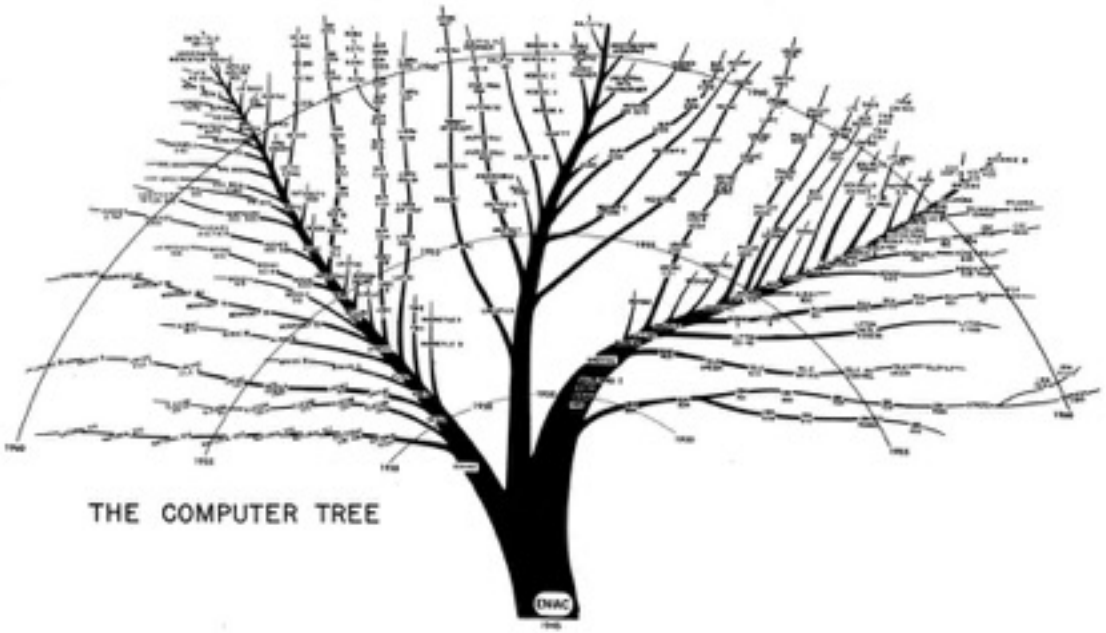
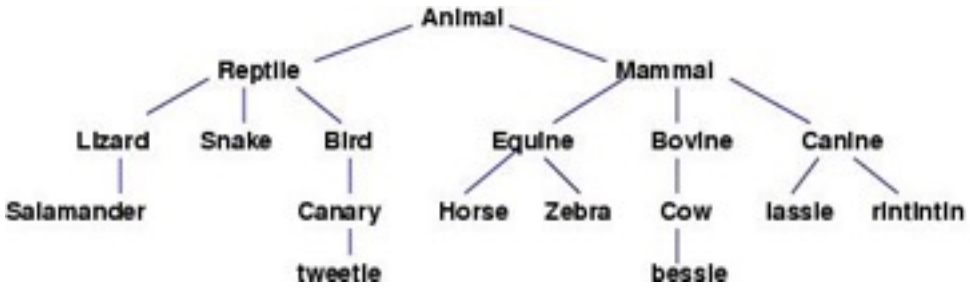


제4장

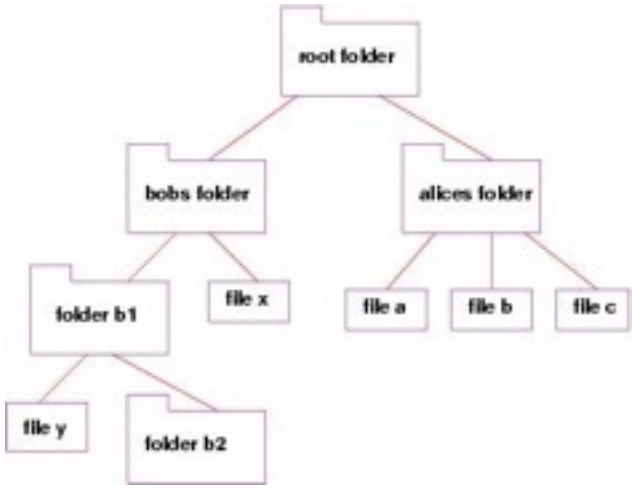
검색트리 (search tree)

트리와 이진트리
Tree and Binary Tree

트리 (Tree)

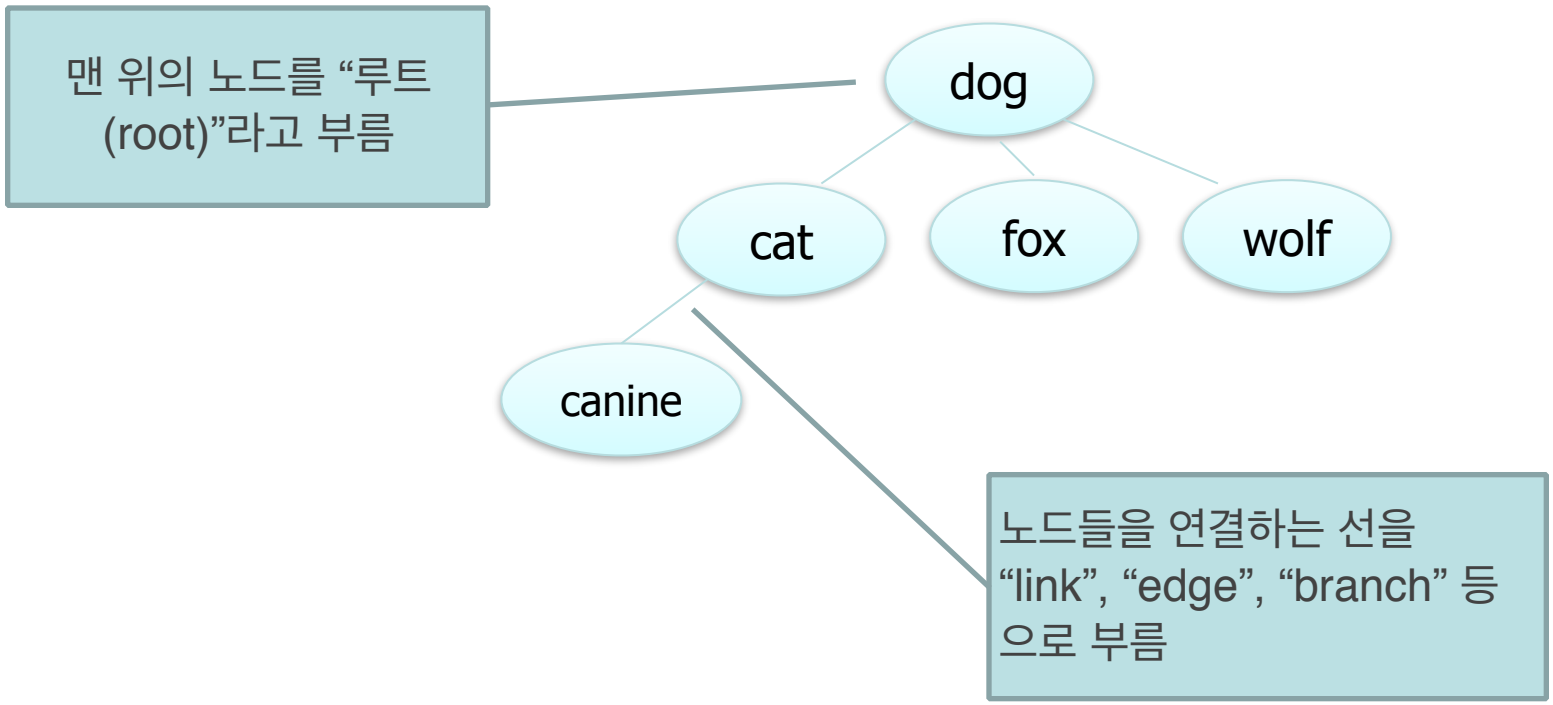


- 계층적인 구조를 표현
  - 조직도
  - 디렉토리와 서브디렉토리 구조
  - 가계도

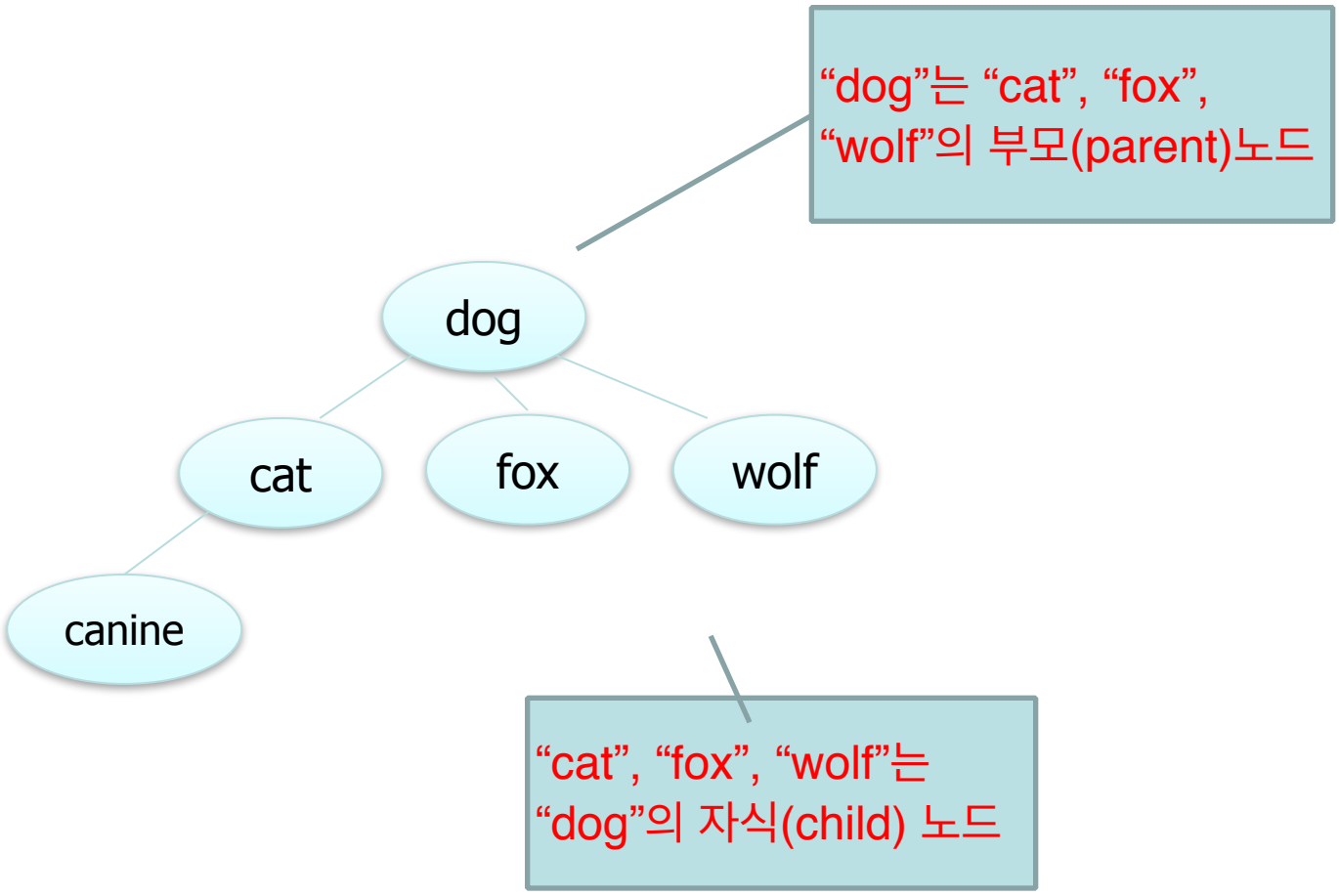


용어: 루트

- 트리는 노드(node)들과 노드들을 연결하는 링크(link)들로 구성됨

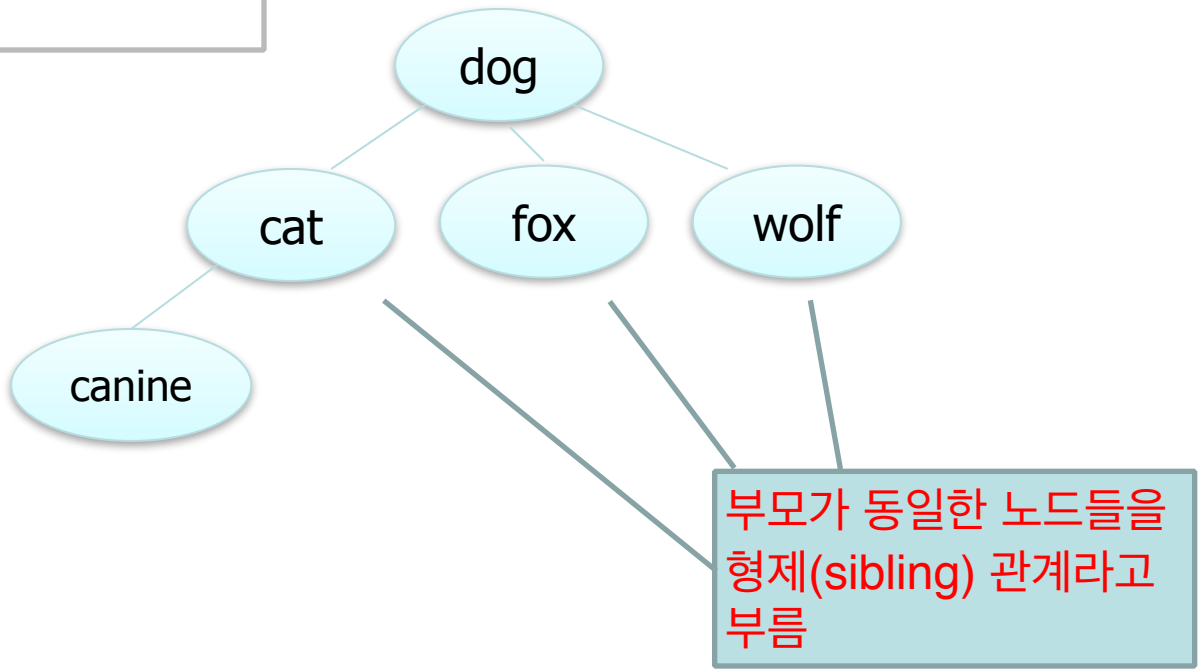


용어: 부모-자식 관계

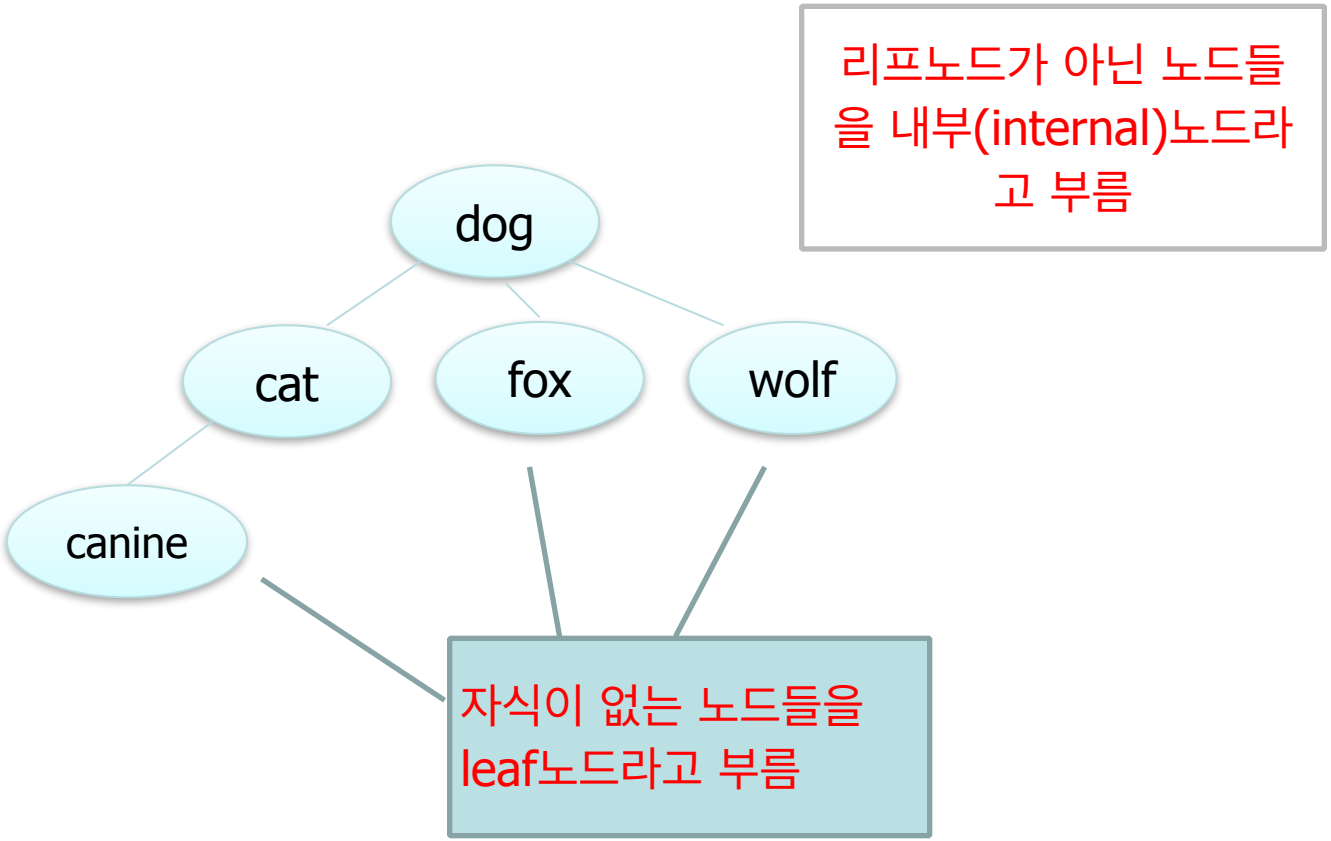

"dog"는 "cat", "fox", "wolf"의 부모(parent)노드
dog
cat
fox
wolf
canine
"cat", "fox", "wolf"는 "dog"의 자식(child) 노드

