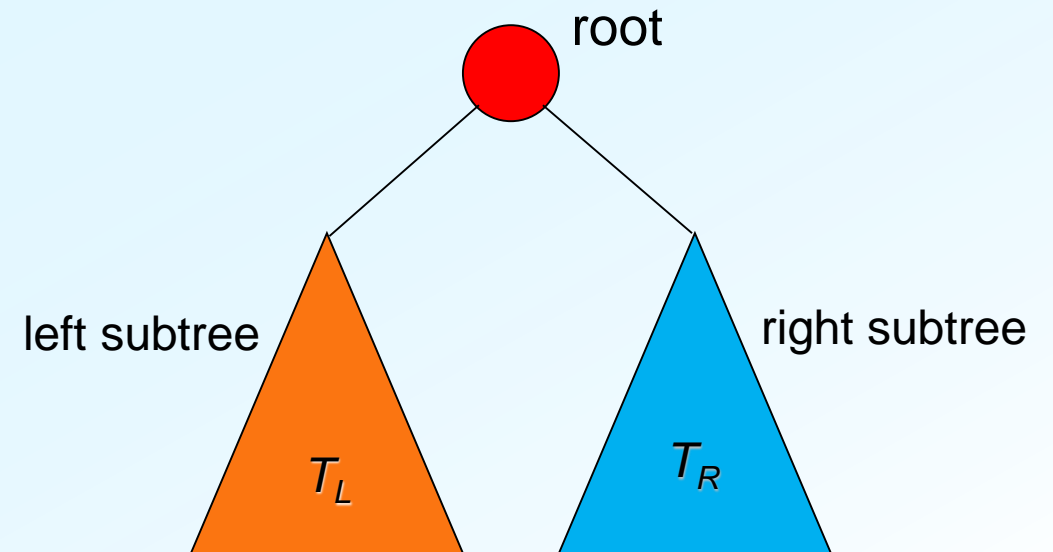


삼진 검색 트리

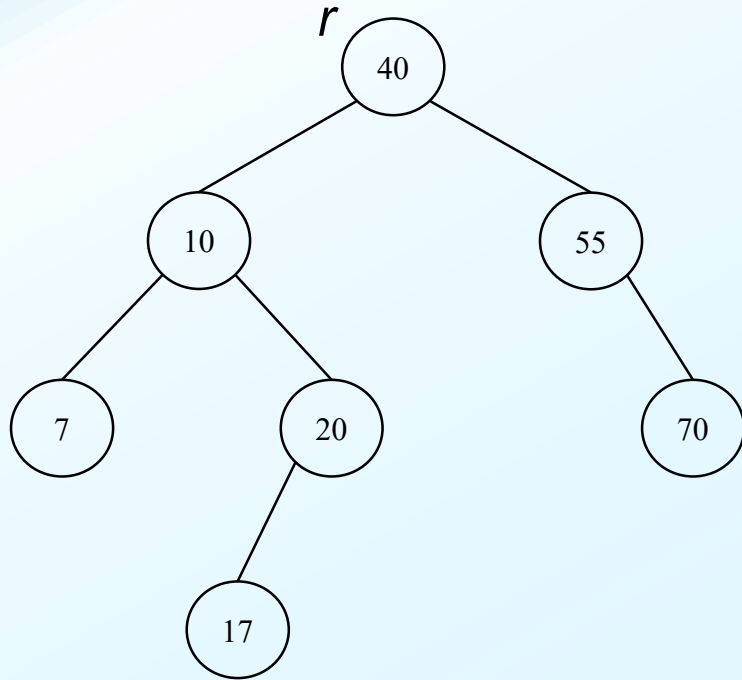
K=3, 최대 3개 분기

이진 검색 트리 binary Search Tree

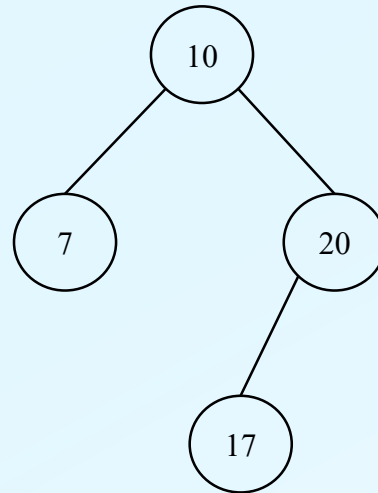
- ① 노드는 모두 서로 다른 키를 갖는다
- ② 노드 중 하나는 뿌리 노드 **Root Node**다
- ③ 각 노드는 최대 2개의 자식을 갖는다(최대 두 분기 **branch**)
- ④ 임의의 노드의 키 **key**는
자신의 좌서브트리 **left subtree**에 있는 모든 노드의 키보다 크고,
자신의 우서브트리 **right subtree**에 있는 모든 노드의 키보다 작다



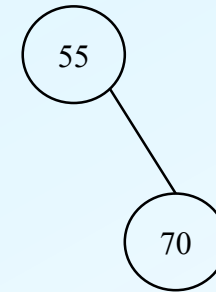
서브트리 Subtree



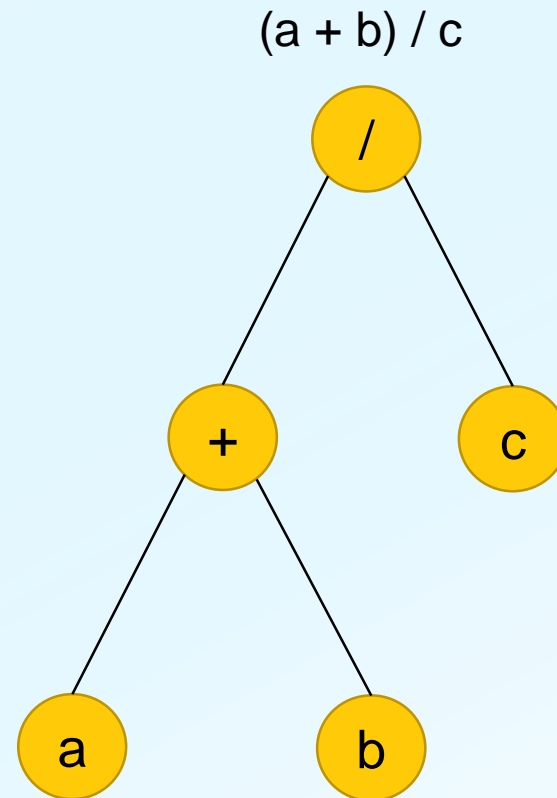
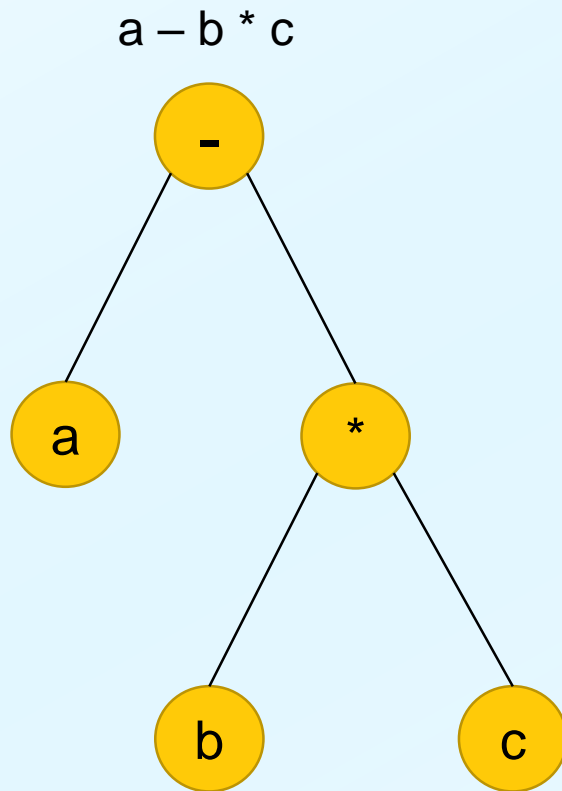
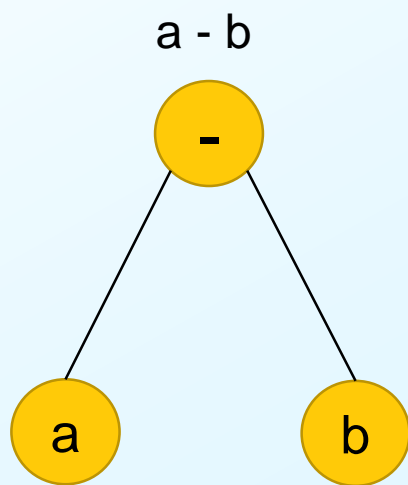
노드 r 의 좌서브트리



