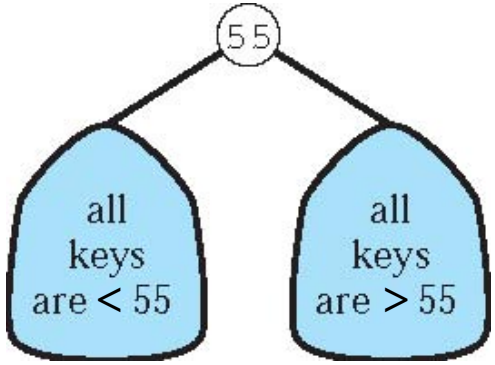
13. 탐색 트리
AVL트리, B-트리, 2-3-4트리

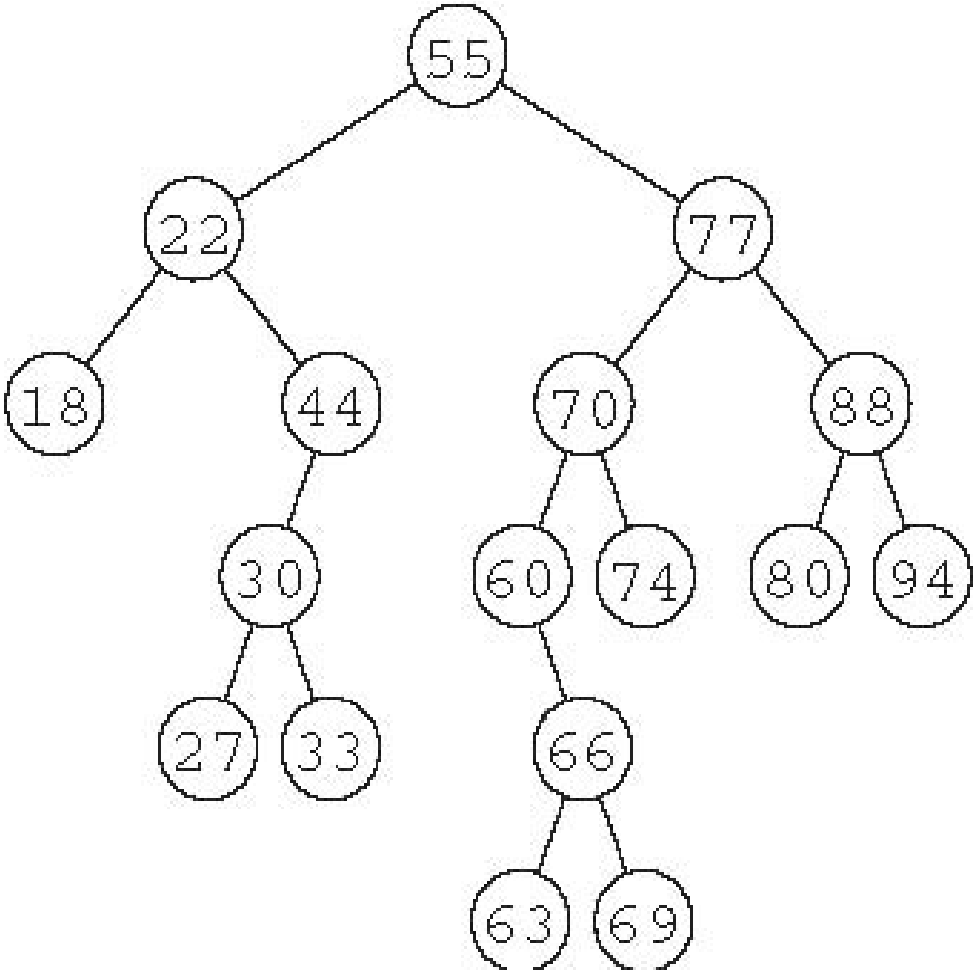
이진 탐색 트리

- 정의
  - 이진 탐색 트리(BST: binary search tree)는 각각의 노드가 BST 특성을 만족하는 키-주소 쌍을 가지고 있는 이진 트리

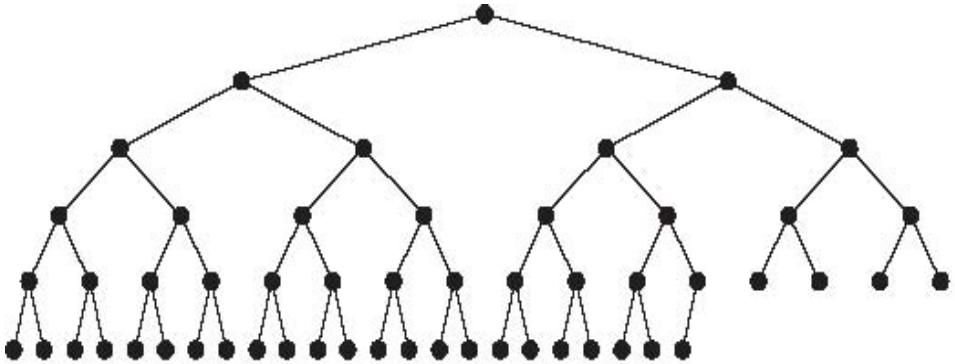
- BST 특성
  - 트리에 있는 각각의 키에 대해, 왼쪽 서브트리에 있는 모든 키는 이것보다 작고, 오른쪽 서브트리에 있는 모든 키는 이것보다 큼