용어: 형제관계

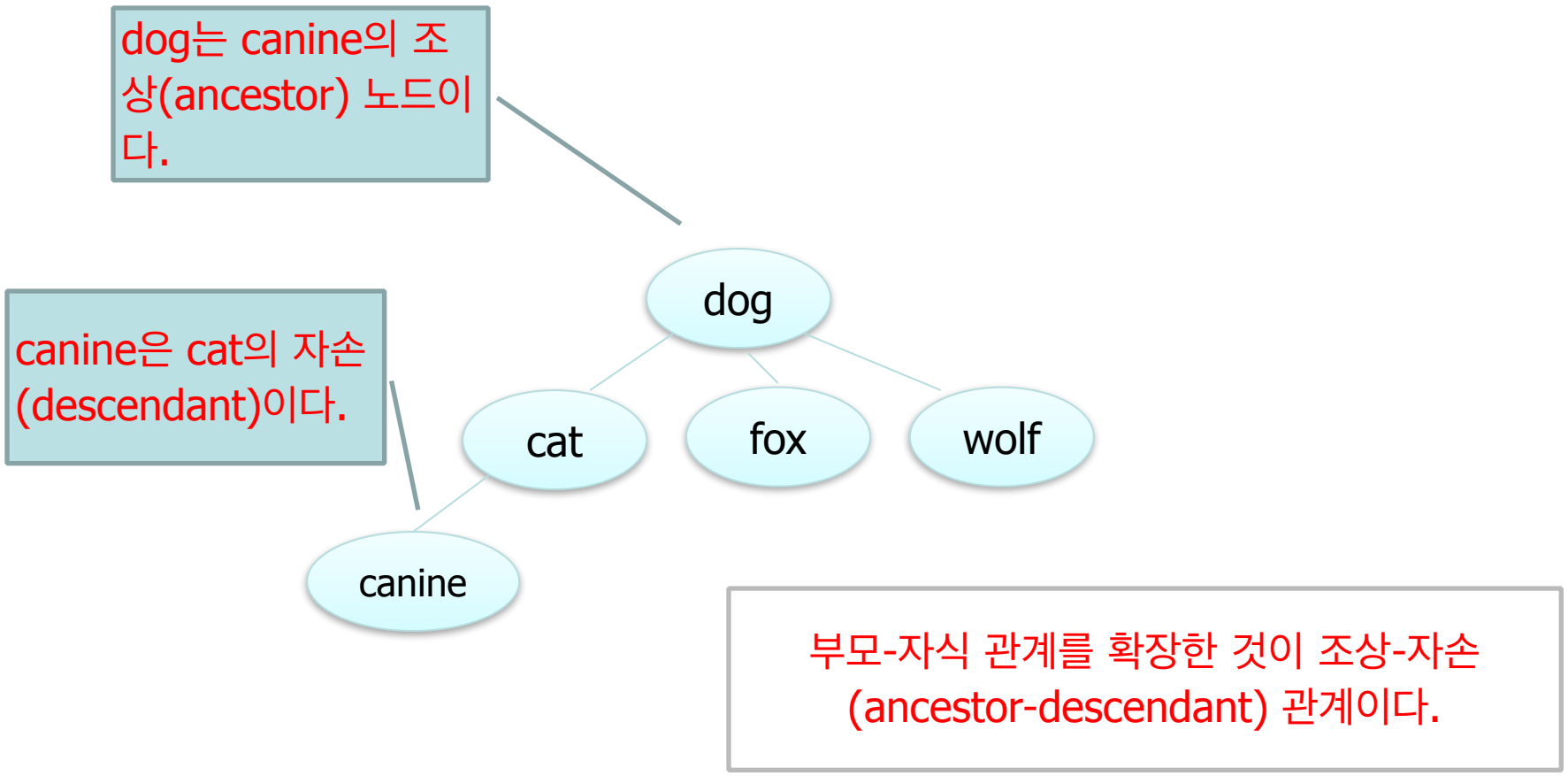
루트노드를 제외한 트리의 모든 노드들은 유일한 부모 노드를 가짐



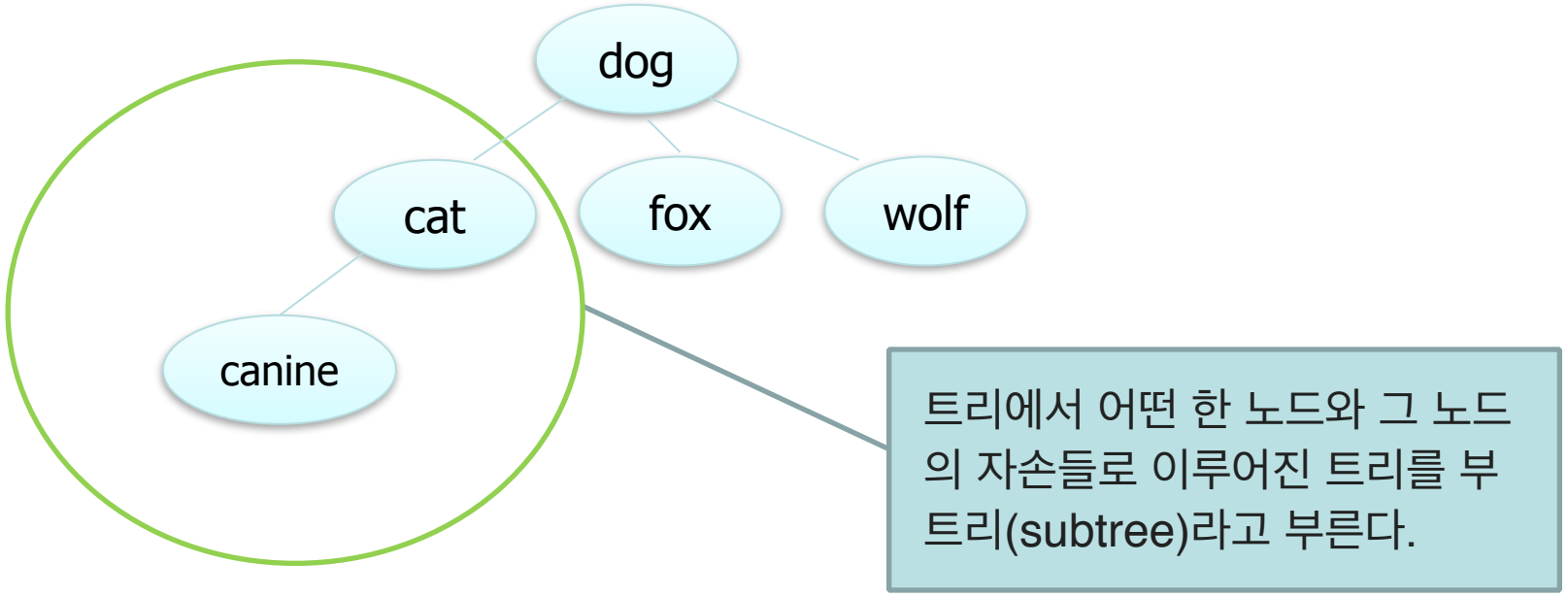
용어: 리프(leaf) 노드


리프노드가 아닌 노드들을 내부(internal)노드라고 부름
dog
cat
fox
wolf
canine
자식이 없는 노드들을 leaf노드라고 부름

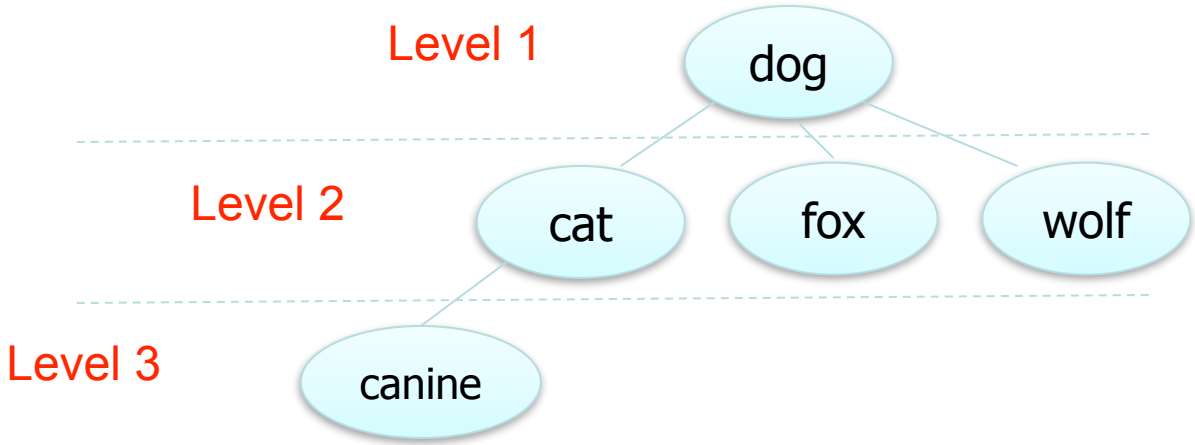
용어: 조상-자손 관계


dog는 canine의 조상(ancestor) 노드이다.
canine은 cat의 자손(descendant)이다.
dog
cat
fox
wolf
canine
부모-자식 관계를 확장한 것이 조상-자손 (ancestor-descendant) 관계이다.

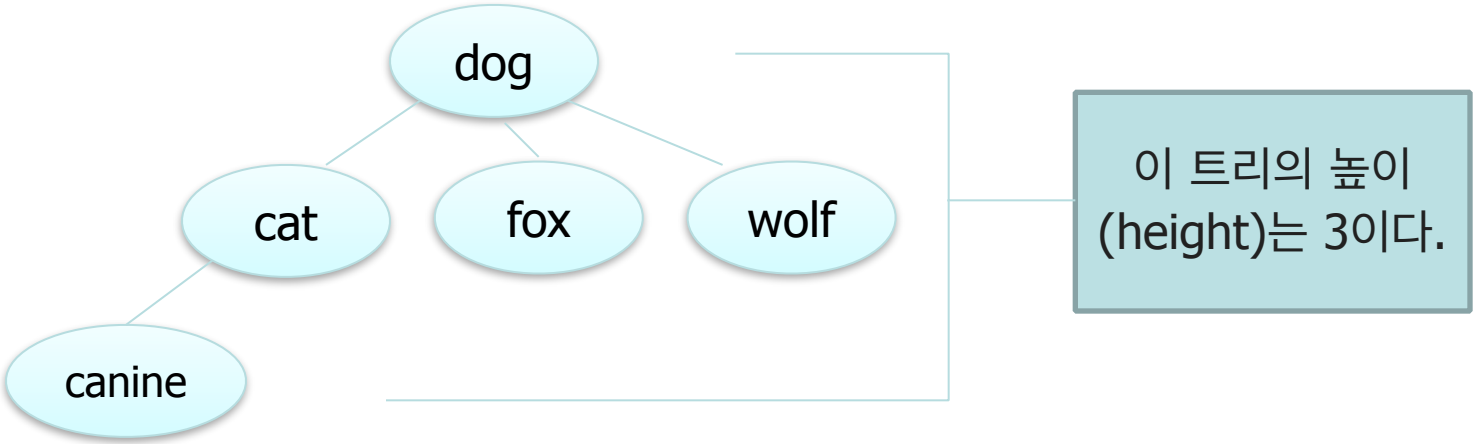
용어: 부트리


dog
cat
fox
wolf
canine
트리에서 어떤 한 노드와 그 노드의 자손들로 이루어진 트리를 부트리(subtree)라고 부른다.

용어: 레벨



용어: 높이

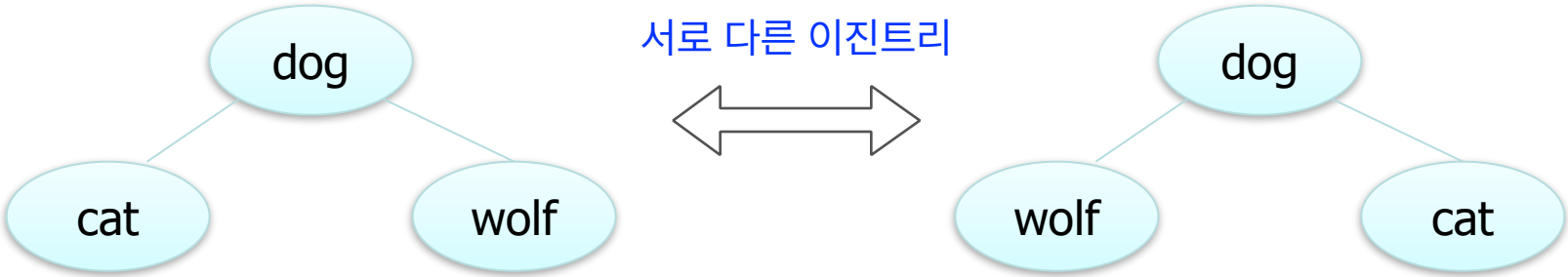


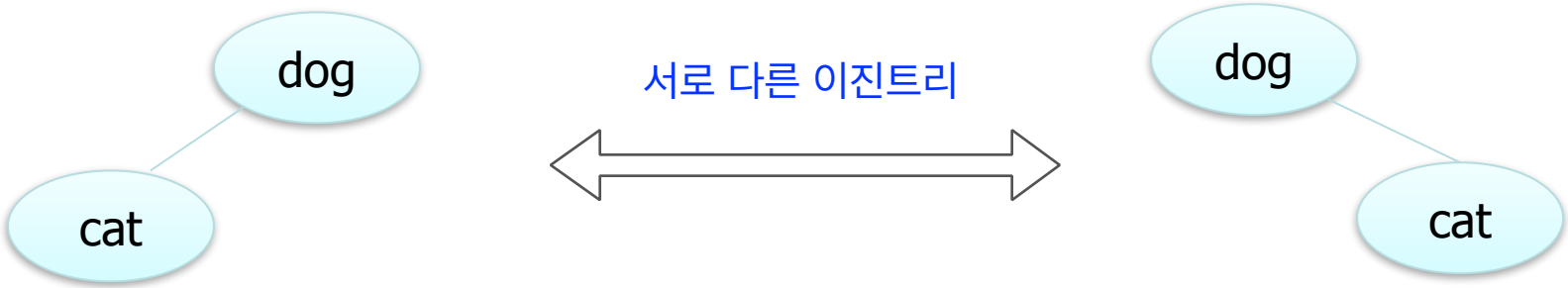
트리의 기본적인 성질

- 노드가 N개인 트리는 항상 N-1개의 링크(link)를 가진다.
- 트리에서 루트에서 어떤 노드로 가는 경로는 유일하다. 또한 임의의 두 노드간의 경로도 유일하다. (같은 노드를 두 번 이상 방문하지 않는다는 조건하에).

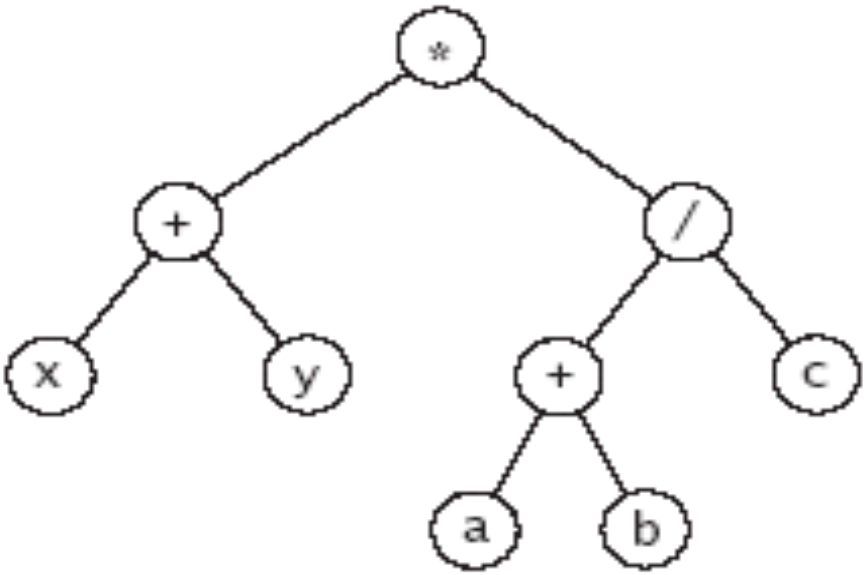
이진 트리 (binary tree)

- 이진 트리에서 각 노드는 최대 2개의 자식을 가진다.
- 각각의 자식 노드는 자신이 부모의 왼쪽 자식인지 오른쪽 자신인지가 지정된다. (자식이 한 명인 경우에도)



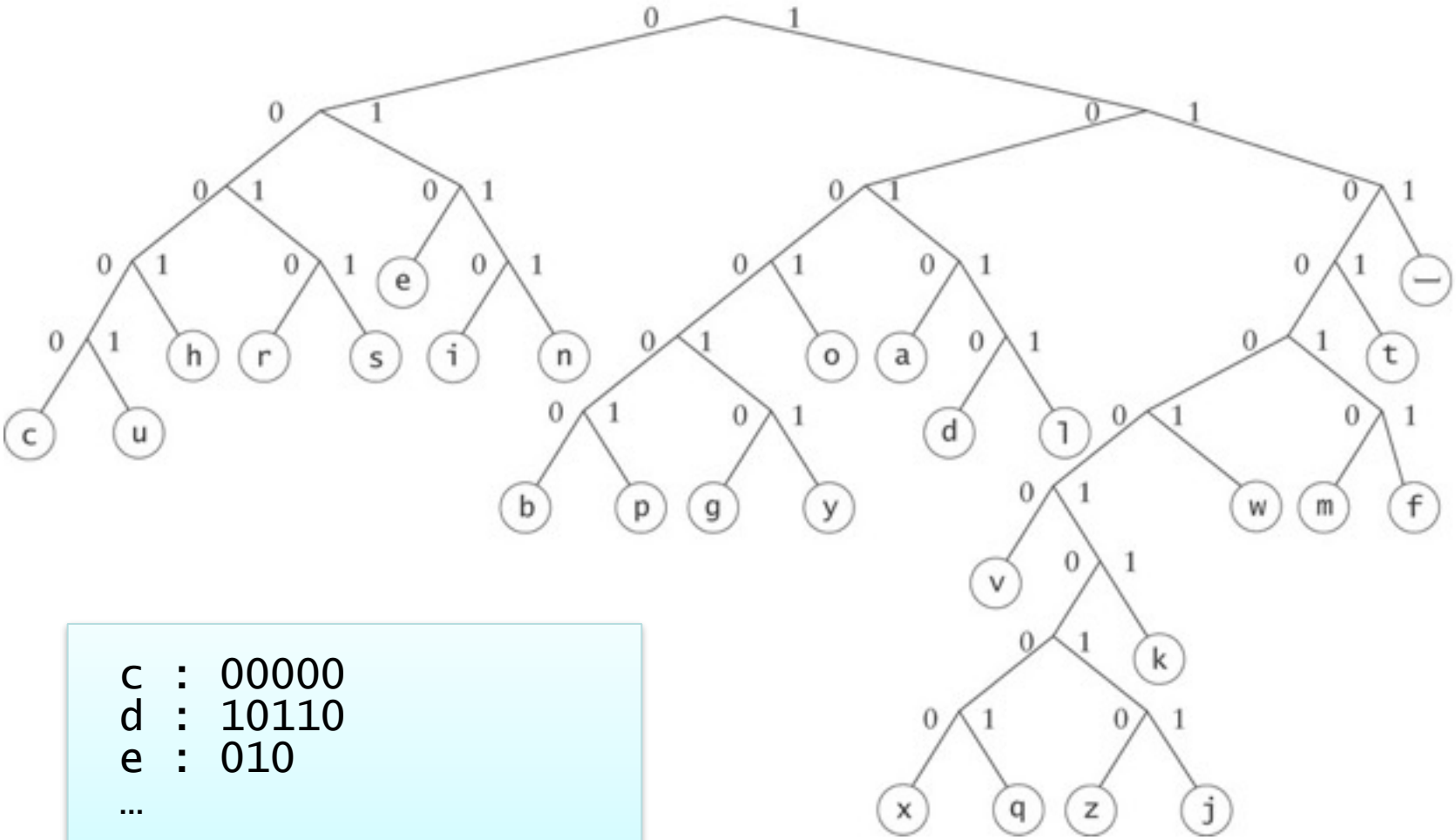


이진 트리 응용의 예: Expression Tree



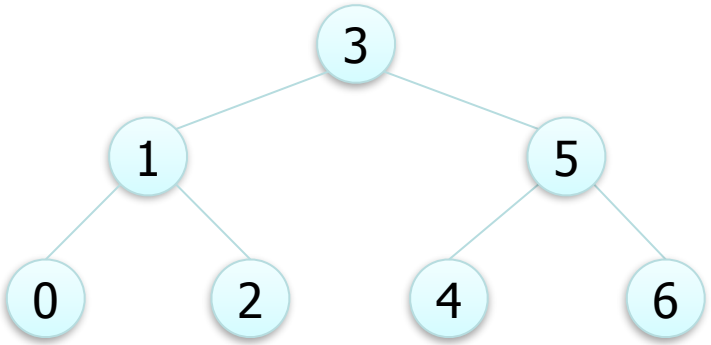

```
(x + y) * ((a + b) / c)
```

이진 트리 응용의 예: Huffman Code

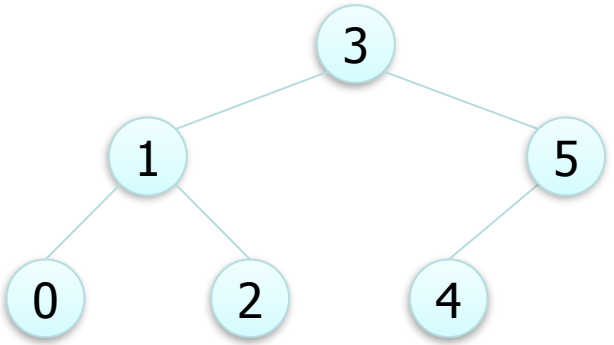



```
c : 00000
d : 10110
e : 010
…
```