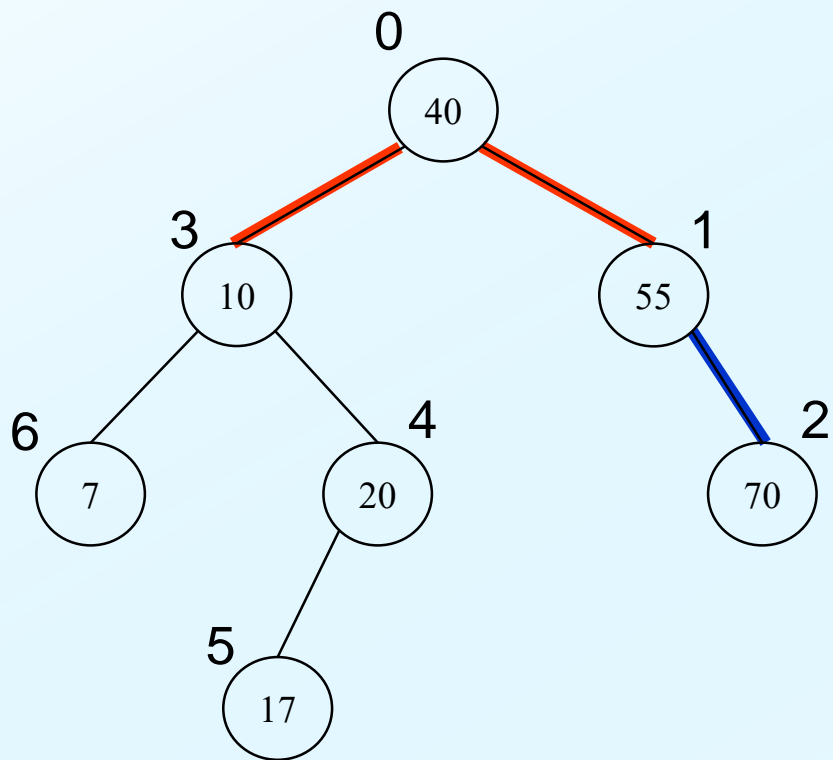
노드 r 의 우서브트리



이진 트리로 수식을 표현할 수 있다



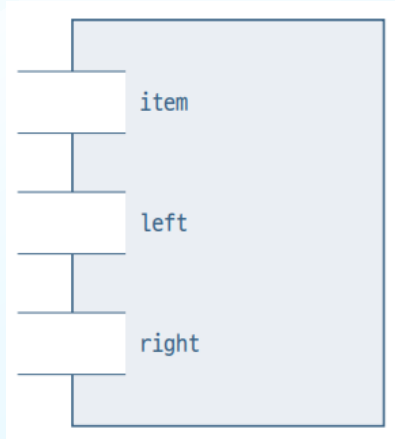
배열 기반의 이진 검색 트리 표현



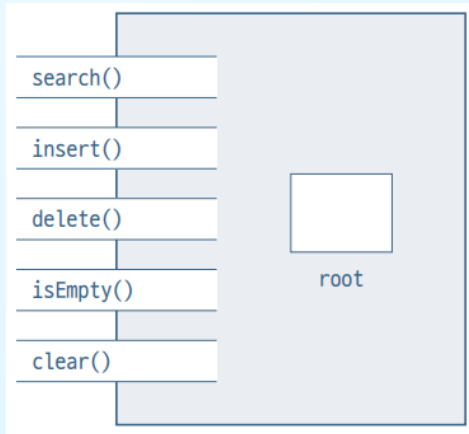
	key	leftChild	rightChild
0	40	3	1
1	55	-1	2
2	70	-1	-1
3	10	6	4
4	20	5	-1
5	17	-1	-1
6	7	-1	-1
7	?		
8	?		
...

2차원 배열

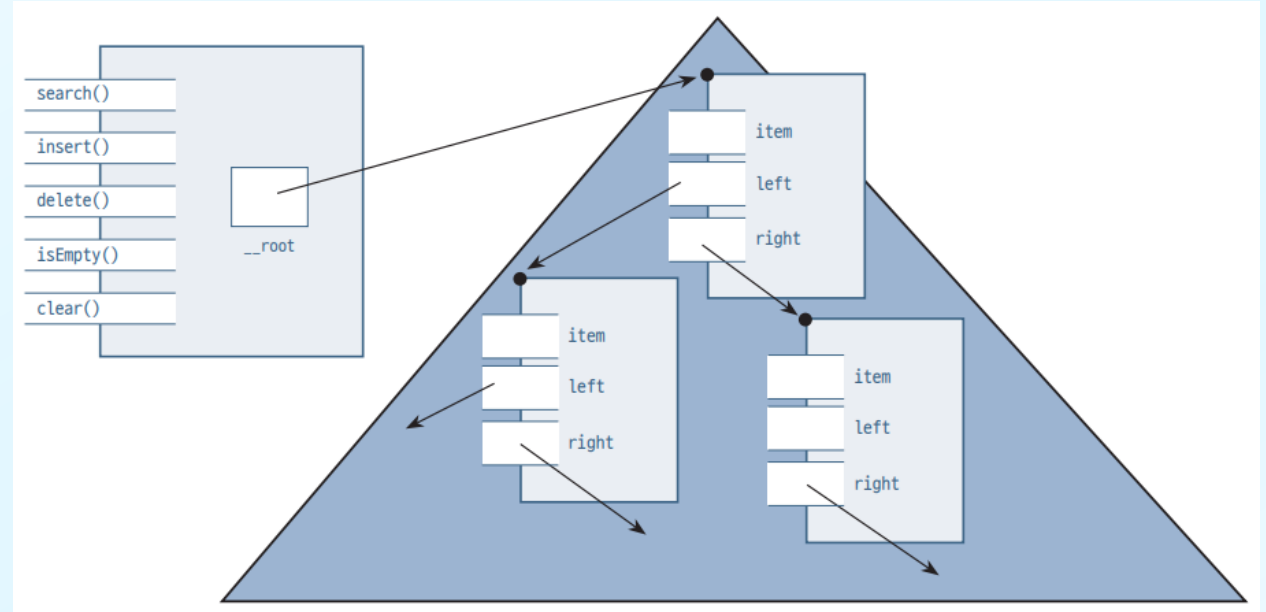
링크 기반의 이진 검색 트리 표현



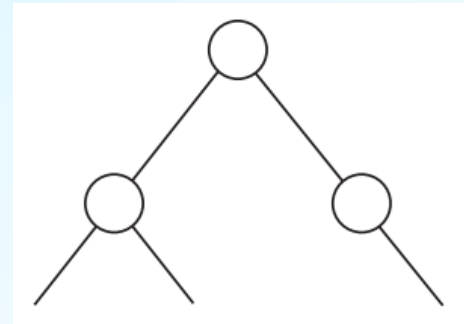
노드 구조



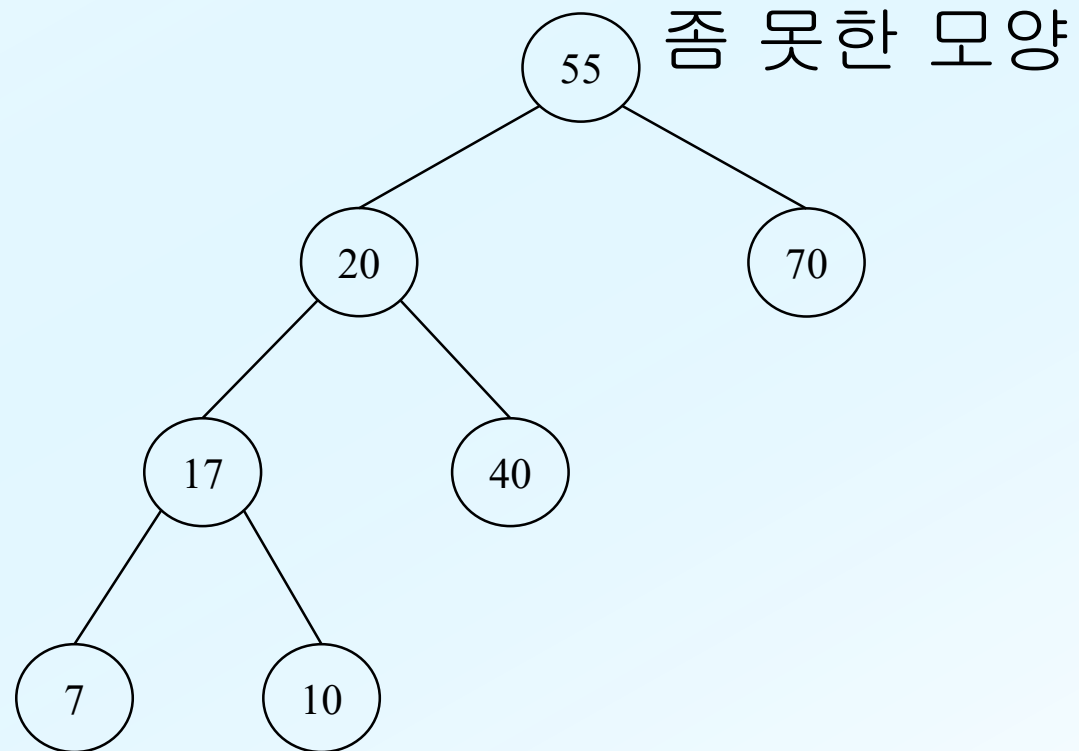
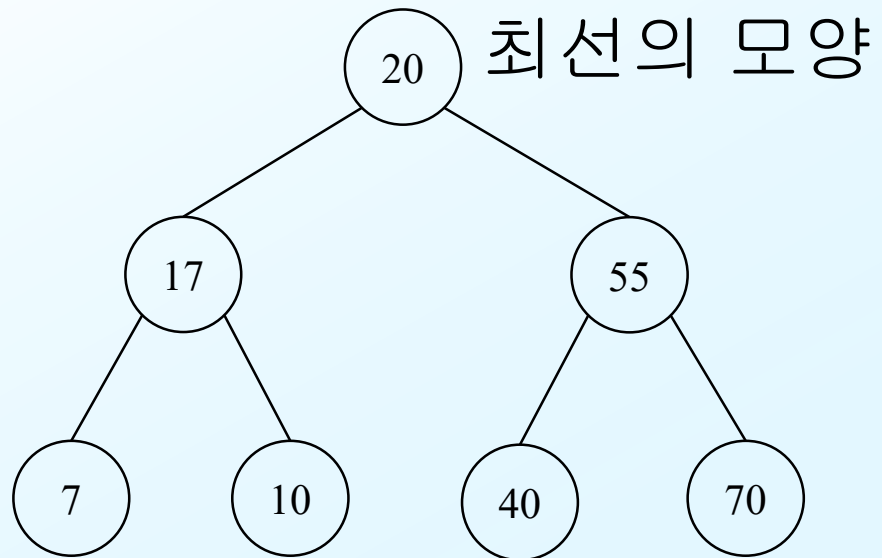
이진 검색 트리 객체 구조