이진 탐색 트리의 예



BST의 최선과 최악의 경우



BST 성능

- 이진 탐색 트리의 삽입과 탐색
  - $B(n) = \Theta(1)$
  - $A(n) = \Theta(\lg n)$
  - $W(n) = \Theta(n)$
- BST 탐색과 삽입 알고리즘에 대한 평균 시간 복잡도

$$A(n) = \Theta(1.39 \lg n) = \Theta(\lg n)$$

13.5 AVL 트리

- **AVL 트리**
  - 어떤 노드에서도 두 서브트리가 거의 같은 높이를 갖도록 강제하여 균형을 유지하는 이진 탐색 트리
  - 불균형이 발생할 때마다 서브트리를 회전하여 균형을 맞추어 줌
  - AVL트리를 발명한 Adelson-Velskii와 Landis의 이름을 땀
  - Average and worst case: $O(\log_2 n)$

- **AVL 회전의 패턴**
  - 왼쪽 단순 회전
  - 오른쪽 단순 회전
  - 복합 오른쪽-왼쪽 회전
  - 복합 왼쪽-오른쪽 회전

AVL 트리(2)

Height balanced tree

- 공백트리(empty tree)는 높이 균형을 이룬다.
- T가 왼쪽 서브트리 T_L과 오른쪽 서브트리 T_R을 가진 공백이 아닌 이진트리라고 할 때,

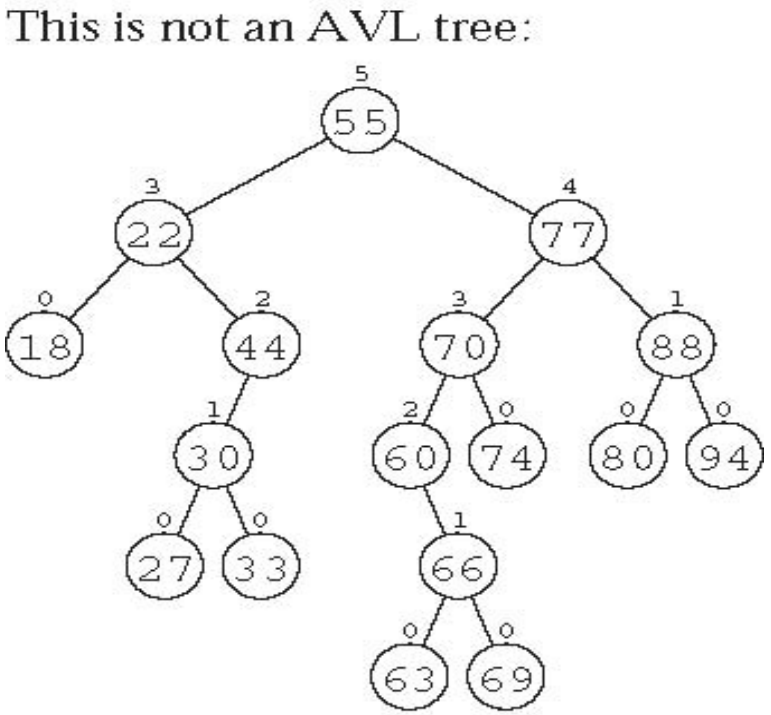
  - T는 높이균형을 이룬다 iff

1) T_L 과 T_R 이 높이 균형을 이룬다

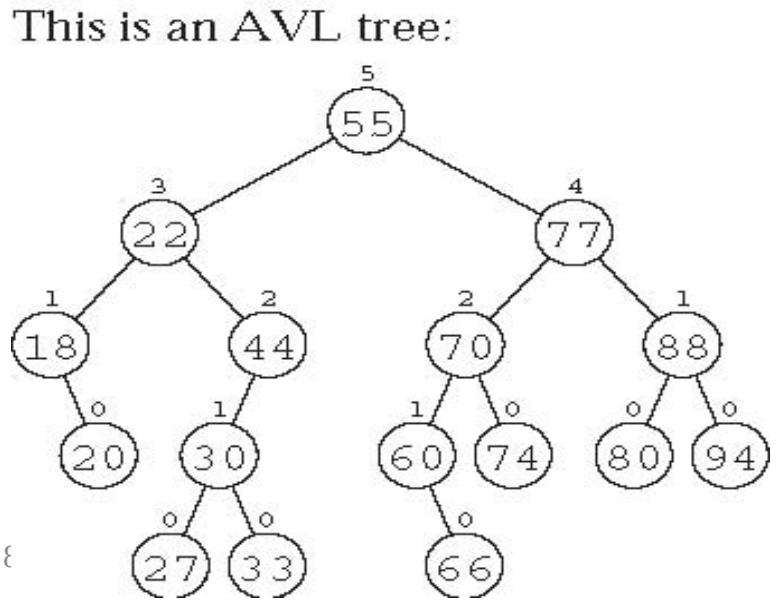
2) $|h_L - h_R| \leq 1$ (h_L과 h_R은 각각 T_L과 T_R의 높이)