Full and Complete Binary Trees



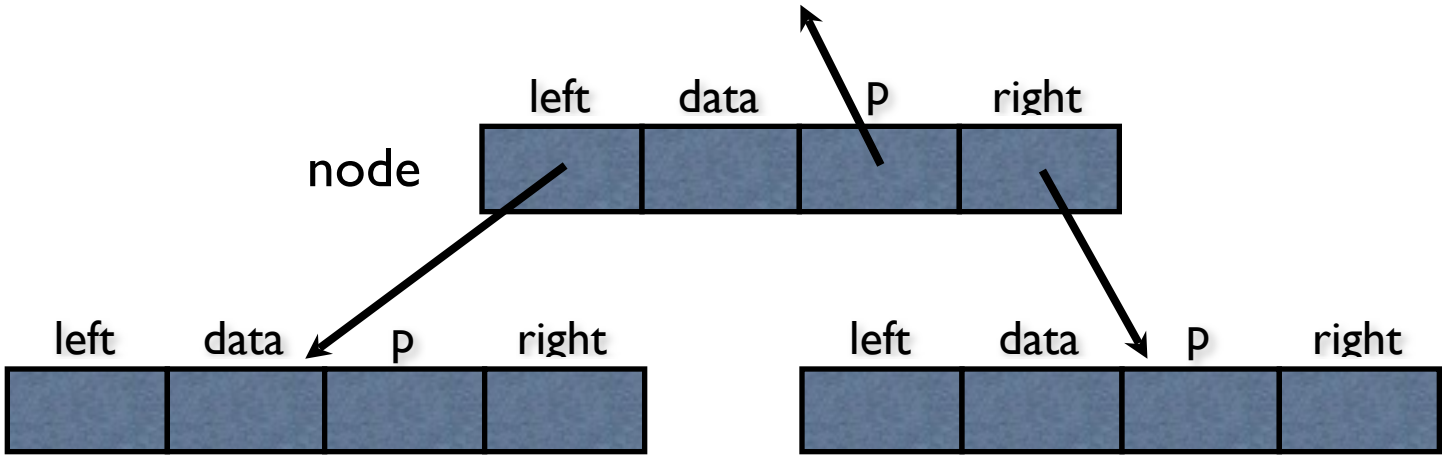
full binary tree



complete binary tree

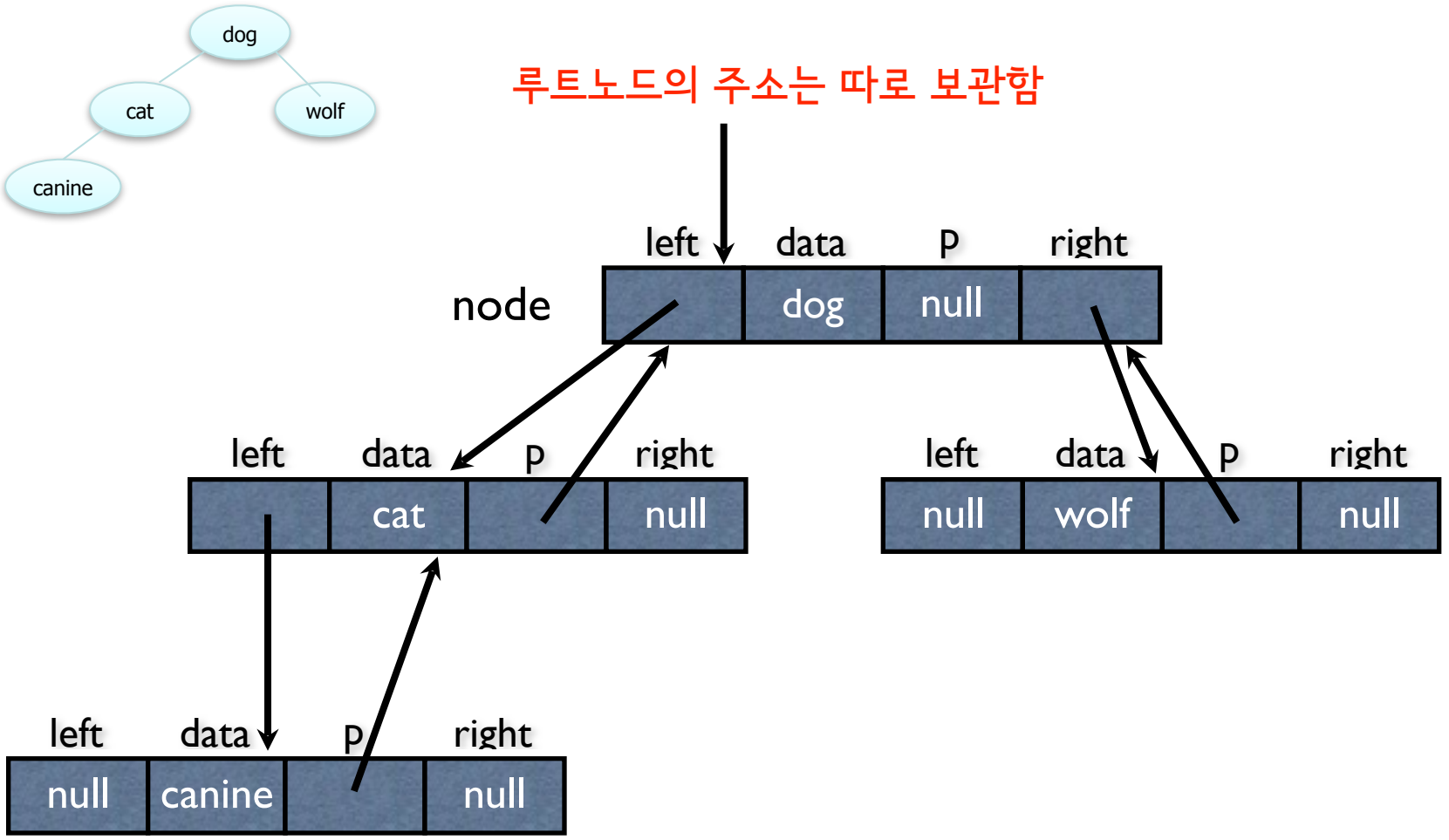
- 높이가 h인 full binary tree는 2^h-1개의 노드를 가진다.
- 노드가 N개인 full 혹은 complete 이진 트리의 높이는 O(logN)이다. 노드가 N개인 이진트리의 높이는 최악의 경우 N이 될수도 있다.

- 연결구조(Linked Structure) 표현



각 노드에 하나의 데이터 필드와 왼쪽자식(left), 오른쪽 자식(right),
그리고 부모노드(p)의 주소를 저장
(부모노드의 주소는 반드시 필요한 경우가 아니면 보통 생략 함)

이진트리의 표현

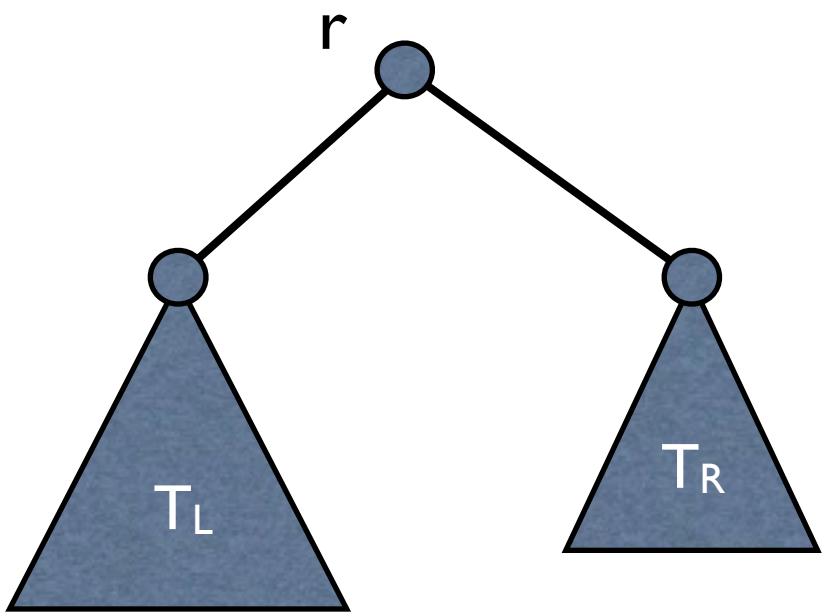

dog
cat
wolf
canine
루트노드의 주소는 따로 보관함
node
left
data
p
right
dog
null
cat
null
null
wolf
null
null
canine
null

이진트리의 순회 (traversal)

- 순회: 이진 트리의 모든 노드를 방문하는 일
- 중순위(inorder) 순회
- 선순위(preorder) 순회
- 후순위(postorder) 순회
- 레벨오더(level-order) 순회

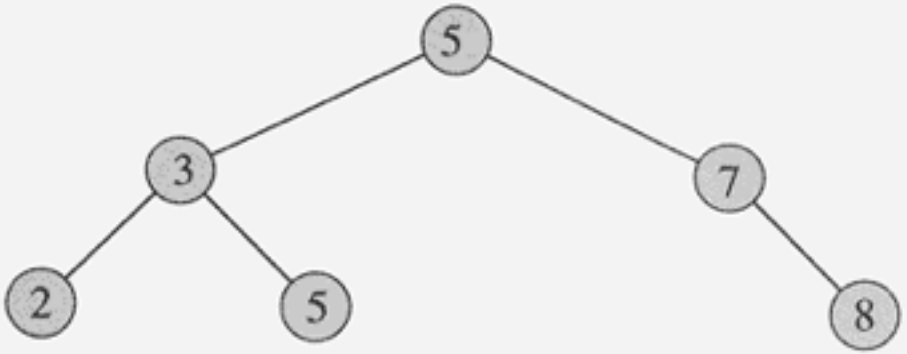
이진트리의 Inorder-순회

1. 먼저 T_L을 inorder로 순회하고,
2. r을 순회하고,
3. T_R을 inorder로 순회.

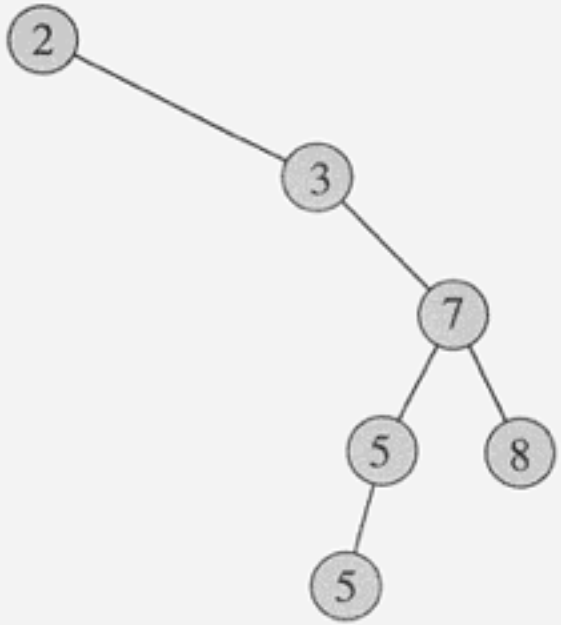


이진트리를 루트노드 r,
왼쪽 부트리 T_L, 그리고
오른쪽 부트리 T_R로 나누어 생각

이진트리의 Inorder-순회



2, 3, 5, 5, 7, 8순으로 방문



2, 3, 5, 5, 7, 8

이진트리의 Inorder-순회

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

x를 루트로 하는 트리를 inorder 순회

시간복잡도 O(n)

Postorder와 Preorder순회

```
PREORDER-TREE-WALK(x)
    if x ≠ NIL
    then   print key[x]
           PRE-ORDER-TREE-WALK(left[x])
           PRE-ORDER-TREE-WALK(right[x]).

POSTORDER-TREE-WALK(x)
    if x ≠ NIL
    then   POST-ORDER-TREE-WALK(left[x])
           POST-ORDER-TREE-WALK(right[x]).
           print key[x]
```

Expression Trees

- Expression 트리를 inorder 순회하면 다음과 같이 출력됨:

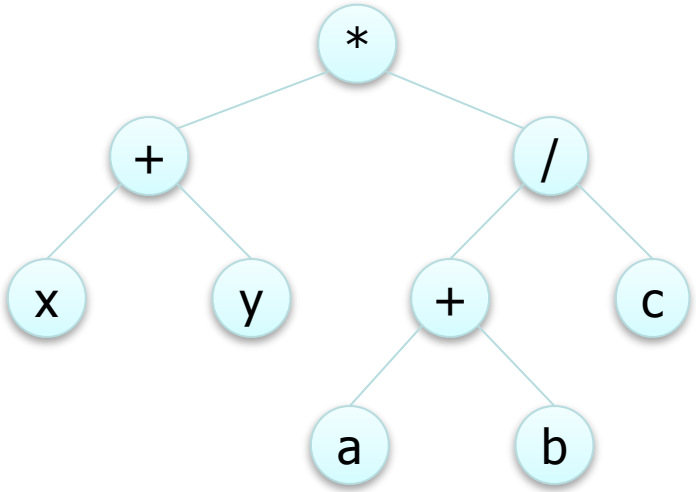
  x + y * a + b / c

- 각 부트리를 순회할 때 시작과 종료시에 괄호를 추가하면 다음과 같이 올바른 수식이 출력됨:

  (x + y) * ((a + b)) / c)

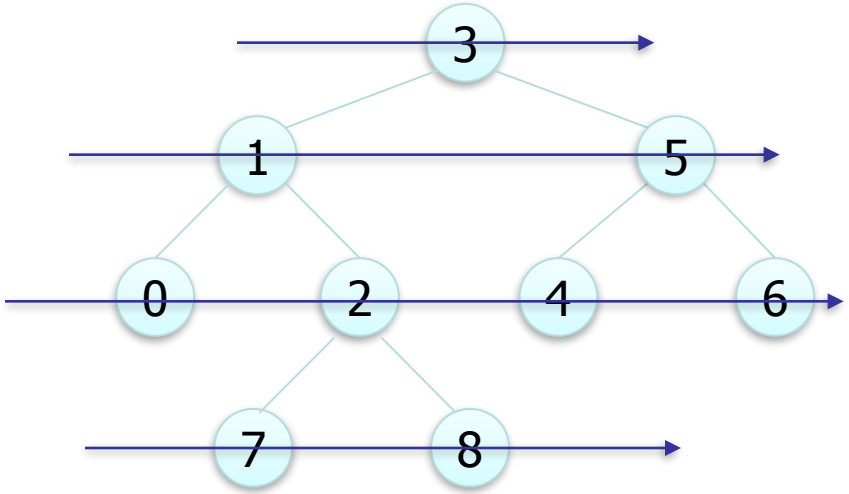
- Postorder 순회하면 후위표기식이 출력됨:

  x y + a b + c / *



Level-Order 순회

- 레벨 순으로 방문, 동일 레벨에서는 왼쪽에서 오른쪽 순서로
- 큐(queue)를 이용하여 구현



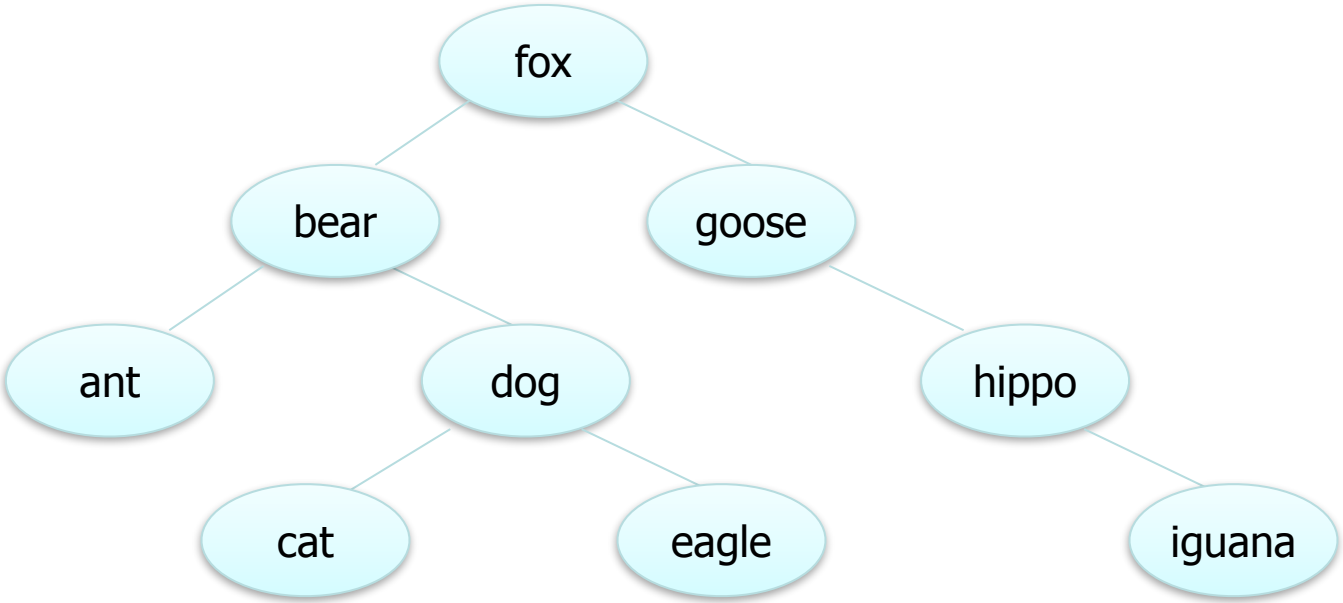
방문순서: 3, 1, 5, 0 ,2, 4, 6, 7, 8

Level-Order 순회

```
LEVEL-ORDER-TREE-TRAVERSAL()
  visit the root;
  Q ← root;                    // Q is a queue
  while Q is not empty do
    v ← dequeue(Q);
    visit children of v;
    enqueue children of v into Q;
  end.
end.
```

연습문제

- 아래 그림과 같이 9개의 노드를 가진 이진트리를 만든 후 inoder, preorder, postorder, 그리고 level-order로 순회하면서 각 노드에 저장된 데이터를 출력하는 프로그램을 작성하라. Java 혹은 C++의 경우 클래스 Node와 클래스 BinaryTree를 작성하고, C의 경우 노드의 구조를 하나의 struct로 정의하라.