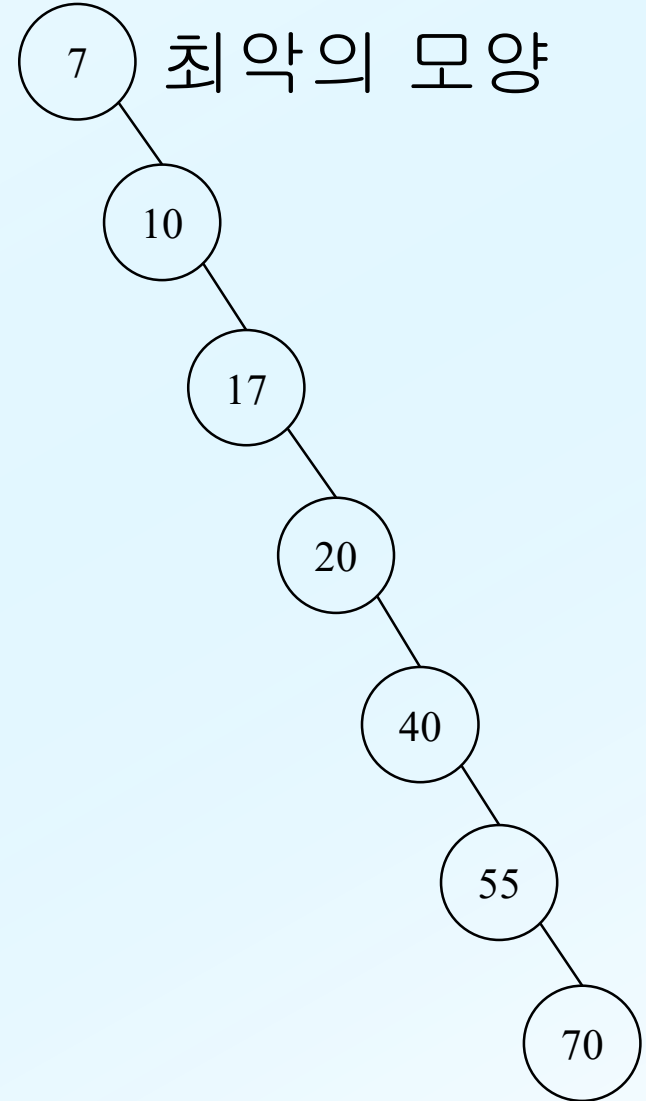
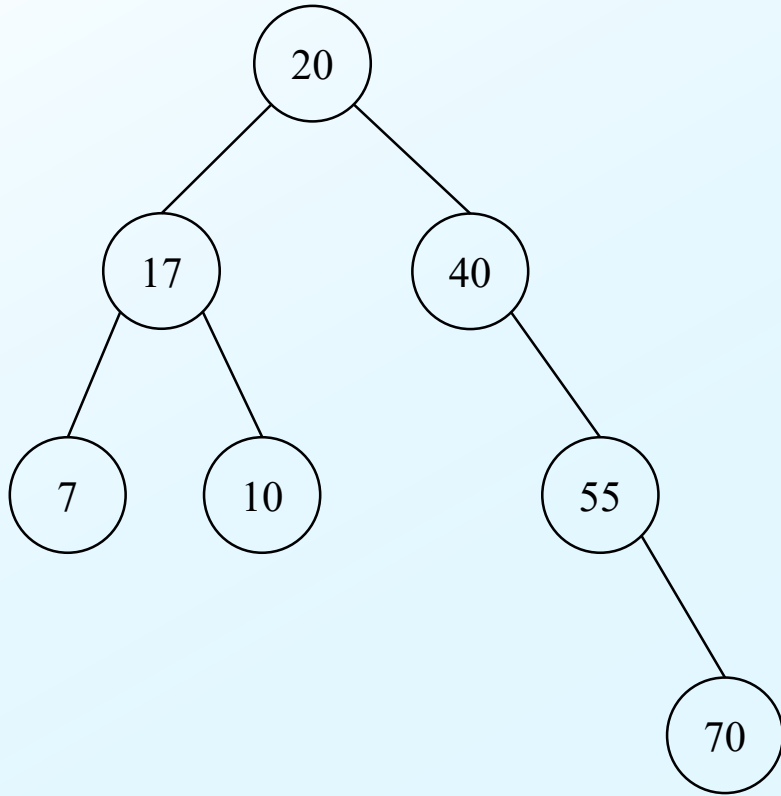
보통 이렇게 그린다



같은 데이터, 다른 이진 검색 트리



이런 모양이 될 수도..



이진검색트리 알고리즘과 구현

검색 Search

search(t, x):

◀ t : (서브) 트리의 루트 노드의 레퍼런스

◀ x : 검색하고자 하는 키

if ($t = \text{null} \parallel t.\text{item} = x$)

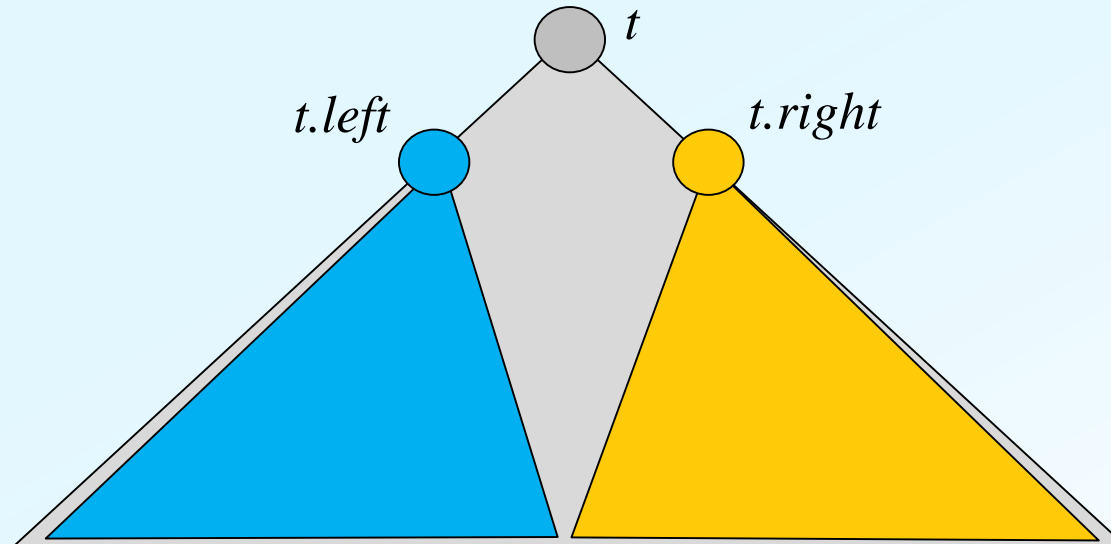
return t ◀ 검색 실패 or 성공

else if ($x < t.\text{item}$)

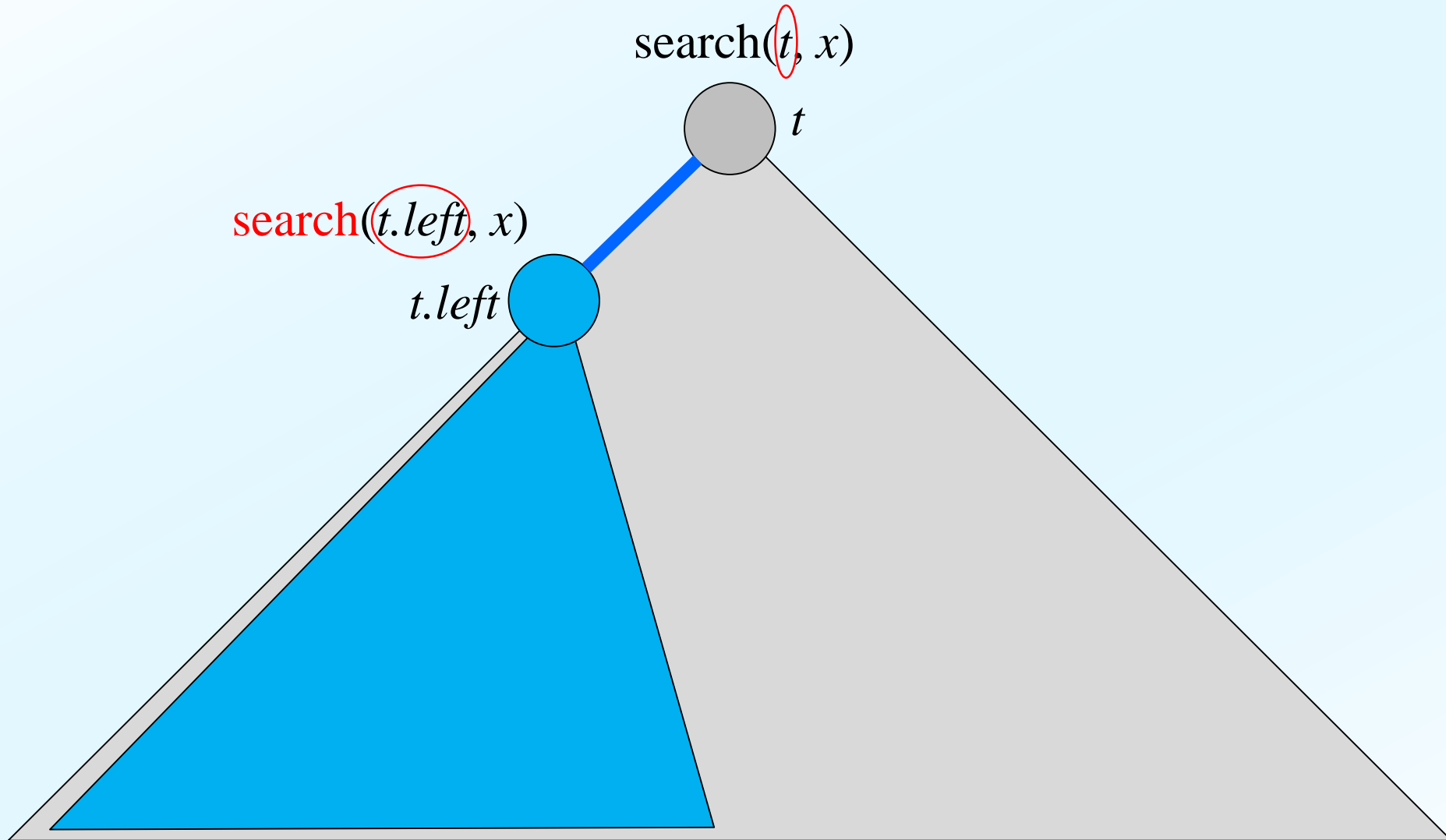
return **search**($t.\text{left}, x$) ◀ $t.\text{left}$: t 's left child

else

return **search**($t.\text{right}, x$) ◀ $t.\text{right}$: t 's right child

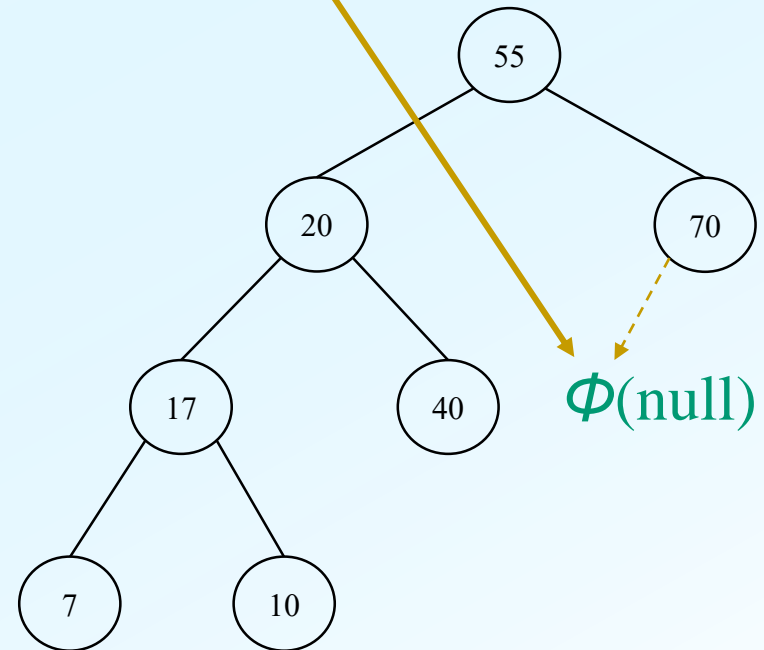


재귀적 관점으로 본 검색



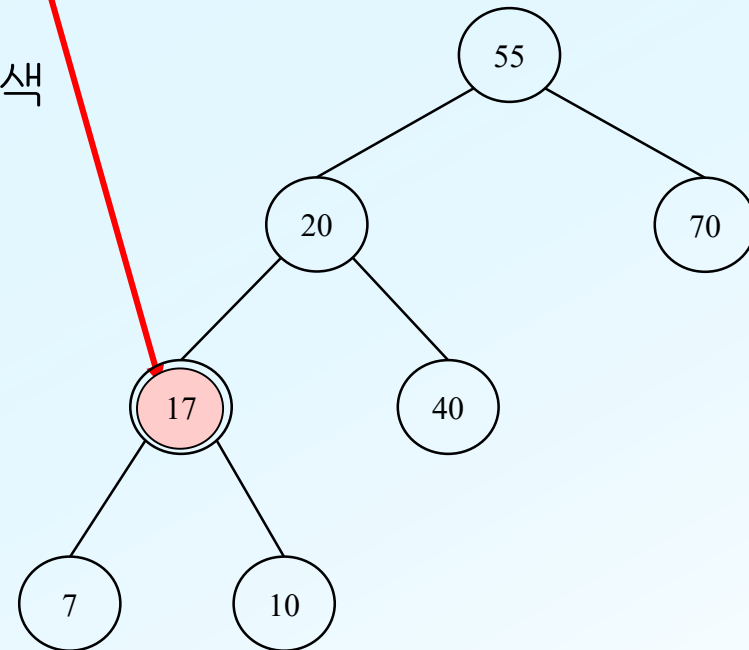
```
search(t, x):  
  if ( t = null || t.item = x )  
    return t  
  else if (x < t.item)  
    return search(t.left, x)  
  else  
    return search(t.right, x)
```

실패하는 검색



```
search(t, x):  
    if ( t = null || t.item = x )  
        return t  
    else if (x < t.item)  
        return search(t.left, x)  
    else  
        return search(t.right, x)
```

성공하는 검색



삽입 Insertion

가정: 삽입 키 x 는 검색트리에 없다

insertSketch(t, x):

◀ t : (서브) 트리의 루트 노드

◀ x : 삽입하고자 하는 키

if ($t = \text{null}$)

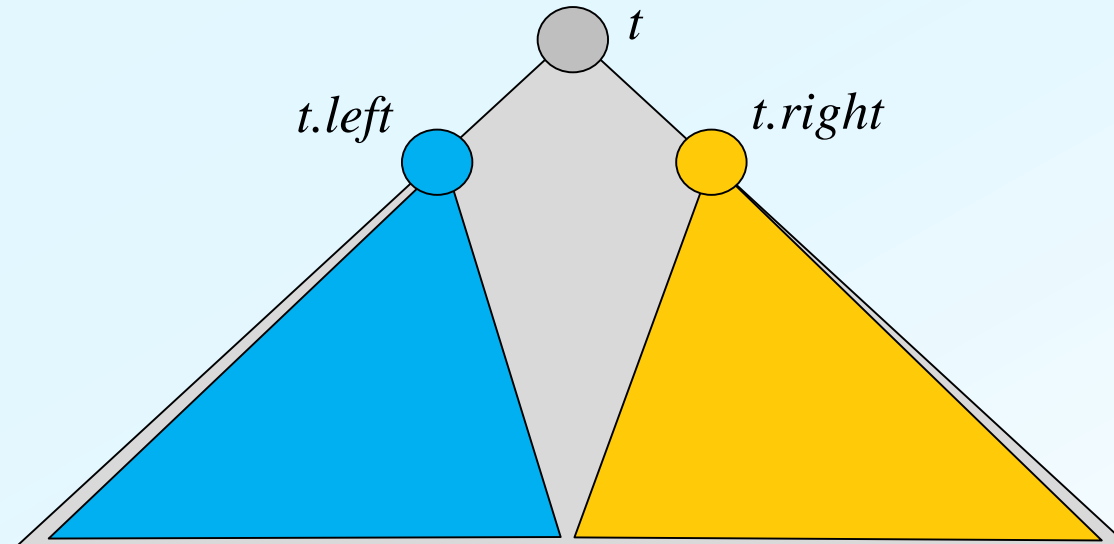
x 를 키로 하는 노드를 t 의 부모 밑에 매달고 끝낸다

else if ($x < t.\text{item}$)

insertSketch($t.\text{left}, x$)

else

insertSketch($t.\text{right}, x$)



Animation

insertSketch(t, x):

◀ t : (서브) 트리의 루트 노드

◀ x : 삽입하고자 하는 키

```
if ( $t = \underline{\text{null}}$ )
```

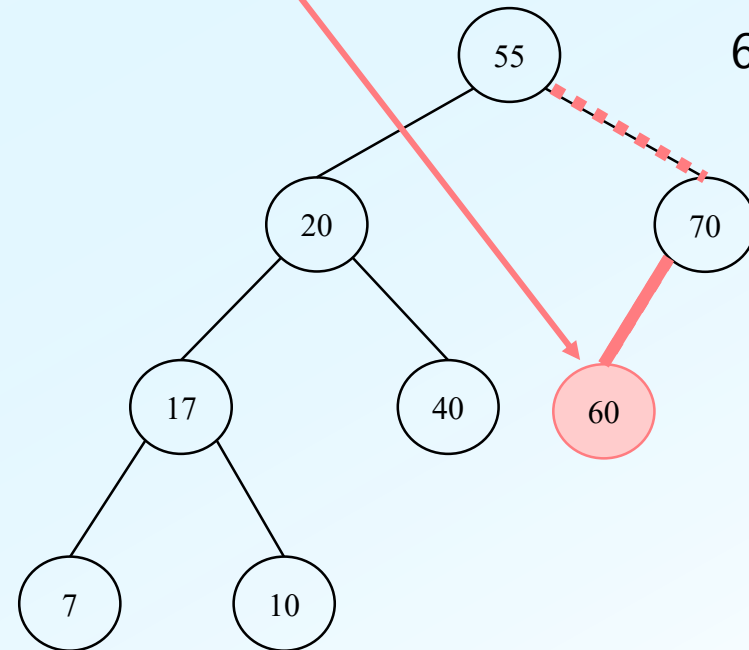
x 를 키로 하는 노드를 t 의 부모 밑에 매달고 끝낸다

```
else if ( $x < t.item$ )
```

insertSketch($t.left, x$)

else

insertSketch($t.right, x$)



60 삽입하라

알고리즘 insert()

insert(x): ◀ $root$: 트리의 루트 노드 레퍼런스
 $root \leftarrow \text{insertItem}(root, x)$

insertItem(t, x):

◀ t : (서브) 트리의 루트 노드

◀ x : 삽입하고자 하는 키

◀ 작업 완료 후 루트 노드의 레퍼런스를 리턴한다

if ($t = \text{null}$)

$r.\text{item} \leftarrow x; r.\text{left} \leftarrow \text{null}; r.\text{right} \leftarrow \text{null}$ ◀ r : 새 노드

return r

else if ($x < t.\text{item}$)

$t.\text{left} \leftarrow \text{insertItem}(t.\text{left}, x)$

return t

else

$t.\text{right} \leftarrow \text{insertItem}(t.\text{right}, x)$

return t

삭제 Deletion

deleteSketch(r): ◀ r : 삭제하고자 하는 노드

if (r 이 리프 노드) ◀ Case 1

그냥 r 을 버린다

else if (r 의 자식이 하나만 있음) ◀ Case 2

r 의 부모가 r 의 자식을 직접 가리키도록 한다

else ◀ Case 3

r 의 right subtree 의 minimum 원소 노드 s 를 찾는다

s 를 r 자리로 복사하고 s 를 삭제한다

deleteSketch(r): ◀ r : 삭제하고자 하는 노드

if (r 이 리프 노드) ◀ Case 1

그냥 r 을 버린다

else if (r 의 자식이 하나만 있음) ◀ Case 2

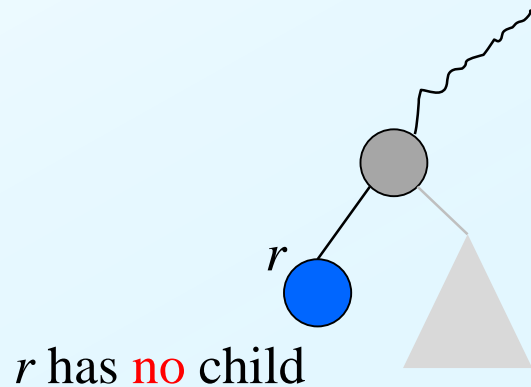
r 의 부모가 r 의 자식을 직접 가리키도록 한다

else ◀ Case 3

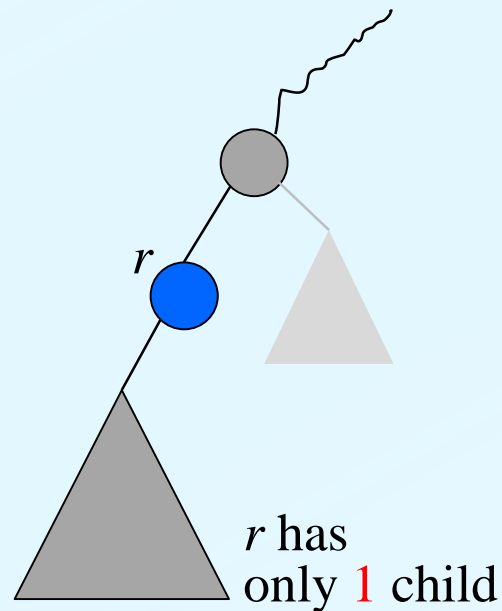
r 의 right subtree 의 minimum 원소 노드 s 를 찾는다