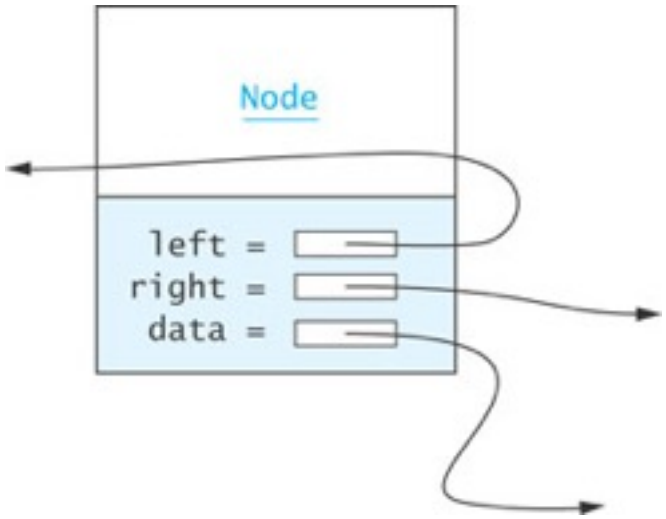
class BinaryTree
in Java

Node<E> Class

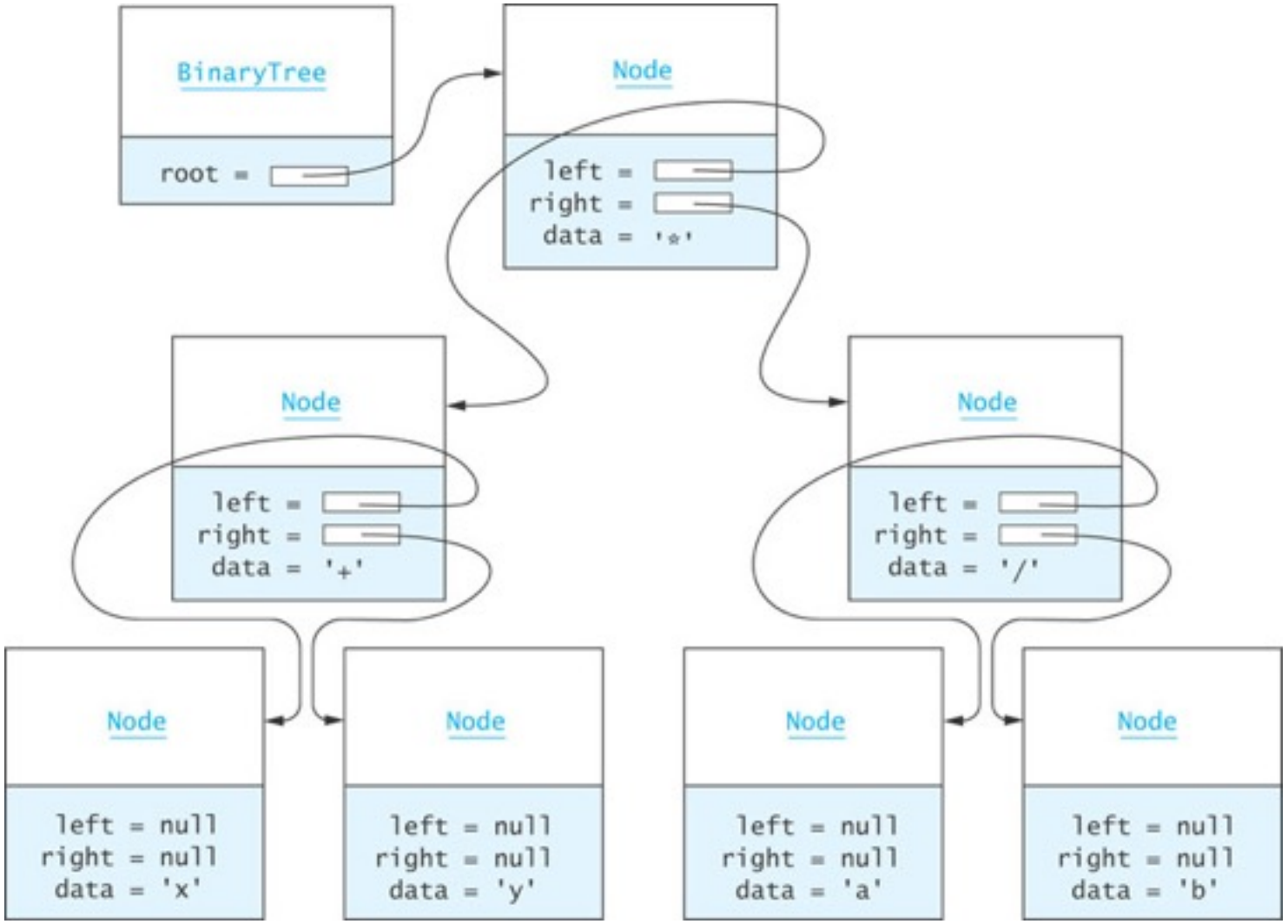
```
protected static class Node<E> {
  protected E data;
  protected Node<E> left;
  protected Node<E> right;

  public Node(E data) {
    this.data = data;
    left = null;
    right = null;
  }

  public String toString() {
    return data.toString();
  }
}
```



BinaryTree<E> Class



BinaryTree<E> Class

```
public class BinaryTree<E> {

    // 앞 페이지의 class Node<E>를 여기에 넣는다.

    protected Node<E> root;     // 루트노드의 주소를 보관한다.

    // 다음 페이지부터 나오는 생성자 및 메서드들을 여기에 넣는다.

}
```

Constructors

```
public BinaryTree() {
    root = null;
}

protected BinaryTree(Node<E> root) {
    this.root = root;
}
```

Constructors

```
public BinaryTree(E data, BinaryTree<E> leftTree,
                  BinaryTree<E> rightTree) {
    root = new Node<E>(data);

    if (leftTree != null)
      root.left = leftTree.root;
    else
      root.left = null;

    if (rightTree != null)
      root.right = rightTree.root;
    else
      root.right = null;
}
```

getLeftSubtree와 getRightSubtree 매서드

```
public BinaryTree<E> getLeftSubtree() {
   if (root != null && root.left != null)
      return new BinaryTree<E>(root.left);
   else
      return null
}

public BinaryTree<E> getLeftSubtree() {

   // getLeftSubtree()와 대칭적

}
```

이진검색트리
Binary Search Tree

Dynamic Set

- 여러 개의 키(key)를 저장
- 다음과 같은 연산들을 지원하는 자료구조
  - INSERT - 새로운 키의 삽입
  - SEARCH - 키 탐색
  - DELETE - 키의 삭제
- 예: 심볼 테이블

called Dynamic Set,
Dictionary,
or Search Structure

다양한 방법들

- 정렬된 혹은 정렬되지 않은 배열 혹은 연결 리스트를 사용할 경우 INSERT, SEARCH, DELETE 중 적어도 하나는 O(n)
- 이진탐색트리(Binary Search Tree), 레드-블랙 트리, AVL-트리 등의 트리에 기반한 구조들
- Direct Address Table, 해쉬 테이블 등

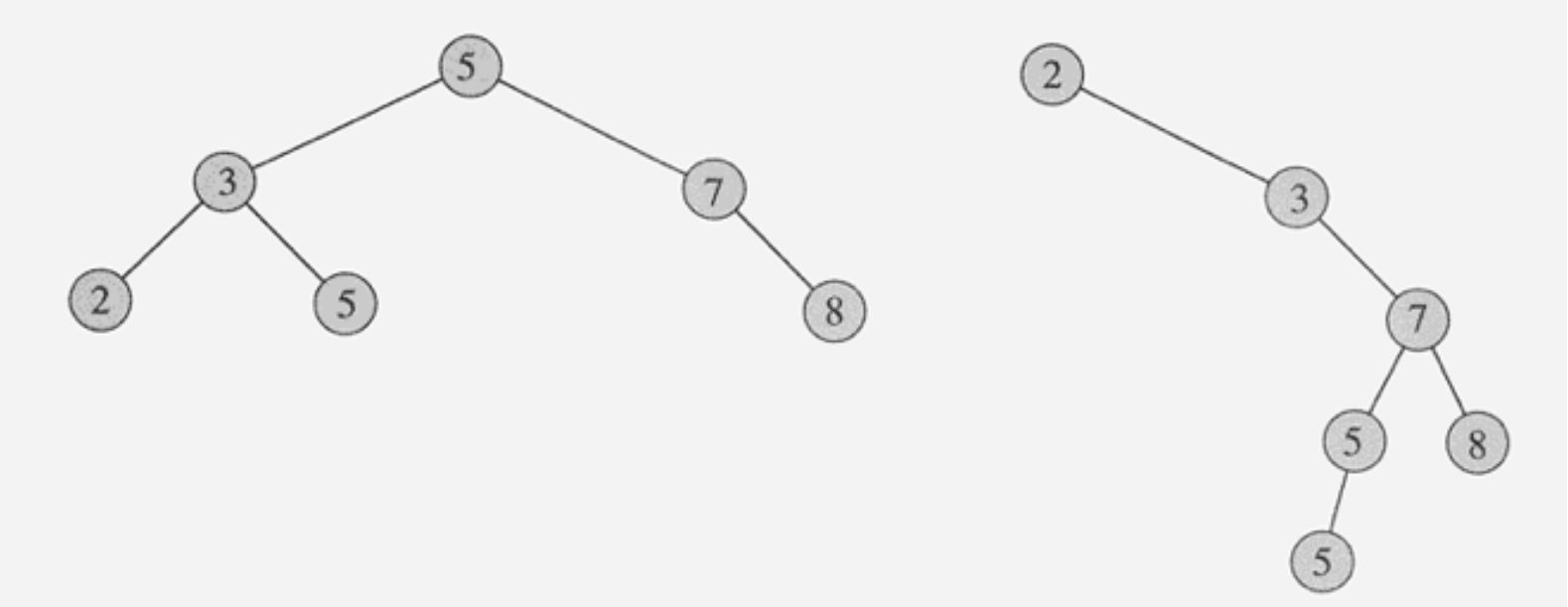
검색트리

- Dynamic set을 트리의 형태로 구현
- 일반적으로 SEARCH, INSERT, DELETE 연산이 트리의 높이(height)에 비례하는 시간복잡도를 가짐
- 이진검색트리(Binary Search Tree), 레드-블랙 트리(red-black tree), B-트리 등

이진검색트리 (BST)

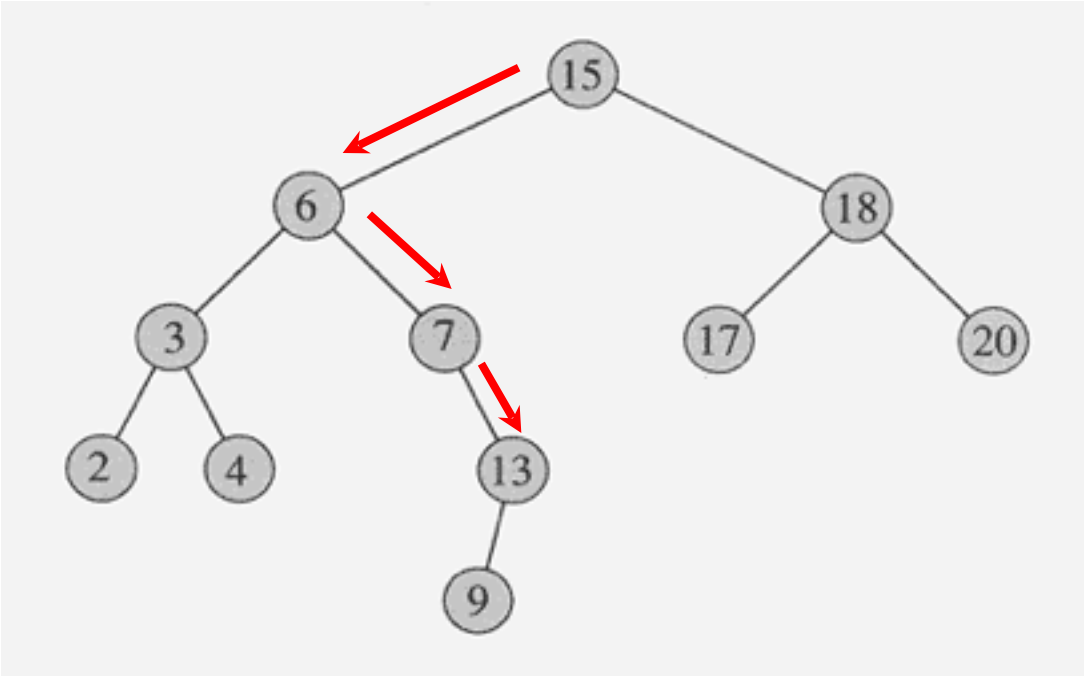
- 이진 트리이면서
- 각 노드에 하나의 키를 저장
- 각 노드 v에 대해서 그 노드의 왼쪽 부트리(subtree)에 있는 키들은 key[v]보다 작거나 같고, 오른쪽 부트리에 있는 값은 크거나 같다.

이진검색트리 (BST)



BST의 예

SEARCH



키 13을 검색: 15 → 6 → 7 → 13

```
TREE-SEARCH(x, k)
1   if x = NIL or k = key[x]
2       then return x
3   if k < key[x]
4       then return TREE-SEARCH(left[x], k)
5       else  return TREE-SEARCH(right[x], k)
```

시간복잡도: O(h), 여기서 h는 트리의 높이

SEARCH - Iterative Version

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

시간복잡도: O(h), 여기서 h는 트리의 높이

```
TREE-MINIMUM(x)
1   while left[x] ≠ NIL
2         do x ← left[x]
3   return x
```

최소값은 항상 가장 왼쪽 노드에 존재

시간복잡도: O(h)

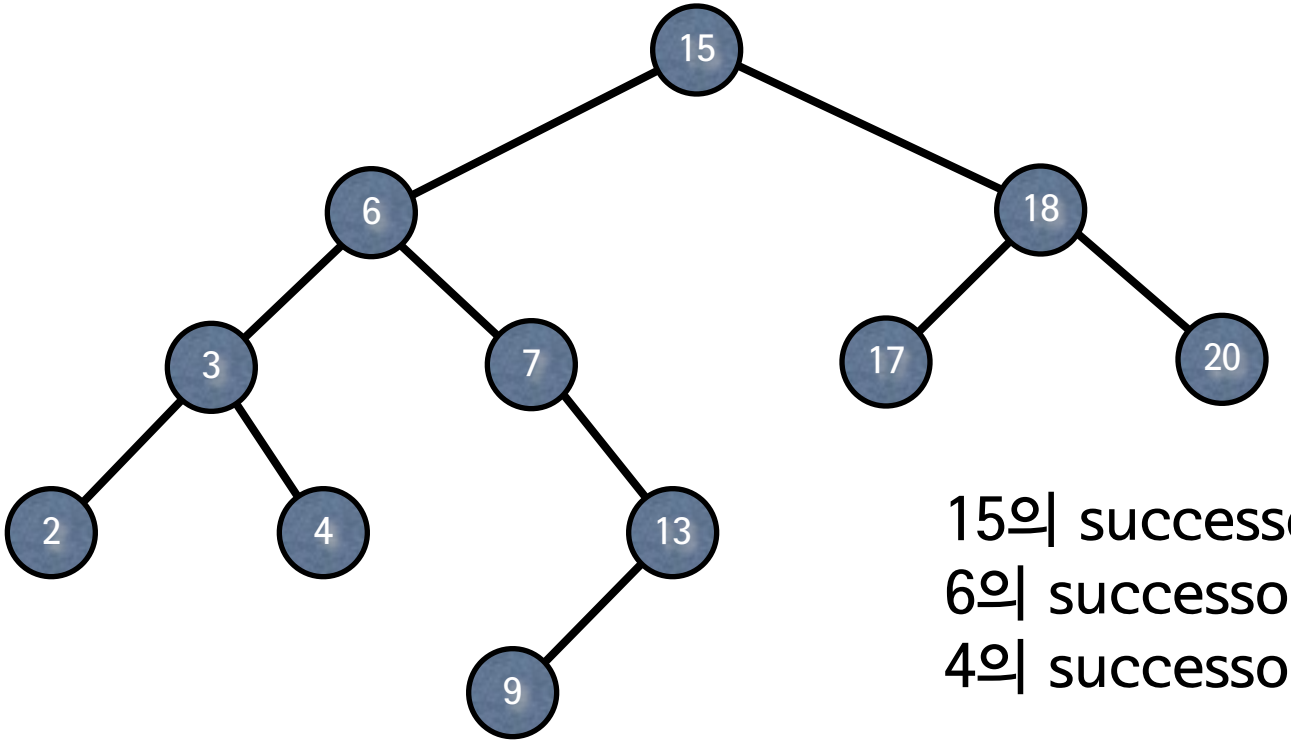
```
TREE-MAXIMUM(x)
1   while right[x] ≠ NIL
2         do x ← right[x]
3   return x
```

최대값은 항상 가장 오른쪽 노드에 존재

시간복잡도: O(h)

Successor

- 노드 x의 successor란 key[x]보다 크면서 가장 작은 키를 가진 노드
- 모든 키들이 서로 다르다고 가정



15의 successor는 17
6의 successor는 7
4의 successor는 6

Successor

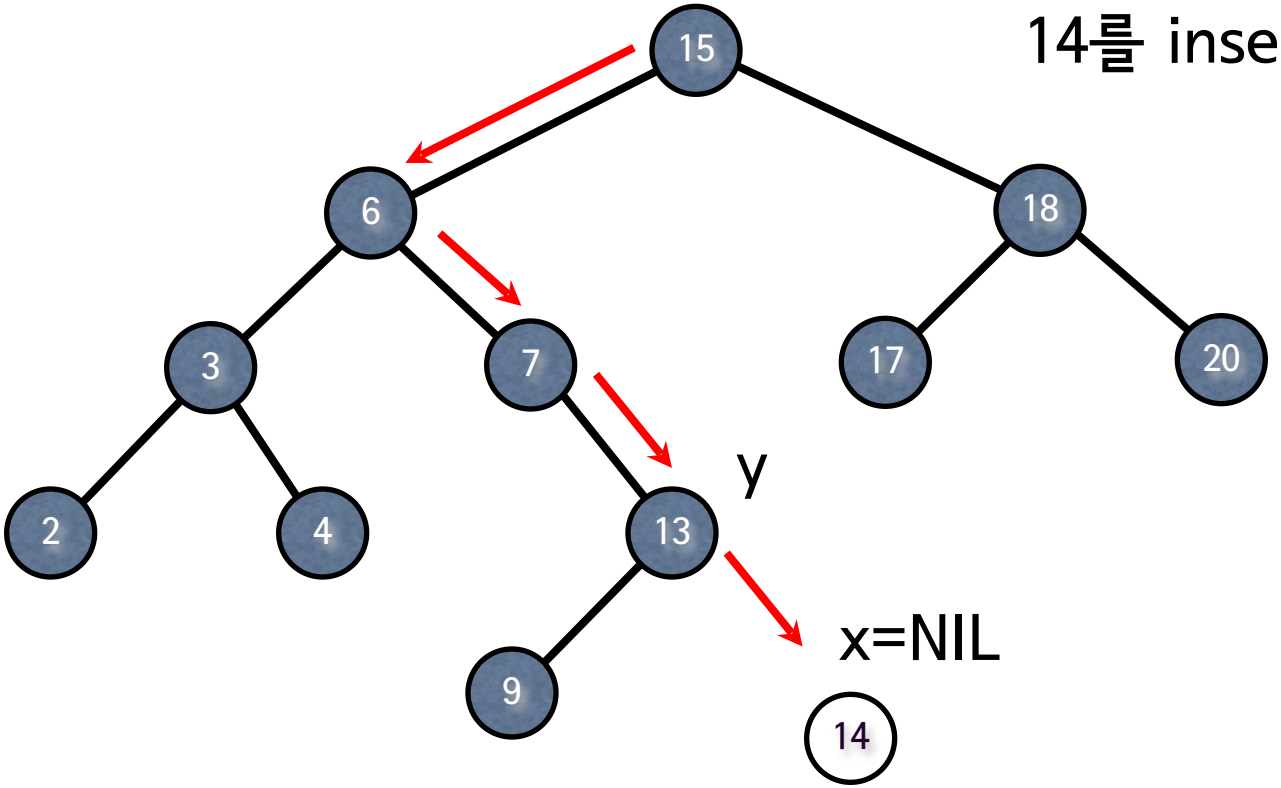
- 3가지 경우
  - 노드 x의 오른쪽 부트리가 존재할 경우, 오른쪽 부트리의 최소값.
  - 오른쪽 부트리가 없는 경우, 어떤 노드 y의 왼쪽 부트리의 최대값이 x가 되는 그런 노드 y가 x의 successor
    - 부모를 따라 루트까지 올라가면서 처음으로 누군가의 왼쪽 자식이 되는 노드
  - 그런 노드 y가 존재하지 않을 경우 successor가 존재하지 않음 (즉, x가 최대값)

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

시간복잡도: O(h)

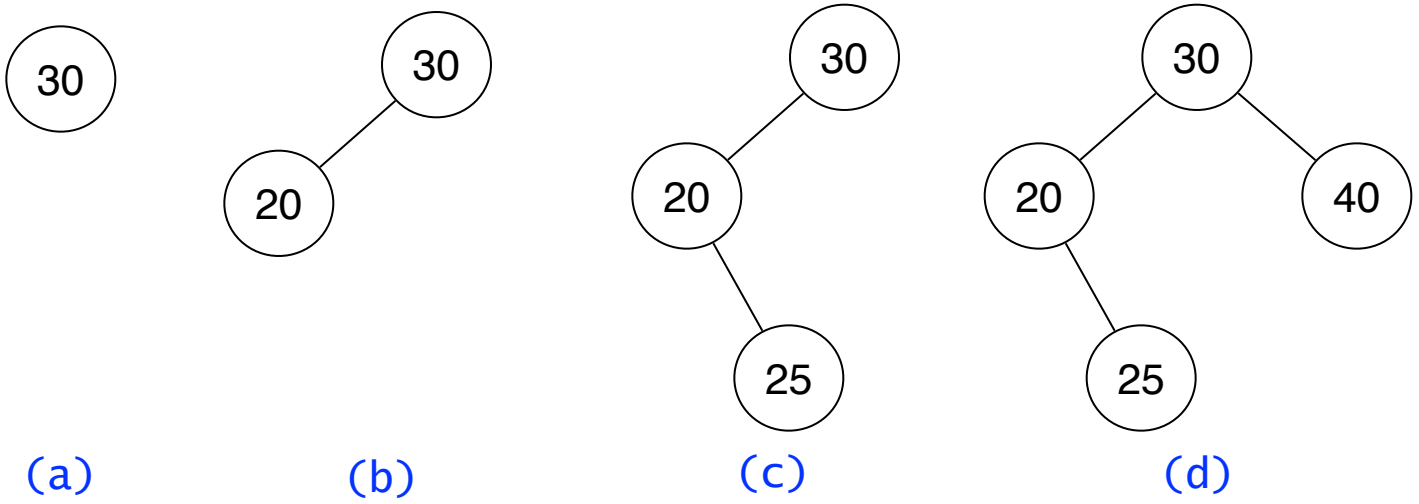
Predecessor

- 노드 x의 predecessor란 key[x]보다 작으면서 가장 큰 키를 가진 노드
- Successor와 반대

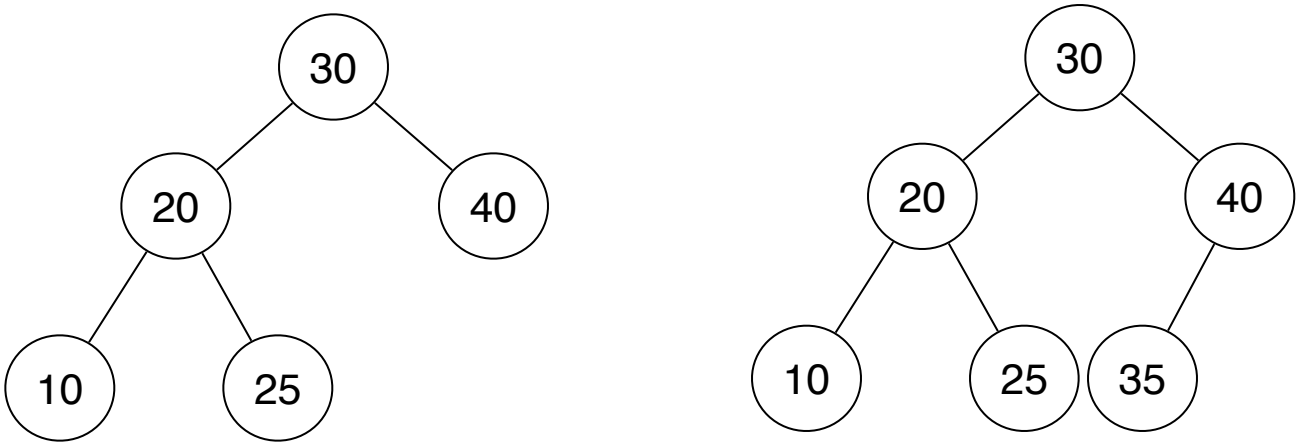


2개의 포인터 x, y를 사용

INSERT



(a) (b) (c) (d)

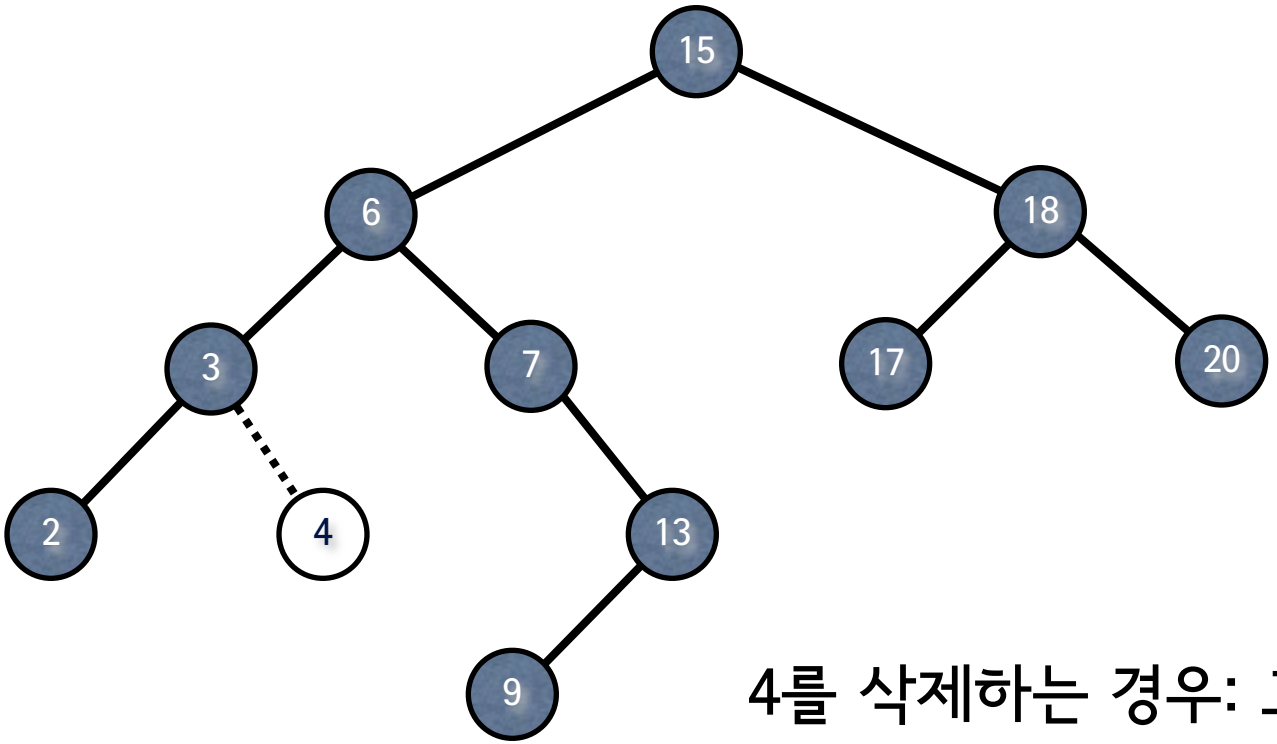


(e) (f)

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ Tree T was empty
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

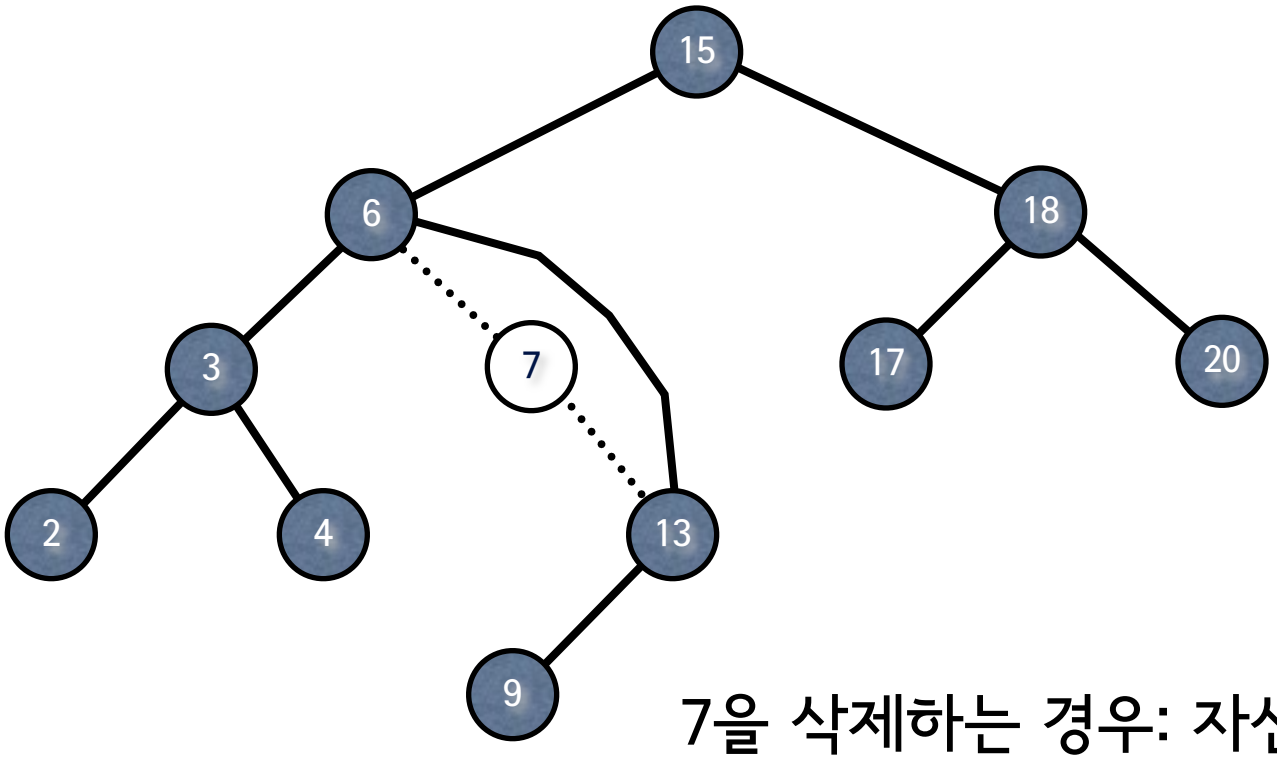
시간복잡도: O(h)