s 를 r 자리로 복사하고 s 를 삭제한다

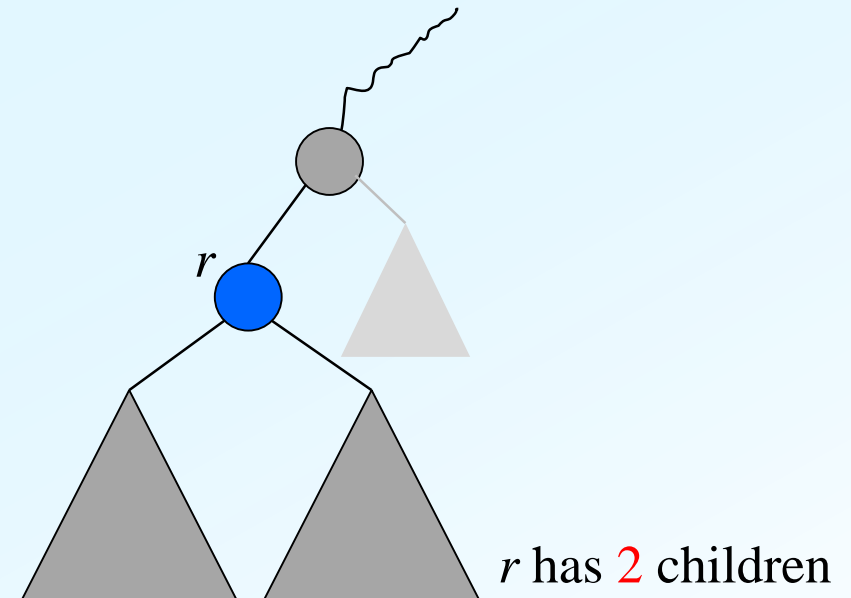
Case 1



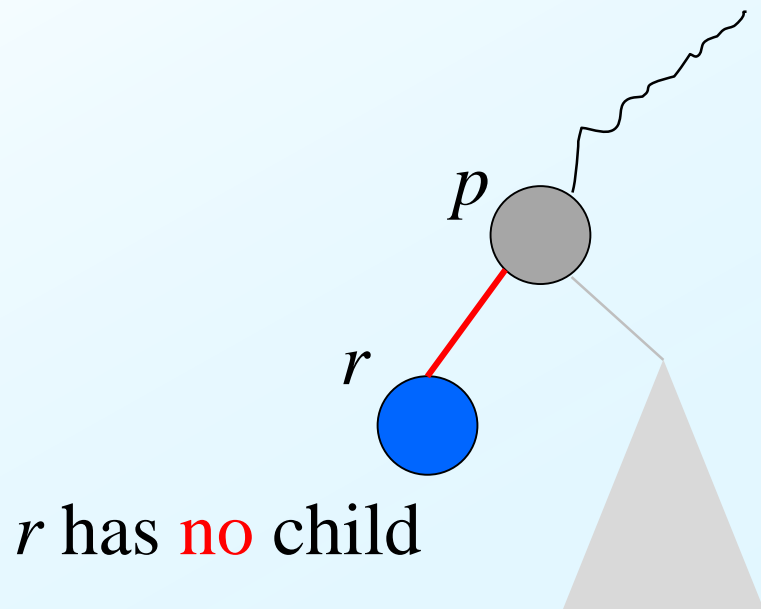
Case 2



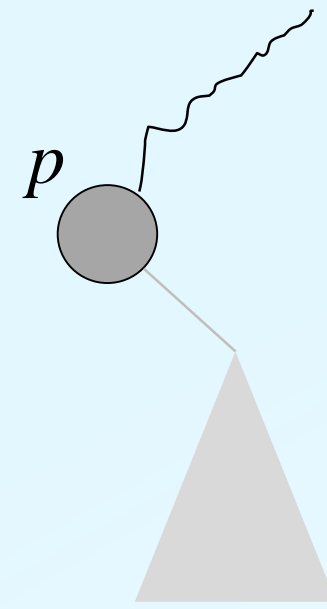
Case 3

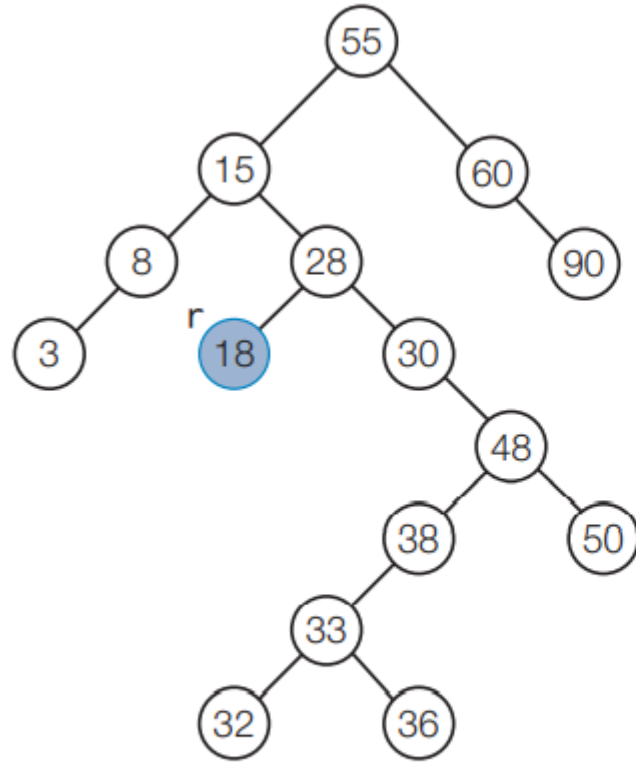


Case 1

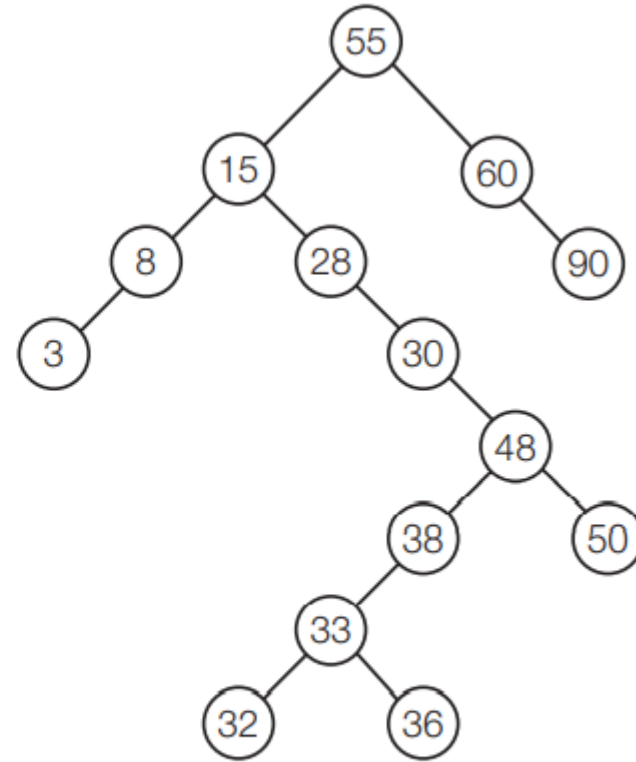


삭제 결과





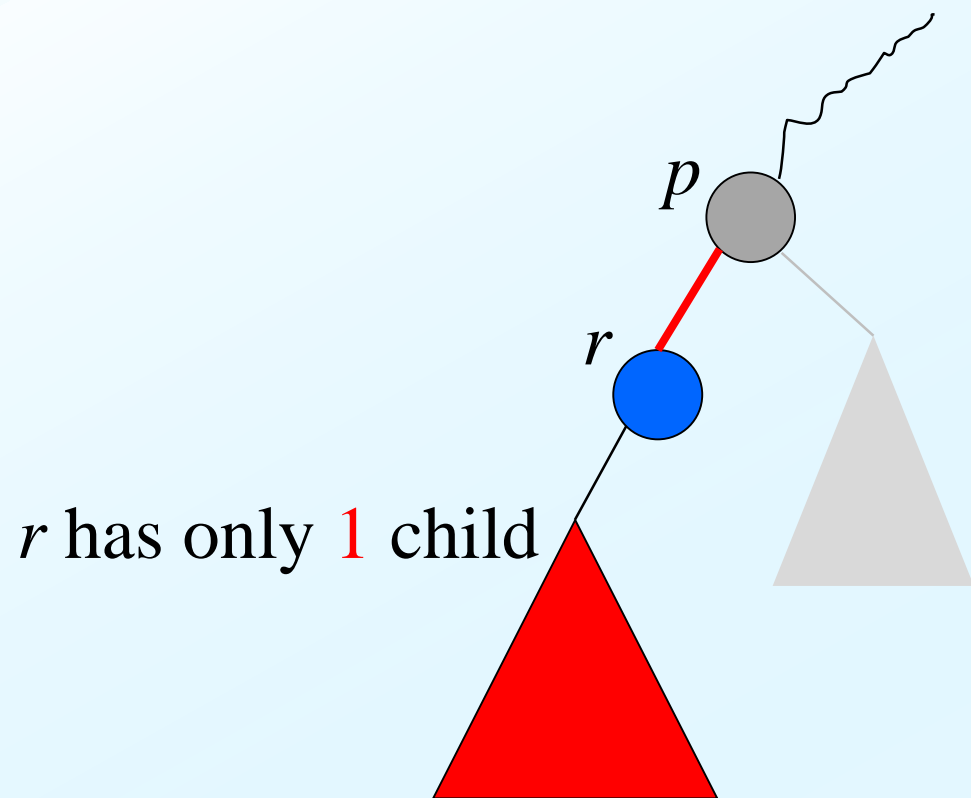