- Case 1: 자식노드가 없는 경우



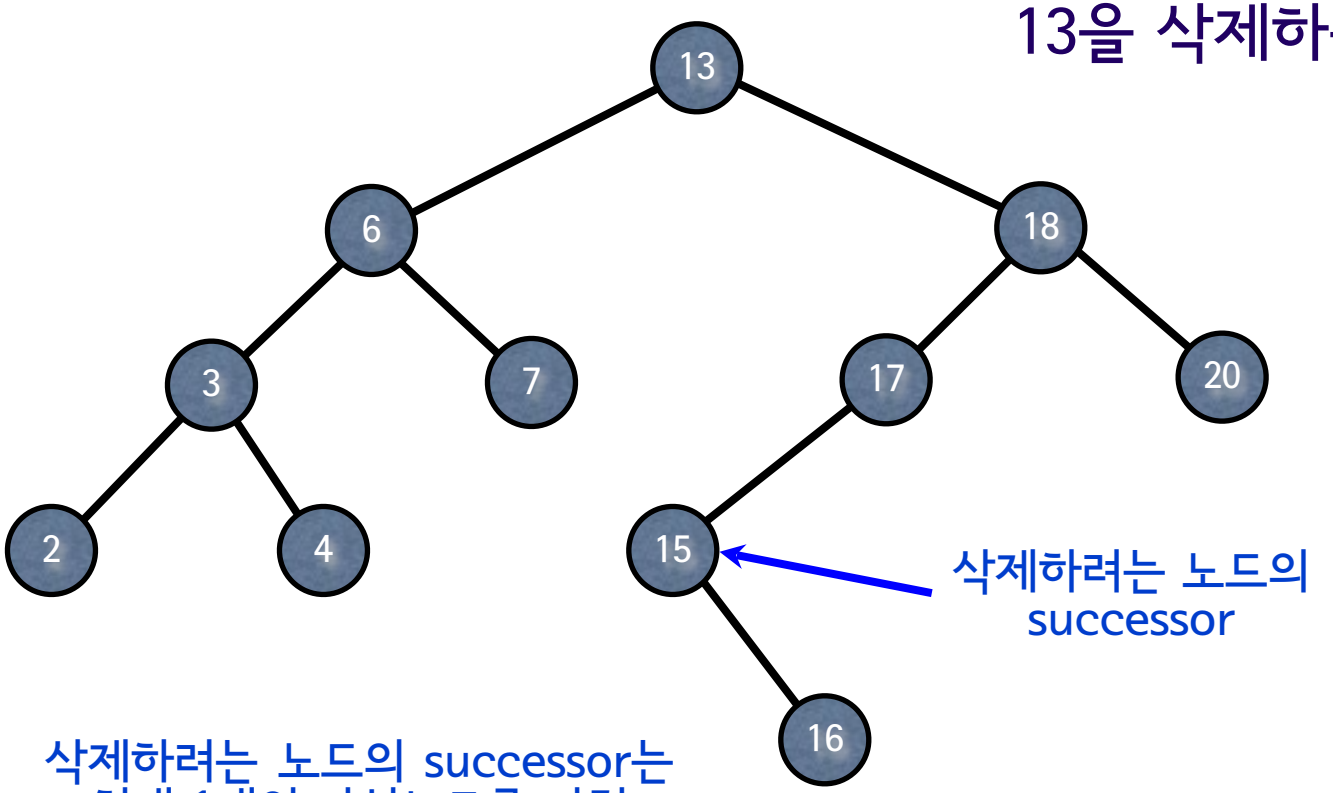
4를 삭제하는 경우: 그냥 삭제

- Case 2: 자식노드가 1개인 경우

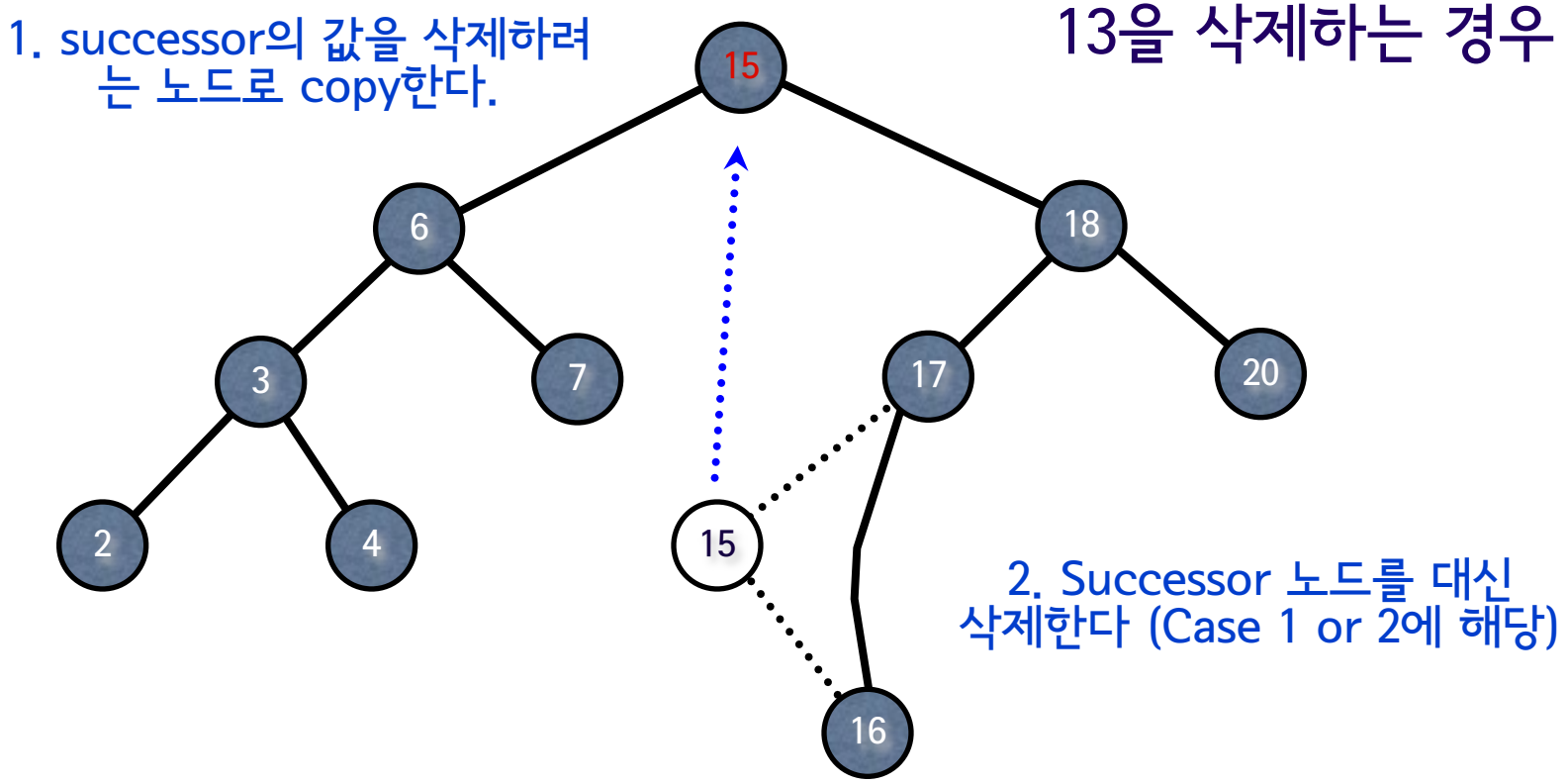


7을 삭제하는 경우: 자신의 자식노드를 원래 자신의 위치로

- Case 3: 자식노드가 2개인 경우



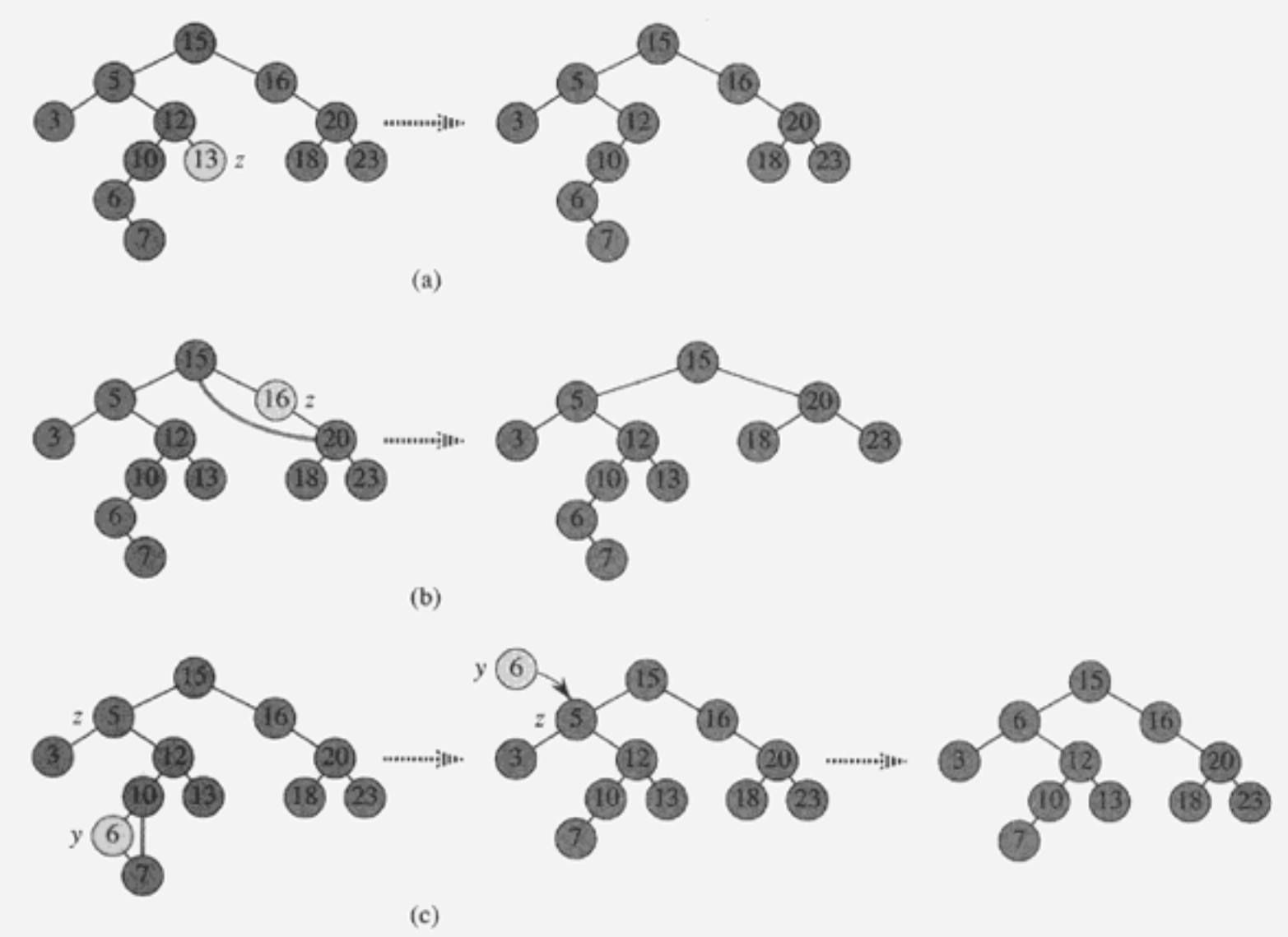
- Case 3: 자식노드가 2개인 경우



```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```

시간복잡도: O(h)

삭제의 예


15
5
16
3
12
20
10
13
z
18
23
6
7
(a)
15
5
16
3
12
20
10
18
23
6
7
15
16
z
5
20
3
12
18
23
10
13
6
7
(b)
15
z
5
16
3
12
20
10
13
18
23
y
6
7
y
6
15
z
5
16
3
12
20
10
13
18
23
7
15
6
16
3
12
20
10
13
18
23
7
(c)

- 각종 연산의 시간복잡도 O(h)
- 그러나, 최악의 경우 트리의 높이 h=O(n)
- 균형잡힌(balanced) 트리
  - 레드-블랙 트리 등
  - 키의 삽입이나 삭제시 추가로 트리의 균형을 잡아줌으로써 높이를 $O(\log_2 n)$으로 유지

연습문제

- 이진검색트리를 구현하라. SEARCH, INSERT, DELETE 기능을 제공하라. 랜덤 데이터를 이용하여 이 연산들이 제대로 실행됨을 보여라.

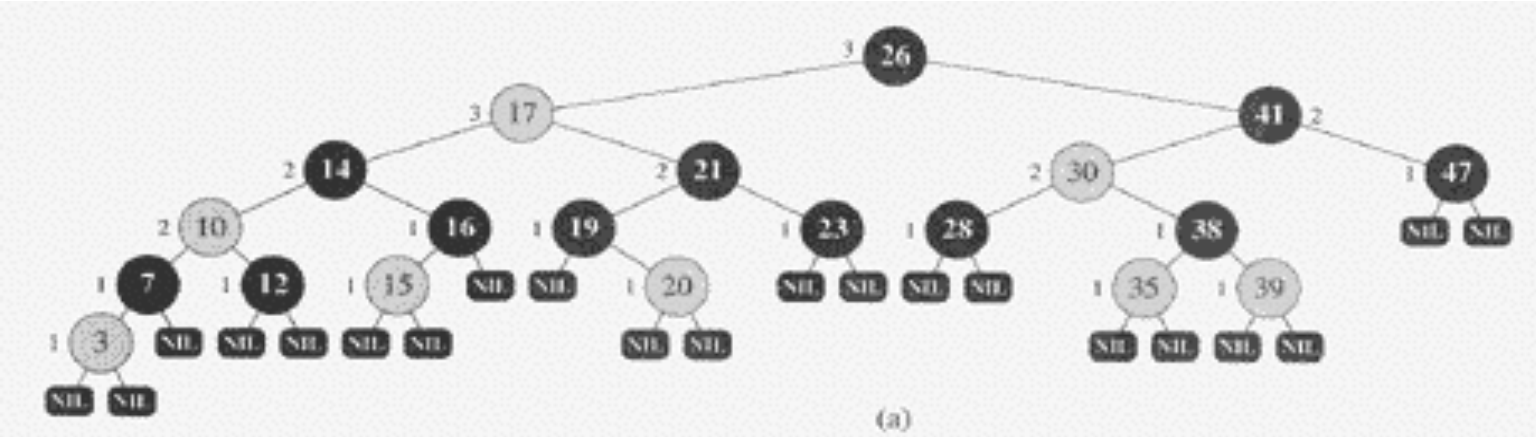
레드블랙트리

Red-Black Tree

레드-블랙 트리

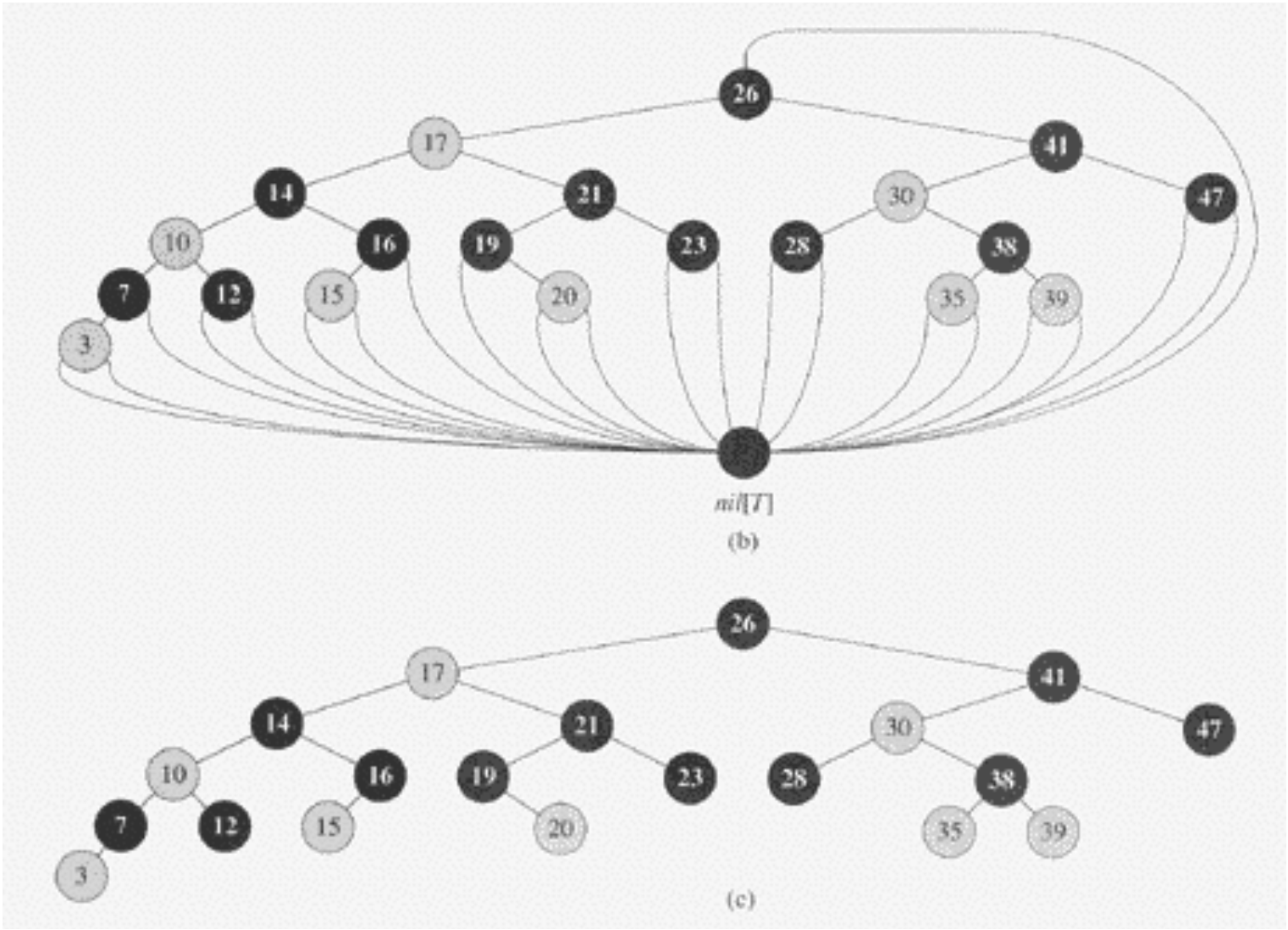
- 이진탐색트리의 일종
- 균형잡힌 트리: 높이가 $O(\log_2 n)$
- SEARCH, INSERT, DELETE 연산을 최악의 경우에도 $O(\log_2 n)$ 시간에 지원

레드-블랙 트리



- 각 노드는 하나의 키(key), 왼쪽자식(left), 오른쪽 자식(right), 그리고 부모노드(p)의 주소를 저장
- 자식노드가 존재하지 않을 경우 NIL 노드라고 부르는 특수한 노드가 있다고 가정
- 따라서 모든 리프노드는 NIL노드
- 루트의 부모도 NIL노드라고 가정
- 노드들은 내부노드와 NIL노드로 분류

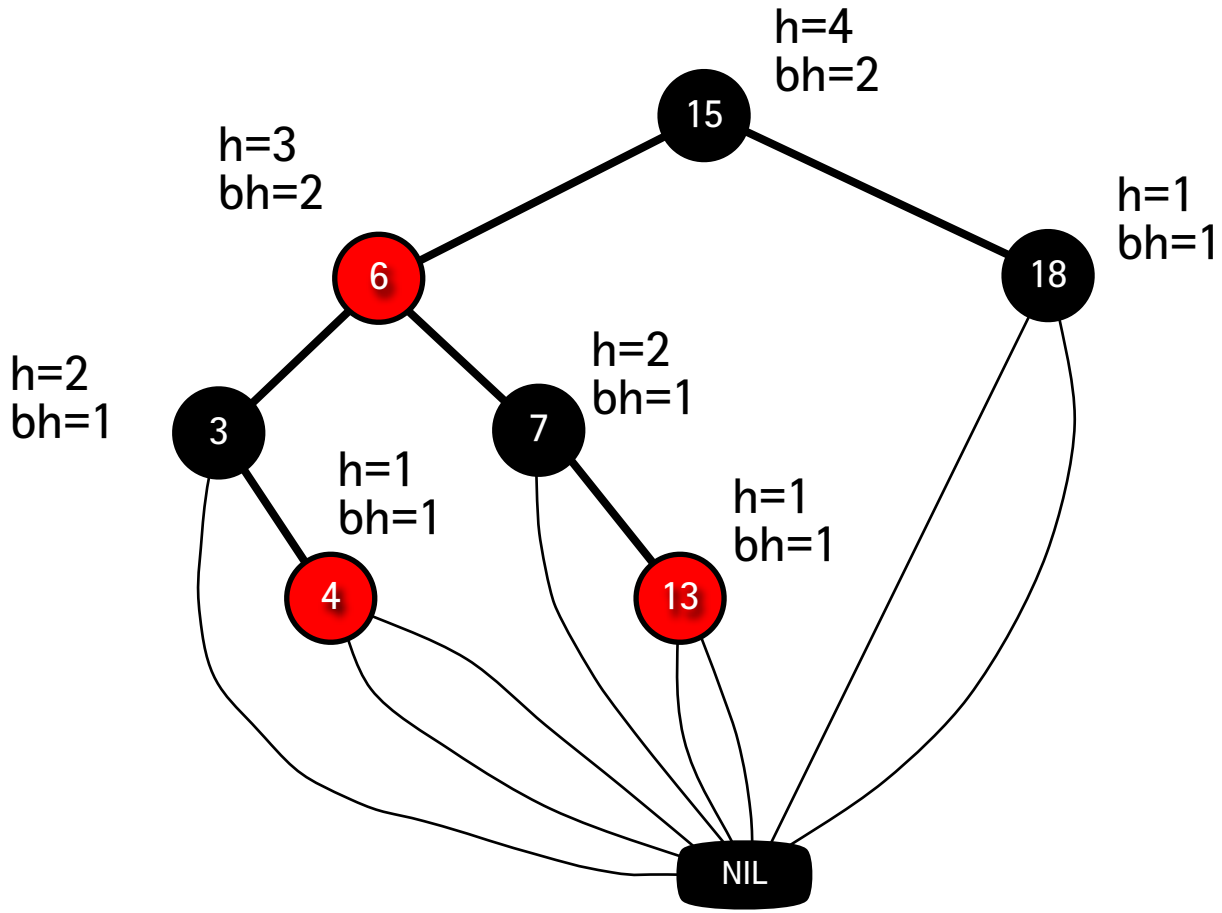
레드-블랙 트리


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
nil[T]
(b)
26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
(c)

레드-블랙 트리: 정의

- 다음의 조건을 만족하는이진탐색트리:
  1. 각 노드는 red 혹은 black이고,
  2. 루트노드는 black이고,
  3. 모든 리프노드(즉, NIL노드)는 black이고,
  4. red노드의 자식노드들은 전부 black이고(즉, red노드는 연속되어 등장하지 않고),
  5. 모든 노드에 대해서 그 노드로부터 자손인 리프노드에 이르는 모든 경로에는 동일한 개수의 black노드가 존재한다.

레드-블랙 트리: 예



레드-블랙 트리의 높이

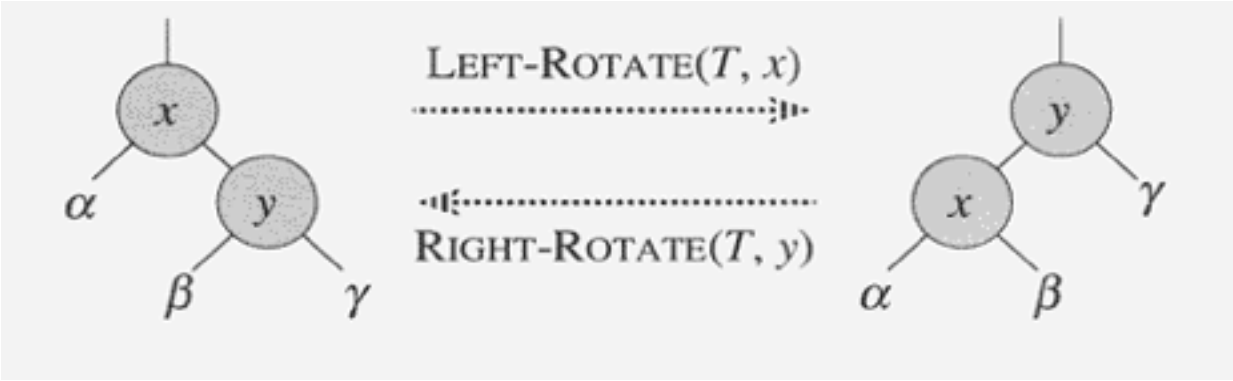
- 노드 x의 높이 h(x)는 자신으로부터 리프노드까지의 가장 긴 경로에 포함된 에지의 개수이다.
- 노드 x의 블랙-높이 bh(x)는 x로부터 리프노드까지의 경로상의 블랙노드의 개수이다 (노드 x 자신은 불포함)

레드-블랙 트리의 높이

- 높이가 h인 노드의 블랙-높이는 $bh \geq h/2$ 이다.
  - 조건4에 의해 레드노드는 연속될수 없으므로 당연
- 노드 x를 루트로하는 임의의 부트리는 적어도 $2^{bh(x)}-1$개의 내부노드를 포함한다(수학적귀납법)
- n개의 내부노드를 가지는 레드블랙트리의 높이는 $2\log_2(n+1)$이하이다.
  - $n \geq 2^{bh}-1 \geq 2^{h/2}-1$ 이므로, 여기서 bh와 h는 각각 루트 노드의 블랙-높이와 높이

Left and Right Rotation

- 시간복잡도 O(1)
- 이진탐색트리의 특성을 유지

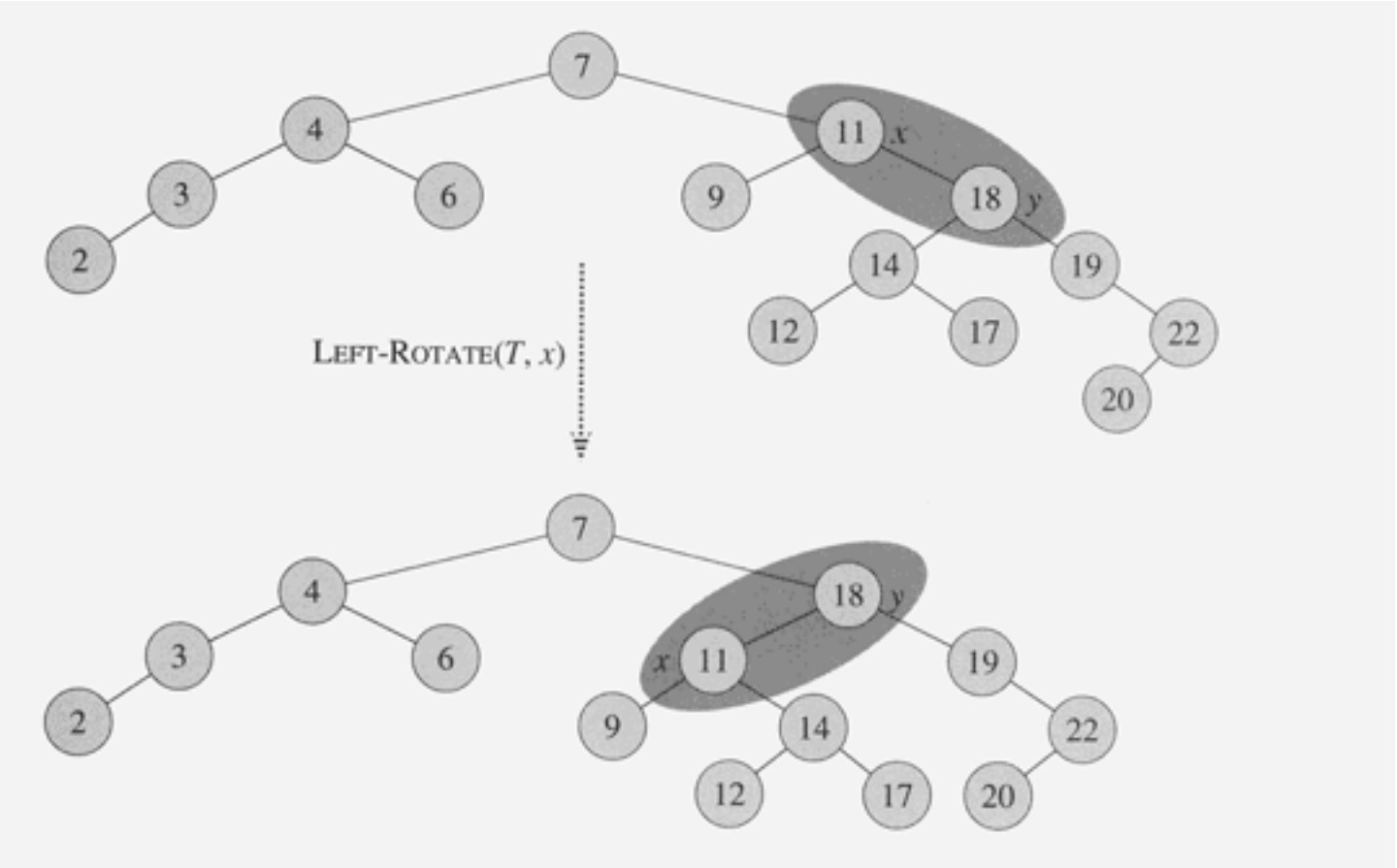


Left Rotation

- y = right[x] ≠NIL이라고 가정
- 루트노드의 부모도 NIL이라고 가정

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
 3  p[left[y]] ← x
 4  p[y] ← p[x]             ▷ Link x's parent to y.
 5  if p[x] = nil[T]
 6     then root[T] ← y
 7     else if x = left[p[x]]
 8             then left[p[x]] ← y
 9             else right[p[x]] ← y
10  left[y] ← x             ▷ Put x on y's left.
11  p[x] ← y
```

Left Rotation



INSERT

```
RB-INSERT(T, z)
 1   y ← nil[T]
 2   x ← root[T]
 3   while x ≠ nil[T]
 4         do y ← x
 5            if key[z] < key[x]
 6               then x ← left[x]
 7               else x ← right[x]
 8   p[z] ← y
 9   if y = nil[T]
10      then root[T] ← z
11      else if key[z] < key[y]
12               then left[y] ← z
13               else right[y] ← z
14   left[z] ← nil[T]
15   right[z] ← nil[T]
16   color[z] ← RED
17   RB-INSERT-FIXUP(T, z)
```

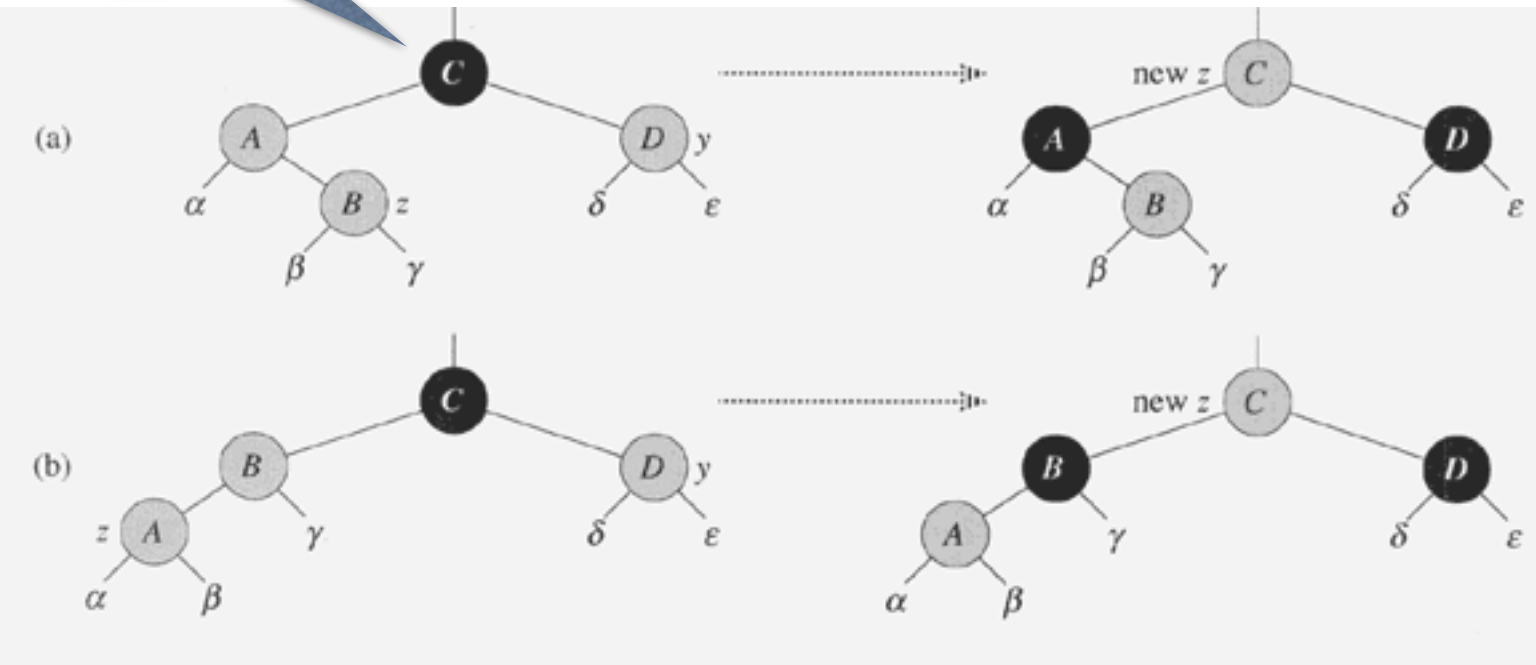
- 보통의 BST에서처럼 노드를 INSERT한다.
- 새로운 노드 z를 red노드로 한다.
- RB-INSERT-FIXUP을 호출한다.

- 위반될 가능성이 있는 조건들
  1. OK.
  2. 만약 z가 루트노드라면 위반, 아니라면 OK.
  3. OK.
  4. z의 부모 p[z]가 red이면 위반.
  5. OK.

- Loop Invariant:
  - z는 red노드
  - 오직 하나의 위반만이 존재한다:
    - 조건2: z가 루트노드이면서 red이거나, 또는
    - 조건4: z와 그 부모 p[z]가 둘 다 red이거나.
- 종료조건:
  - 부모노드 p[z]가 black이되면 종료한다. 조건2가 위반일 경우 z를 블랙으로 바꿔주고 종료한다.

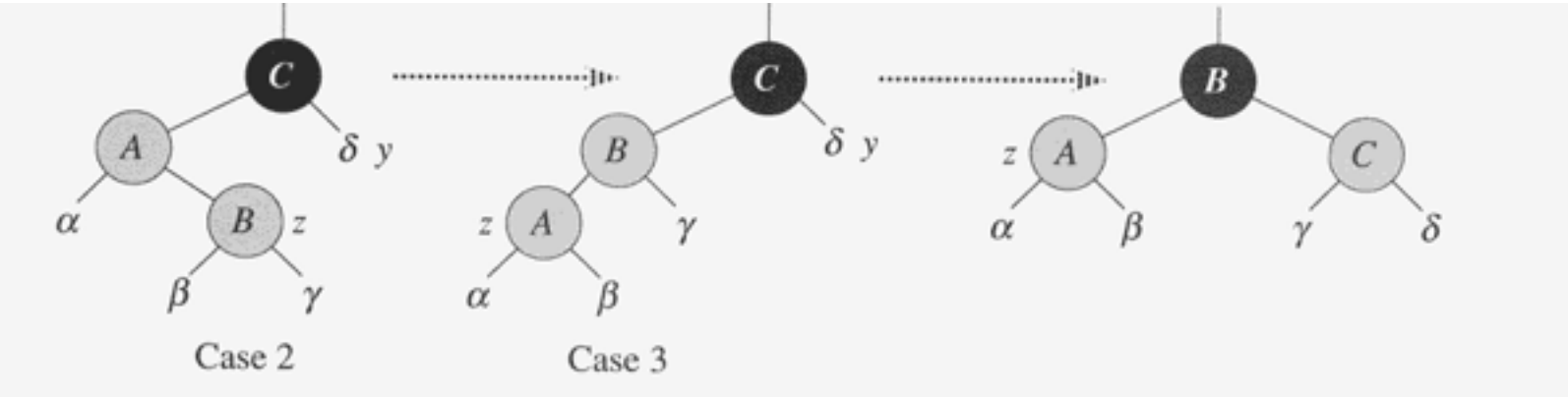
경우1: z의 삼촌이 red

왜 z의 할아버지가 반드시 존재할까?



조건 2와 4 이외의 조건들은 여전히 OK면서
z가 두칸 위로 올라감

경우2와 3: z의 삼촌이 black



- 경우2: z가 오른쪽 자식인 경우
  - p[z]에 대해서 left-rotation한 후 원재 p[z]를 z로
  - 경우3으로
- 경우3: z가 왼쪽 자식인 경우
  - p[z]를 black, p[p[z]]를 red로 바꿈
  - p[p[z]]에 대해서 right-rotation

경우 4,5,6

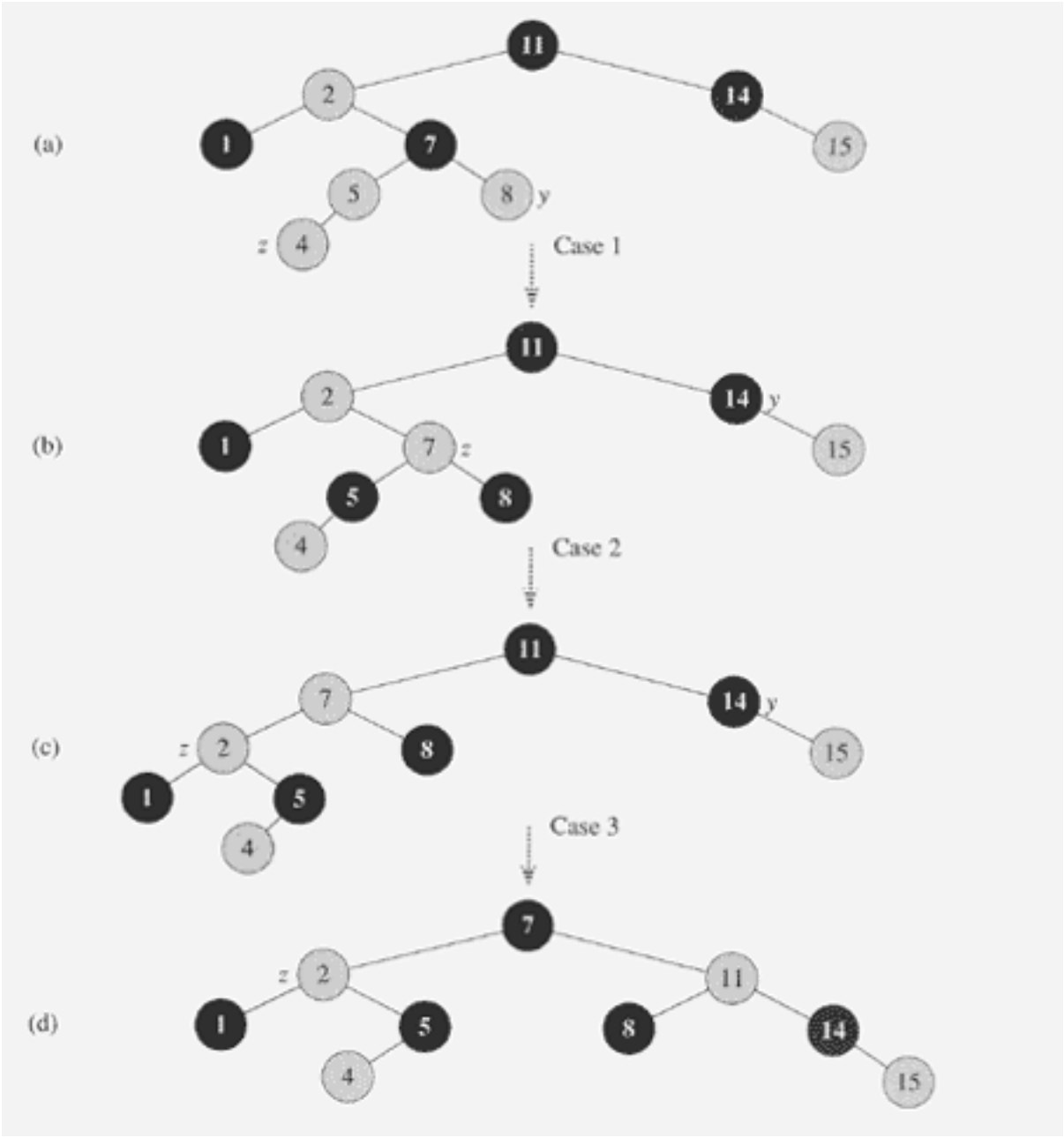
- 경우 1,2,3은 p[z]가 p[p[z]]의 왼쪽자식인 경우들
- 경우 4,5,6은 p[z]가 p[p[z]]의 오른쪽 자식인 경우
  - 경우 1,2,3과 대칭적이므로 생략

RB-INSERT-FIXUP

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2        do if p[z] = left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y] = RED
 5                     then color[p[z]] ← BLACK                    ▷ Case 1
 6                          color[y] ← BLACK                       ▷ Case 1
 7                          color[p[p[z]]] ← RED                   ▷ Case 1
 8                          z ← p[p[z]]                            ▷ Case 1
 9                     else if z = right[p[z]]
10                             then z ← p[z]                       ▷ Case 2
11                                  LEFT-ROTATE(T, z)              ▷ Case 2
12                          color[p[z]] ← BLACK                    ▷ Case 3
13                          color[p[p[z]]] ← RED                   ▷ Case 3
14                          RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
15              else (same as then clause
                      with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

INSERT의 시간복잡도

- BST에서의 INSERT: $O(\log_2 n)$
- RB-INSERT-FIXUP
  - 경우1에 해당할 경우 z가 2레벨 상승
  - 경우 2,3에 해당할 경우 O(1)
  - 따라서 트리의 높이에 비례하는 시간
- 즉, INSERT의 시간복잡도는 $O(\log_2 n)$


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15

DELETE

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

- 보통의 BST에서처럼 DELETE한다.
- 실제로 삭제된 노드 y가 red였으면 종료.
- y가 black이었을 경우RB-DELETE-FIXUP을 호출한다.

여기서 x는 y가 자식이 있었을 경우 그 자식노드, 없었을 경우 NIL노드.