(a) r의 자식이 없음



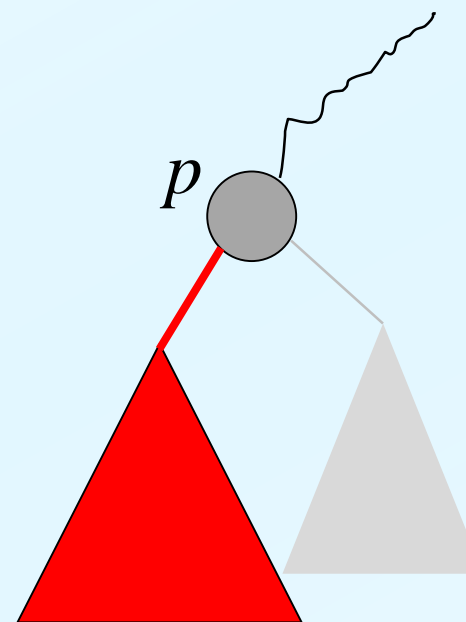
(b) 단순히 r을 제거함

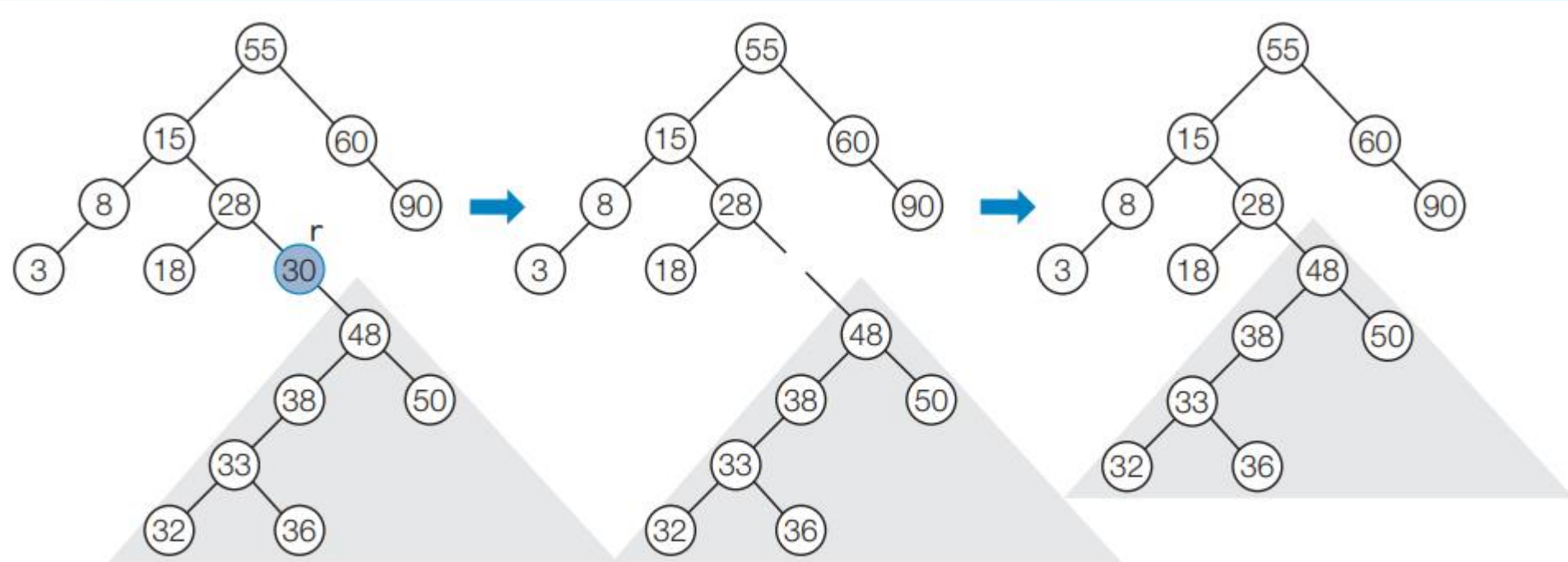
그림 10-17 이진 검색 트리의 삭제 Case 1

Case 2



삭제 결과





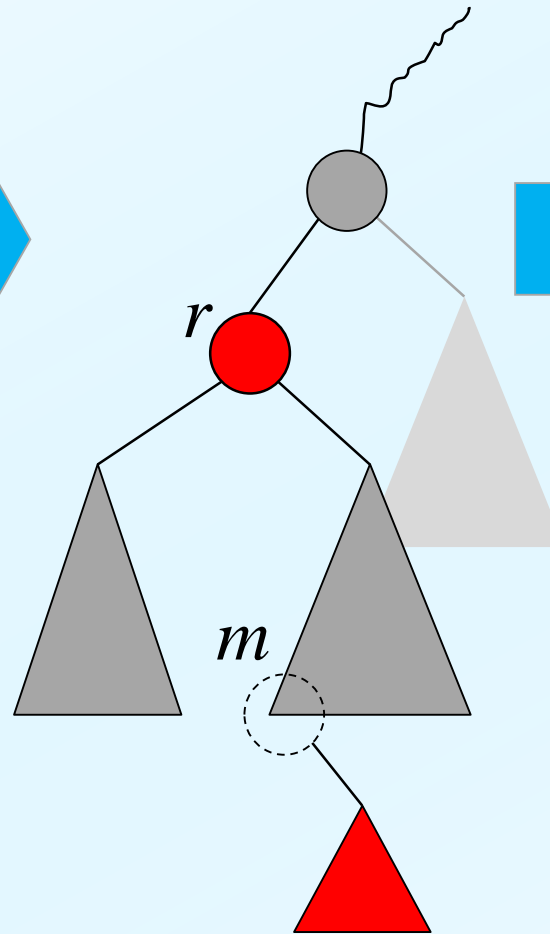
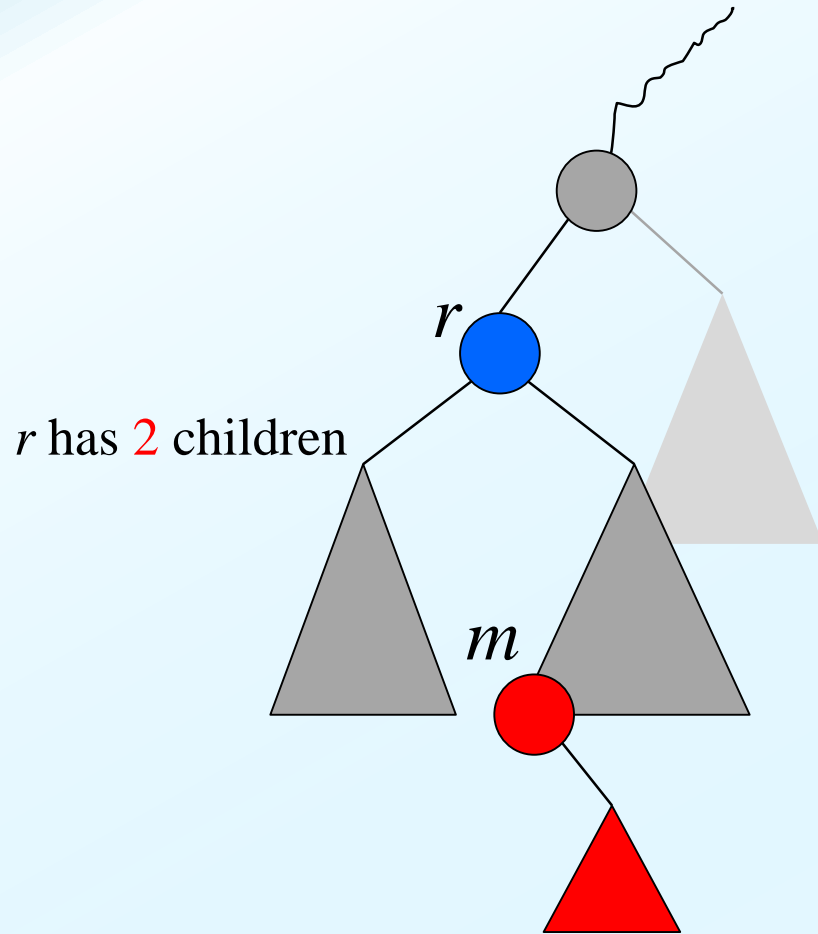
(a) r의 자식이 하나뿐임

(b) r을 제거함(개념상)