두 경우 모두 p[x]는 원래 p[y]였던 노드

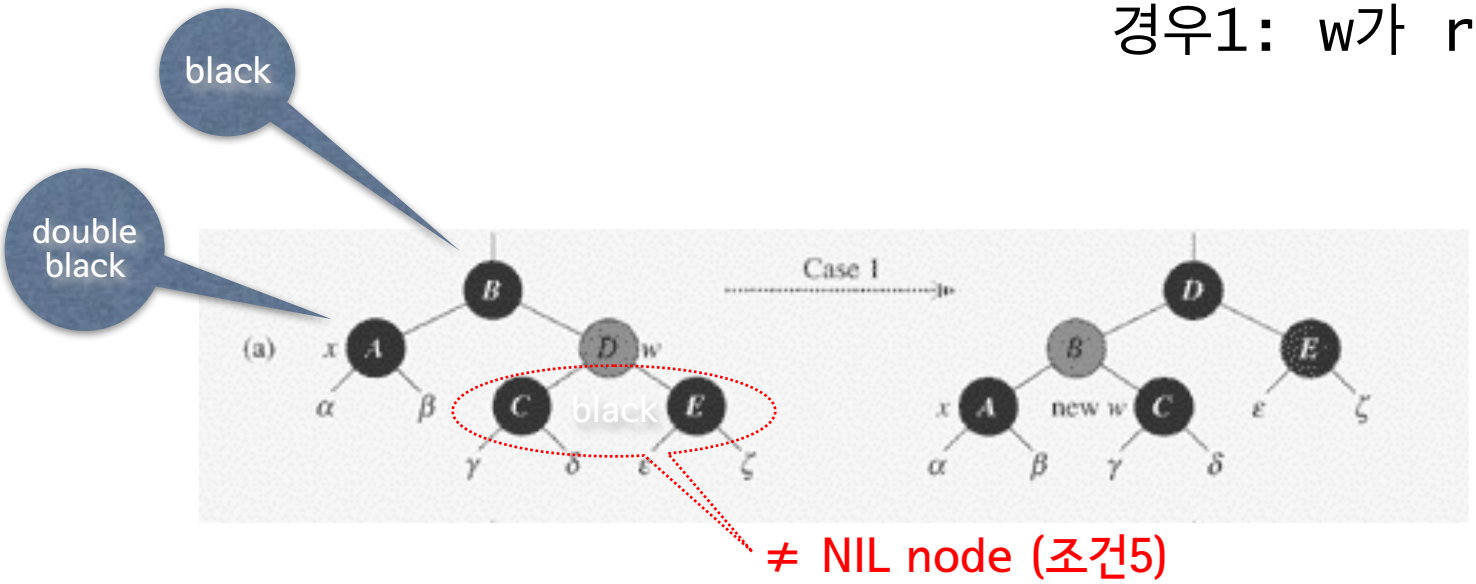
RB-DELETE-FIXUP(T,x)

1. OK.
2. y가 루트였고 x가 red인 경우 위반
3. OK.
4. p[y]와 x가 모두 red일 경우 위반
5. 원래 y를 포함했던 모든 경로는 이제 black노드가 하나 부족
   1) 노드 x에 “extra black”을 부여해서 일단 조건5를 만족
   2) 노드 x는 “double black” 혹은 “red & black”

RB-DELETE-FIXUP(T,x)

- 아이디어:
  - extra black을 트리의 위쪽으로 올려보냄
  - x가 red&black상태가 되면 그냥 black노드로 만들고 끝냄
  - x가 루트가 되면 그냥 extra black을 제거
- Loop Invariant
  - x는 루트가 아닌 double-black노드
  - w는 x의 형제노드
  - w는 NIL 노드가 될수 없음 (아니면 x의 부모에 대해 조건5가 위반)

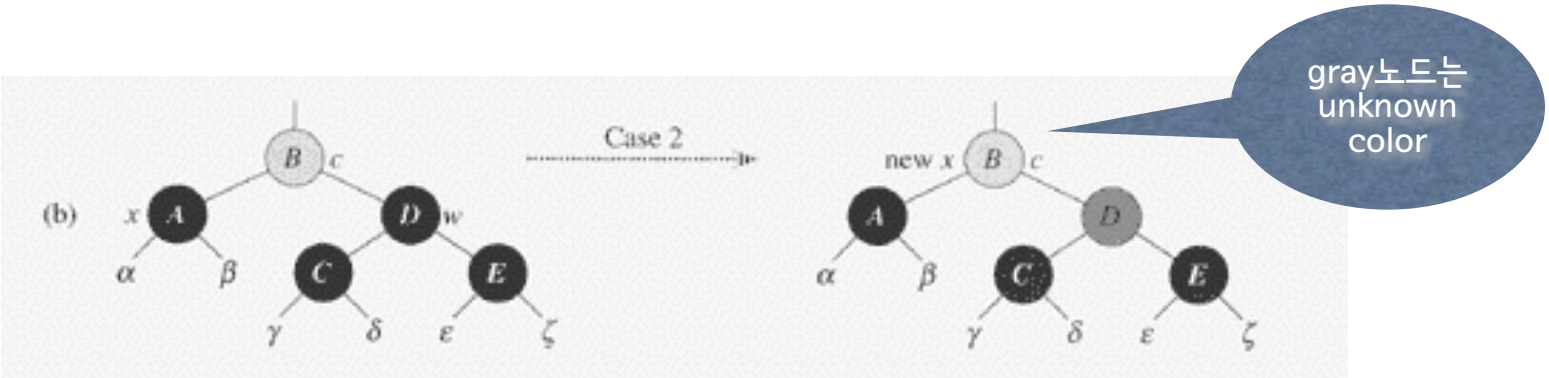
경우1: w가 red인 경우



≠ NIL node (조건5)

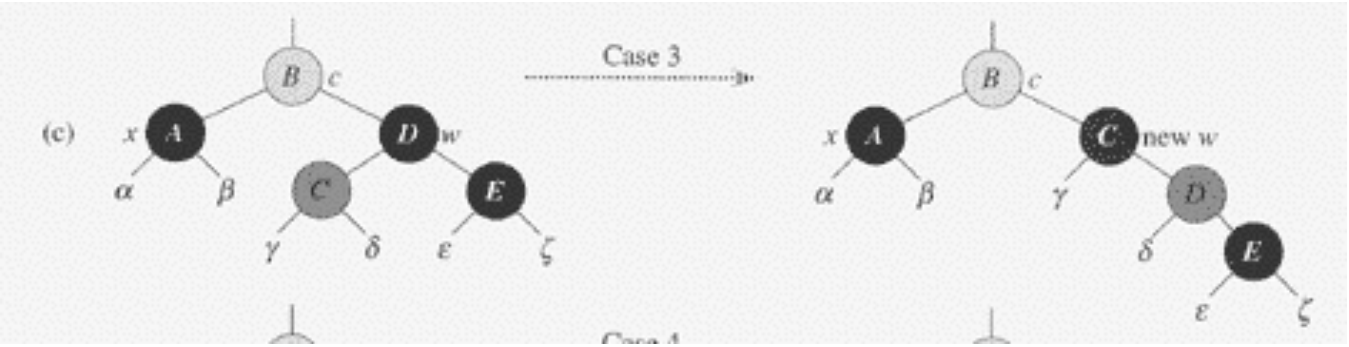
- w의 자식들은 black
- w를 black으로, p[x]를 red로
- p[x]에 대해서 left-rotation적용
- x의 새로운 형제노드는 원래 w의 자식노드, 따라서 black노드
- 경우 2,3,혹은 4에 해당

경우2: w는 black, w의 자식들도 black



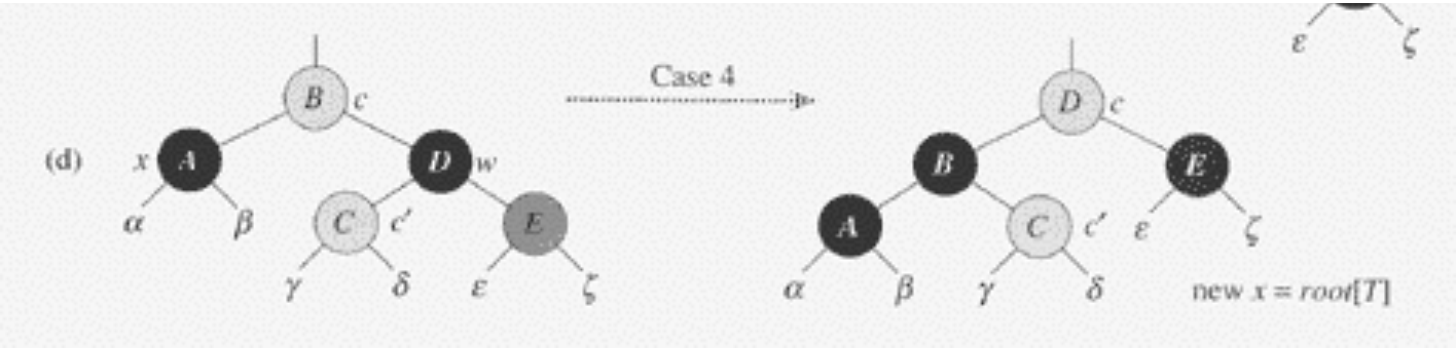
- x의 extra-black을 뺏고, w를 red로 바꿈.
- p[x]에게 뺏은 extra-black을 준다.
- p[x]를 새로운 x로 해서 계속.
- 만약 경우1에서 이 경우에 도달했다면 p[x]는 red였고, 따라서 새로운 x는 red&black이 되어서 종료

경우3: w는 black, w의 왼쪽자식이 red



- w를 red로, w의 왼자식을 black으로
- w에 대해서 right-rotation적용
- x의 새로운 형제 w는 오른자식이 red: 경우4에 해당

경우4: w는 black, w의 오른쪽자식이 red

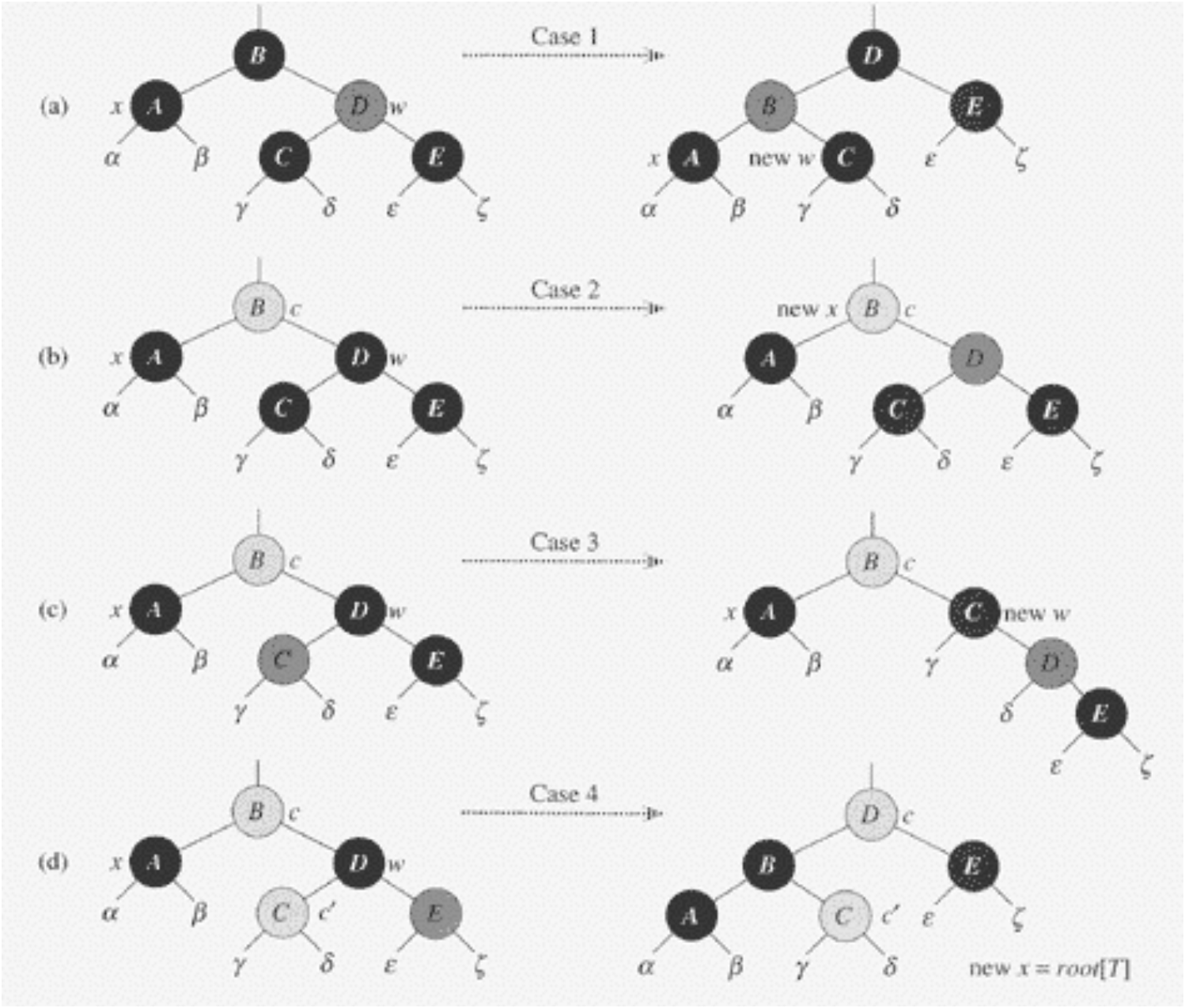


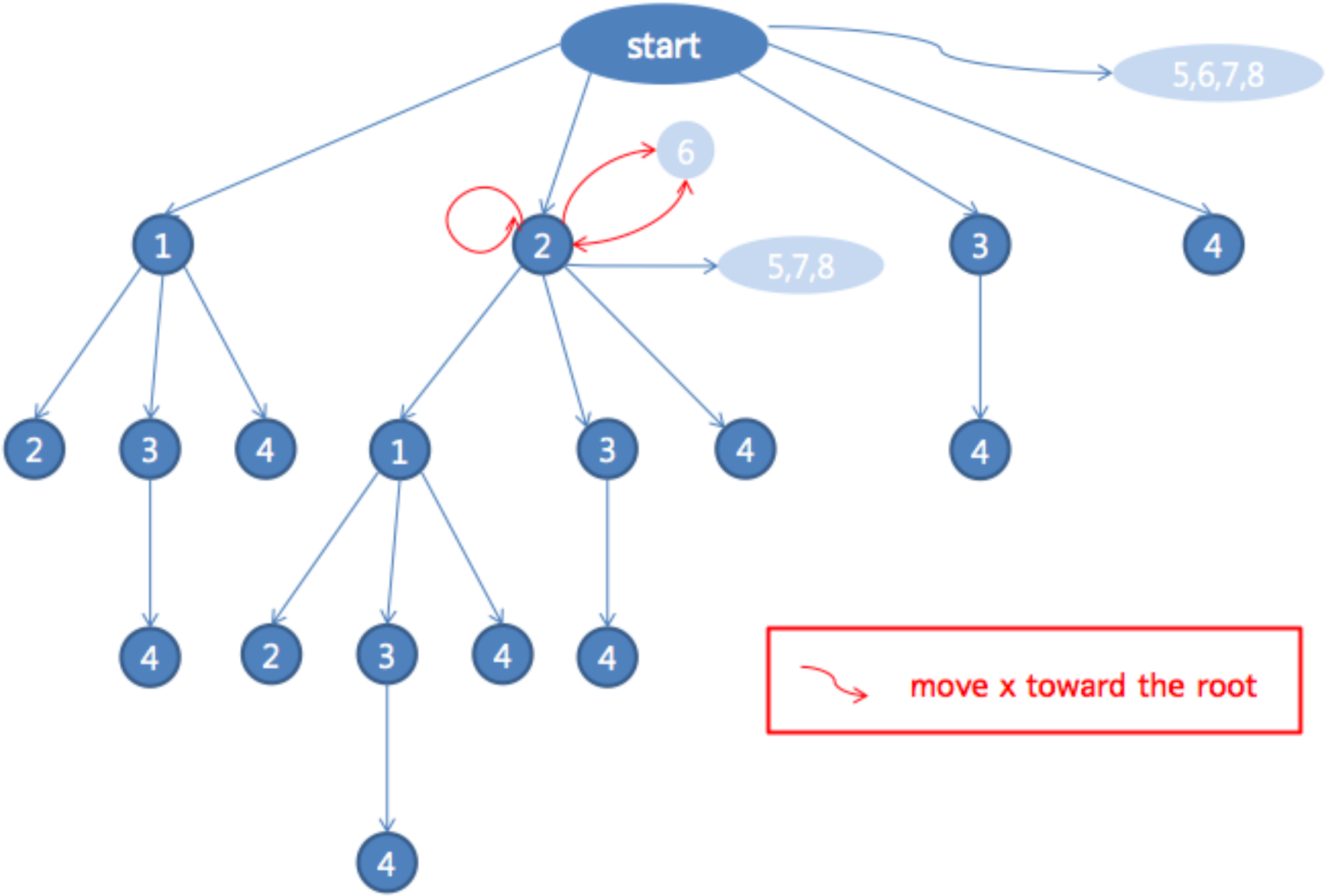
- w의 색을 현재 p[x]의 색으로 (unknown color)
- p[x]를 black으로, w의 오른자식을 black으로
- p[x]에 대해서 left-rotation적용
- x의 extra-black을 제거하고 종료

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2      do if x = left[p[x]]
 3          then w ← right[p[x]]
 4               if color[w] = RED
 5                  then color[w] ← BLACK                          ▷ Case 1
 6                       color[p[x]] ← RED                         ▷ Case 1
 7                       LEFT-ROTATE(T, p[x])                      ▷ Case 1
 8                       w ← right[p[x]]                           ▷ Case 1
 9               if color[left[w]] = BLACK and color[right[w]] = BLACK
10                  then color[w] ← RED                            ▷ Case 2
11                       x ← p[x]                                  ▷ Case 2
12                  else if color[right[w]] = BLACK
13                          then color[left[w]] ← BLACK            ▷ Case 3
14                               color[w] ← RED                    ▷ Case 3
15                               RIGHT-ROTATE(T, w)                ▷ Case 3
16                               w ← right[p[x]]                   ▷ Case 3
17                       color[w] ← color[p[x]]                    ▷ Case 4
18                       color[p[x]] ← BLACK                       ▷ Case 4
19                       color[right[w]] ← BLACK                   ▷ Case 4
20                       LEFT-ROTATE(T, p[x])                      ▷ Case 4
21                       x ← root[T]                               ▷ Case 4
22          else (same as then clause with "right" and "left" exchanged)
23  color[x] ← BLACK
```

경우 5,6,7,8

- 경우 1,2,3,4는 x가 왼쪽 자식인 경우
- 경우 5,6,7,8은 x가 p[x]의 오른쪽 자식인 경우
  - 경우 1,2,3,4와 각각 대칭적임


(a)
Case 1
new w
(b)
Case 2
new x
(c)
Case 3
new w
(d)
Case 4
new x = root[T]


start
5,6,7,8
6
1
2
5,7,8
3
4
2
3
4
1
3
4
4
4
2
3
4
4
4
move x toward the root

시간복잡도

- BST에서의 DELETE: $O(\log_2 n)$
- RB-DELETE-FIXUP: $O(\log_2 n)$