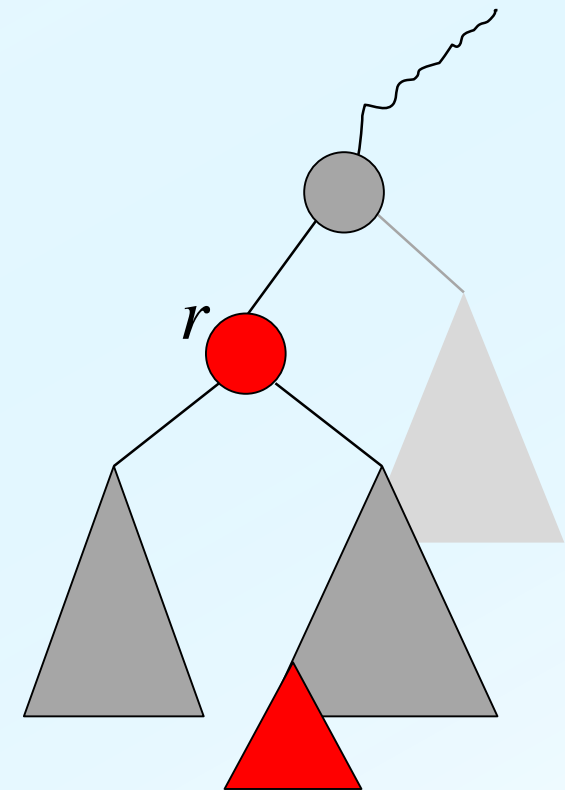
(c) r 자리에 r의 자식을 놓음

그림 10-18 이진 검색 트리에서의 삭제 Case 2

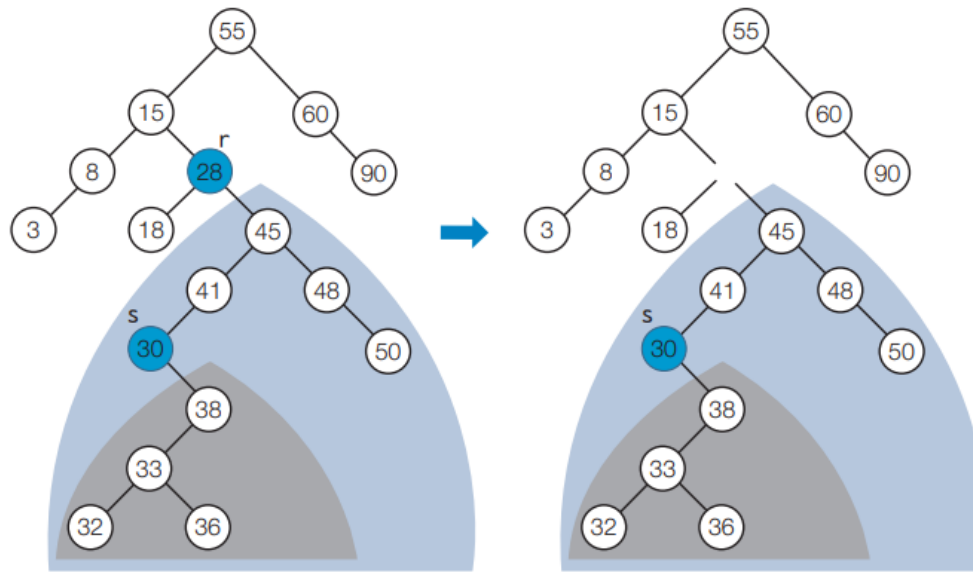
Case 3



삭제 결과

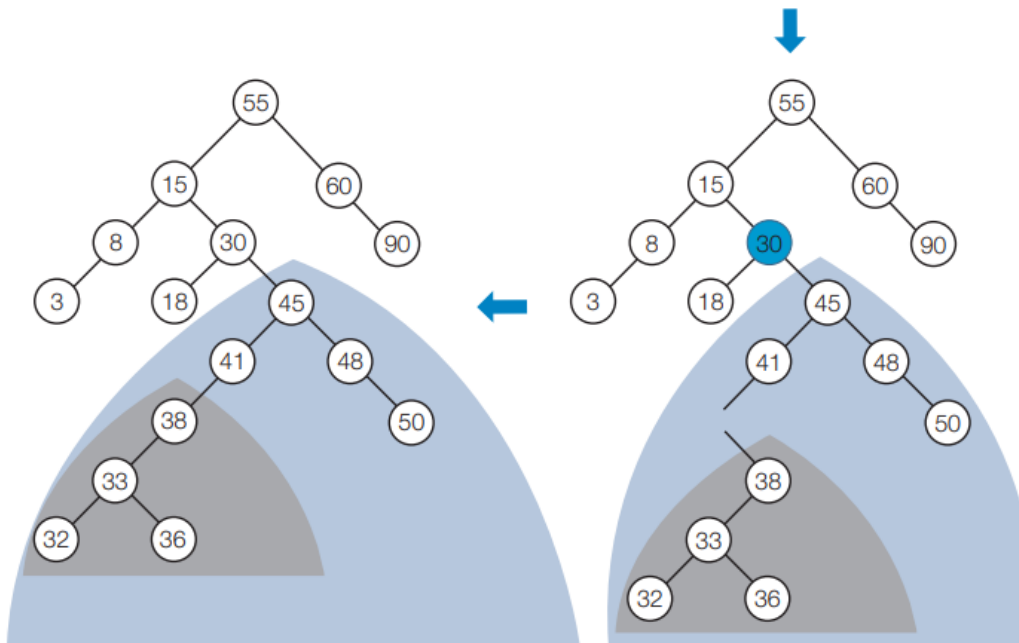


실제 예



(a) r의 직후 원소 s를 찾음

(b) r을 없앴(개념상)



(d) s가 있던 자리에 s의 자식을 놓음

(c) s를 r자리로 옮김

알고리즘 delete()

delete(*root*, *x*): ◀ 삭제할 '원소'가 주어지는 버전

root \leftarrow findAndDelete(*root*, *x*)

findAndDelete(*t*, *x*): ◀ *t*: (서브) 트리의 루트. 교재의 deleteItem()

if (*t* = **null**)

return ItemNotFound

else if (*x* = *t.item*)

t \leftarrow deleteNode(*t*);

◀ item found at *t*

return *t*

else if (*x* < *t.item*)

t.left \leftarrow findAndDelete(*t.left*, *x*)

return *t*

else

t.right \leftarrow findAndDelete(*t.right*, *x*)

return *t*

```

deleteNode(t): ◀ t: node to delete
    if (t.left = null && t.right = null)           ◀ case 1 (no child)
        return null
    else if (t.left = null)                         ◀ case 2-1 (only right child)
        return t.right
    else if (t.right = null)                       ◀ case 2-2 (only left child)
        return t.left
    else                                             ◀ case 3 (2 children)
        (minItem, node) = deleteMinItem(t.right)
        t.item ← minItem
        t.right ← node
        return t ◀ t를 직접 지우지 않음

```

```

deleteMinItem(t):
    if (t.left = null)                             ◀ found minimum at t
        return (t.item, t.right)
    else                                             ◀ branch left, then backtrack
        (minItem, node) ← deleteMinItem(t.left)
        t.left ← node
        return (minItem, t)

```

검색 시간

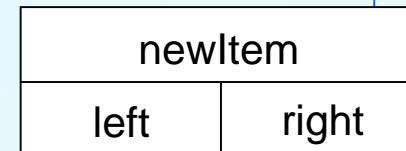
Fact:

총 n 개의 노드를 가진 binary search tree의
점근적 평균 검색 시간은 $\Theta(\log n)$ 이다.

자바 코드

```
public interface IndexInterface<T> {  
    public void insert(Comparable x);  
    public T search(Comparable x);  
    public void delete(Comparable x);  
}
```

```
public class TreeNode {  
    public Comparable item;  
    public TreeNode left;  
    public TreeNode right;  
    public TreeNode(Comparable newItem) {  
        item = newItem;  
        left = right = null;  
    }  
    public TreeNode(Comparable newItem,  
                    TreeNode leftChild, TreeNode rightChild) {  
        item = newItem;  
        left = leftChild;  
        right = rightChild;  
    }  
}
```



```

public class BinarySearchTree implements IndexInterface<TreeNode> {
    private TreeNode root;
    public BinarySearchTree() { // 생성자
        root = null;
    }
    ///// 검색 //////////////////////////////////////
    public TreeNode search(Comparable x) {
        return searchItem(root, x);
    }
    private TreeNode searchItem(TreeNode t, Comparable x) {
        if (t == null) return null;
        else if (x.compareTo(t.item) == 0) return t;
        else if (x.compareTo(t.item) < 0)
            return searchItem(t.left, x);
        else
            return searchItem(t.right, x);
    }
    ///// 삽입 //////////////////////////////////////
    public void insert(Comparable x) {
        root = insertItem(root, x);
    }
    private TreeNode insertItem(TreeNode t, Comparable x) {
        if (t == null) { // insert after a leaf (or into an empty tree)
            t = new TreeNode(x);
        }
        else if (x.compareTo(t.item) < 0)
            t.left = insertItem(t.left, x); // branch left
        else
            t.right = insertItem(t.right, x); // branch right
        return t;
    }
    ...
}

```

////// 삭제 //////////////////////////////////////

```
public void delete(Comparable x) {
    root = findAndDelete(root, x);
}
private TreeNode findAndDelete(TreeNode t, Comparable x) {
    if (t == null) return null;          // item not found!
    else {
        if (x.compareTo(t.item) == 0)
            t = deleteNode(t);          // item found at t
        else if (x.compareTo(t.item) < 0)
            t.left = findAndDelete(t.left, x);
        else
            t.right = findAndDelete(t.right, x);
        return t;
    }
}
private TreeNode deleteNode(TreeNode t) {
    if (t.left == null && t.right == null)
        return null;                  // case 1 (no child)
    else if (t.left == null)
        return t.right;                // case 2-1 (only right child)
    else if (t.right == null)
        return t.left;                 // case 2-2 (only left child)
    else { // case 3 (two children)
        returnPair rPair = deleteMinItem(t.right);
        t.item = rPair.minItem;
        t.right = rPair.node;
        return t // t survived
    }
}
...
}
```

```

private returnPair deleteMinItem(TreeNode t) {
    if (t.left == null) { // found minimum at t
        return new returnPair(t.item, t.right);
    } else { // branch left, then backtrack
        returnPair rPair = deleteMinItem(t.left);
        t.left = rPair.node;
        rPair.node = t;
        return rPair;
    }
}

private class returnPair {
    Comparable minItem;
    TreeNode node;
    private returnPair(Comparable x, TreeNode t) {
        minItem = x;
        node = t;
    }
}

} // end BinarySearchTree

```

3/3 of class BinarySearchTree

이진 검색 트리의 성질

팩트 1

정수 $h \geq 1$ 에 대해, 높이 h 인 포화 이진 트리의 리프 노드의 총 수 $l(h)$ 는 2^{h-1} 이다.

<증명>

- i) (베이스 케이스) $h=1$ 이면 리프 노드가 1개이므로 $l(1)=2^{1-1}=2^0=1$ 이 만족된다.
- ii) (귀납적 가정) $h=k$ 인 포화 이진 트리의 리프 노드 수 $l(k)=2^{k-1}$ 이라 가정하자.
- iii) (귀납적 전개) 이제 $l(k+1)=2^k$ 임을 증명하면 된다. 높이 $k+1$ 인 포화 이진 트리는 루트의 두 서브 트리가 각각 높이 k 인 포화 이진 트리다. 총 리프 노드 수는 $l(k+1)=2 \cdot l(k)=2 \cdot 2^{k-1}=2^k$ 이 되어 증명이 완료된다.

이진 검색 트리의 성질

팩트 2

정수 $h \geq 0$ 에 대해, 높이 h 인 포화 이진 트리의 총 노드 수 $n(h)$ 는 $2^h - 1$ 이다.

<증명>

- i) (베이스 케이스) $h=0$ 이면 비어 있는 이진 트리다. 포화 이진 트리에 속한다. 노드 수는 0이다. $n(0)=2^0-1=0$ 이 성립한다.
- ii) (귀납적 가정) 모든 $0 \leq k < h$ 에 대해 노드 수 $n(k)=2^k-1$ 이라 가정하자.
- iii) (귀납적 전개) 이제 $n(h)=2^h-1$ 임을 증명하면 된다. 앞의 귀납적 가정에 의해 높이 $h-1$ 인 포화 이진 트리의 노드 수 $n(h-1)=2^{h-1}-1$ 이다. 높이 h 인 포화 이진 트리는 루트의 두 서브 트리가 높이 $h-1$ 인 포화 이진 트리이므로 총 노드 수는 $n(h)=1+2 \cdot n(h-1)=1+2(2^{h-1}-1)=2^h-1$ 이 되어 증명이 완료된다. (여기서 1을 더하는 것은 루트 노드를 더한 것)

이진 검색 트리의 성질

팩트 3 정수 $h \geq 0$ 에 대해, 높이 h 인 이진 트리는 $2^h - 1$ 개 이하의 노드를 갖는다.

<증명>

높이 h 인 이진 트리 중 가장 많은 노드 수를 갖는 것은 포화 이진 트리다. 높이 h 인 포화 이진 트리는 $2^h - 1$ 개의 노드를 가지므로([팩트 2]), 높이 h 인 모든 이진 트리는 기껏해야 $2^h - 1$ 개의 노드를 가진다.

이진 검색 트리의 성질

팩트 4 총 n 개의 노드를 가진 이진 트리의 높이는 적어도 $h \geq \lceil \log_2(n+1) \rceil$ 이다.

<증명>

이진 트리의 높이가 h 라면 해당 이진 트리의 노드 수는 [팩트 3]으로부터 $n \leq 2^h - 1$ 이 된다. 변환하면 $h \geq \log_2(n+1)$ 이다. 높이 h 는 정수이므로 $h \geq \lceil \log_2(n+1) \rceil$ 가 된다. 즉, 총 n 개의 노드를 가진 이진 트리의 높이는 $\Omega(\log n)$ 로 표현할 수 있다.

팩트 5 총 n 개의 노드를 가진 이진 트리의 최대 높이는 n 이다.

팩트 6 이진 검색 트리의 삽입에 드는 점근적 시간은 검색에 드는 시간과 동일하다.

팩트 7 n 개의 노드를 가진 이진 검색 트리의 점근적 평균 검색 시간은 $\Theta(\log n)$ 이다.

이진 트리에서의 순회 Traversal

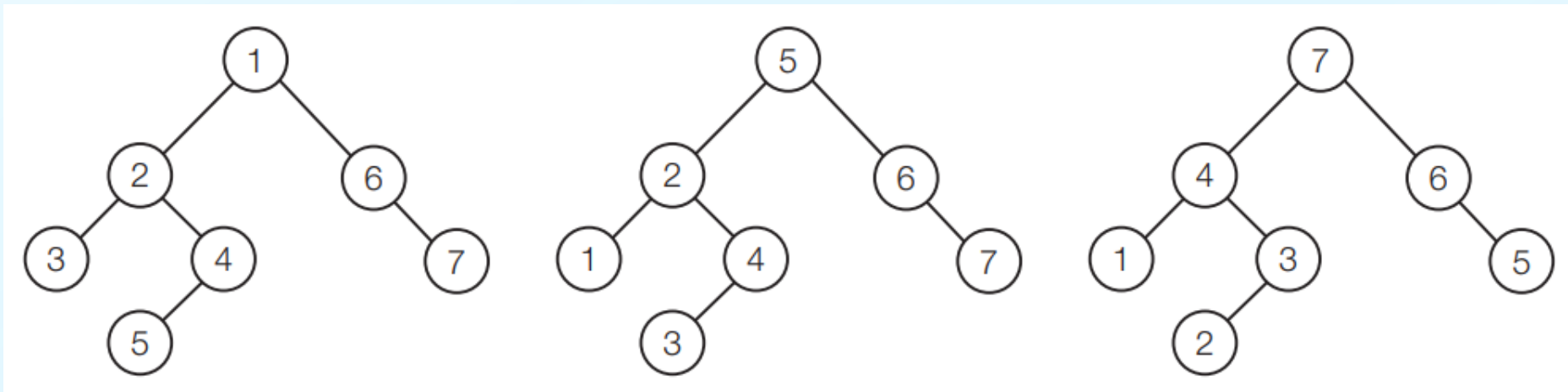
순회 Traversal: 이진 트리에서 모든 노드를 방문하기

전위 순회 Preorder Traversal

중위 순회 Inorder Traversal

후위 순회 Postorder Traversal

노드에 적은 번호는 방문 순서



전위 순회
- 루트를 맨 먼저

중위 순회
- 루트를 중간에

후위 순회
- 루트를 맨 뒤에

알고리즘 preOrder()

```
preOrder(root):  
    if (root is not empty)  
        Mark root  
        preOrder(Left child of root)  
        preOrder(Right child of root)
```

알고리즘 inOrder()

```
inOrder(root):  
    if (root is not empty)  
        inOrder(Left child of root)  
        Mark root  
        inOrder(Right child of root)
```

알고리즘 postOrder()

```
postOrder(root):  
    if (root is not empty)  
        postOrder(Left child of root)  
        postOrder(Right child of root)  
    Mark root
```


중위 순회는 이진 검색 트리를 정렬 순서로 방문한다

Theorem: 이진 검색 트리 T 의 중위 순회는 노드들을 정렬된 순서로 방문한다

<증명>

베이스 케이스: $h = 1$. (h : height)

T 는 단 한 개의 노드(루트)를 갖는다.

유일한 노드를 방문하는 것은 당연히 정렬된 순서다.

귀납적 가정: $k < h$ 인 모든 k 에 대해 성립한다 가정하자.

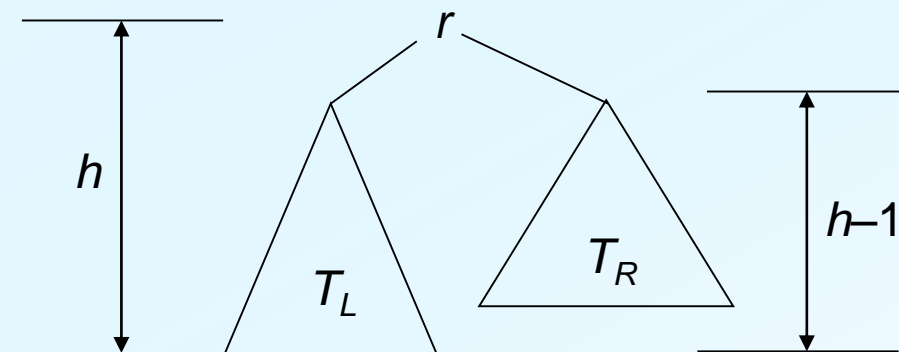
귀납적 전개: (정리가 $k = h$ 에 대해 성립함을 보인다)

이진 검색 트리 T 는 오른쪽과 같이 그릴 수 있다.

귀납적 가정에 의해 중위 순회는 T_L 과 T_R 을 각각 정렬된 순서로 방문한다.

T_L 의 모든 키는 r 의 키보다 작고, T_R 의 모든 키는 r 의 키보다 크므로,

중위 순회 $T_L \rightarrow r \rightarrow T_R$ 는 정렬된 순서다.

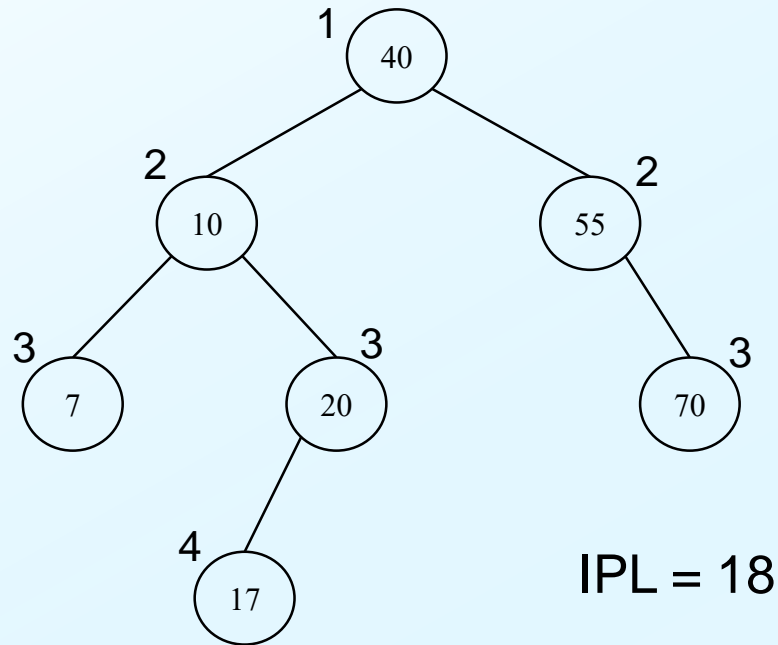


이진 검색 트리 작업의 효율

작업	평균적인 경우	최악의 경우
검색	$\Theta(\log n)$	$\Theta(n)$
삽입	$\Theta(\log n)$	$\Theta(n)$
삭제	$\Theta(\log n)$	$\Theta(n)$
순회	$\Theta(n)$	$\Theta(n)$

Internal Path Length (IPL)

- 각 노드의 깊이를 모두 더한 것



Fact:

n 개의 원소가 차례로 입력으로 들어와서 이진 검색 트리를 만들 때, 입력 원소의 모든 순서가 이루는 모든 순열이 같은 확률을 가진다면, 만들어진 이진 검색 트리의 기대 IPL은 $O(n \log n)$ 이다

의미하는 것: 평균 검색 시간은 $O(\log n)$

트리 사이즈 구하기

size(t):

if (t is **null**) return 0

else return (1 + **size**(t.left) + **size**(t.right))

Tree Sort

이진 검색 트리를 이용해서 정렬한다

treeSort(A[], n):

A[0...n-1]을 차례로 읽어 이진 검색 트리 T를 만든다
T를 중위 순회하면서 A[0...n-1]에 저장한다

시간 복잡도: 평균 $\Theta(n \log n)$

이진 검색 트리 만들기 - 평균 $\Theta(n \log n)$

중위 순회 - $\Theta(n)$

이론적인 성능은 좋으나 $\Theta(n \log n)$ 에 포함된 상수 인자가 크다

정렬용으로 사용해도 되나, 말리고 싶다

이진 검색 트리를 파일에 저장하기

이진 검색 트리를 파일에 저장했다가 필요할 때 다시 메모리로 로딩하려 한다면
전위 순회 순으로 저장할 수도 있고, 중위 순회 순으로 저장할 수도 있다

- 원래 모양 그대로 복원하려면 – 전위 순회
- 완벽하게 균형잡힌 모양으로 복원하려면 – 중위 순회

중위 순회로 저장된 리스트를 이진 검색 트리로 복원하기

recursiveRestore (L): ◀ L : 원소 리스트

if (L is **null**)

return null

else

L 의 중앙값 원소 r 을 찾아 가져와 루트로 삼는다

$r.left = \text{recursiveRestore}(r \text{ 앞에 있는 원소들의 리스트})$

$r.right = \text{recursiveRestore}(r \text{ 뒤에 있는 원소들의 리스트})$

return r