AVL 특성

AVL 트리가 아님 :



AVL 트리임 :



AVL TREES

Def) 노드 T 의 *balance factor*, BF(T)

$h_L - h_R$

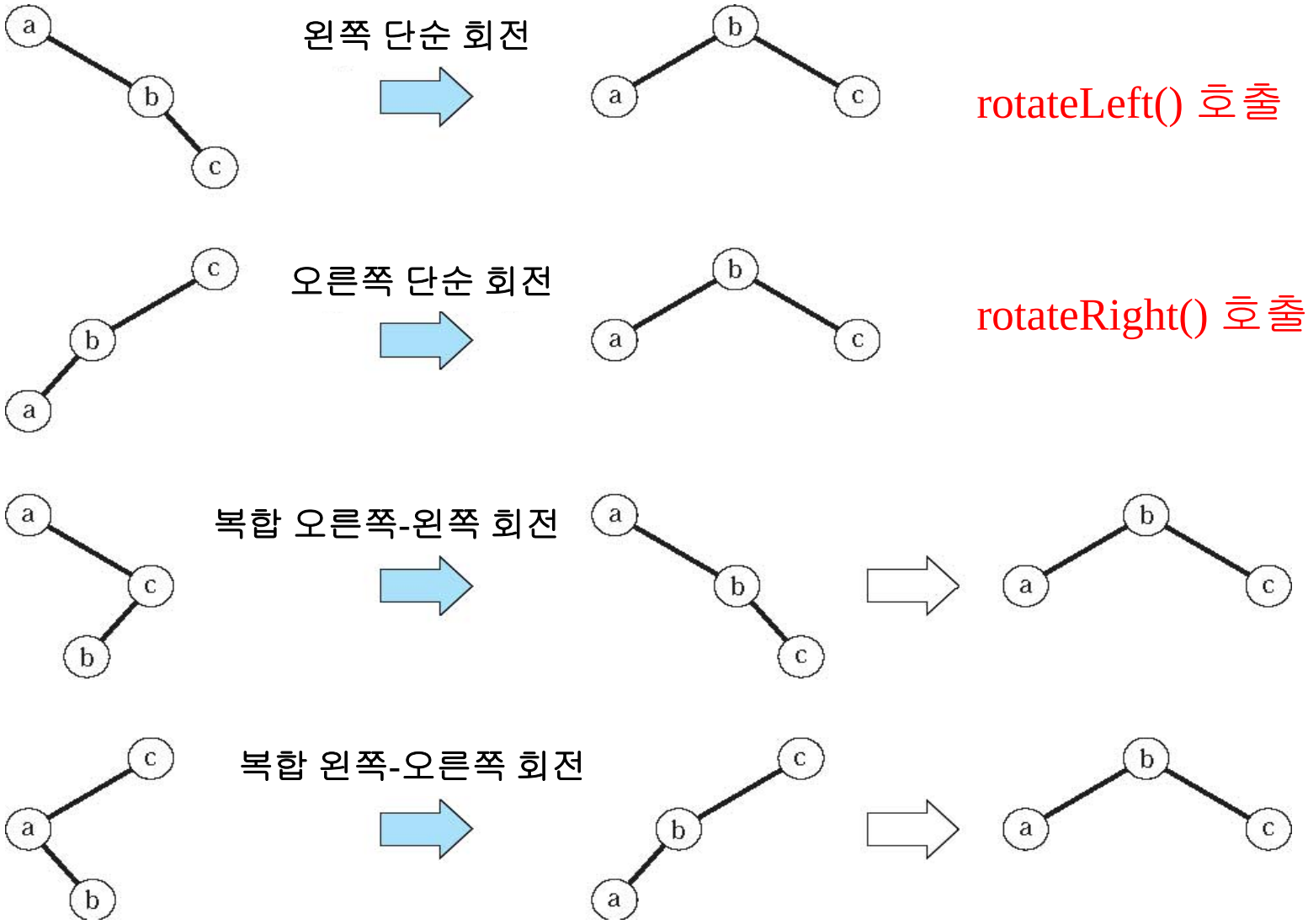
(h_L과 h_R은 각각 왼쪽 서브트리 T_L과 오른쪽 서브트리 T_R의 높이)

– AVL 트리의 임의의 노드 T에 대해

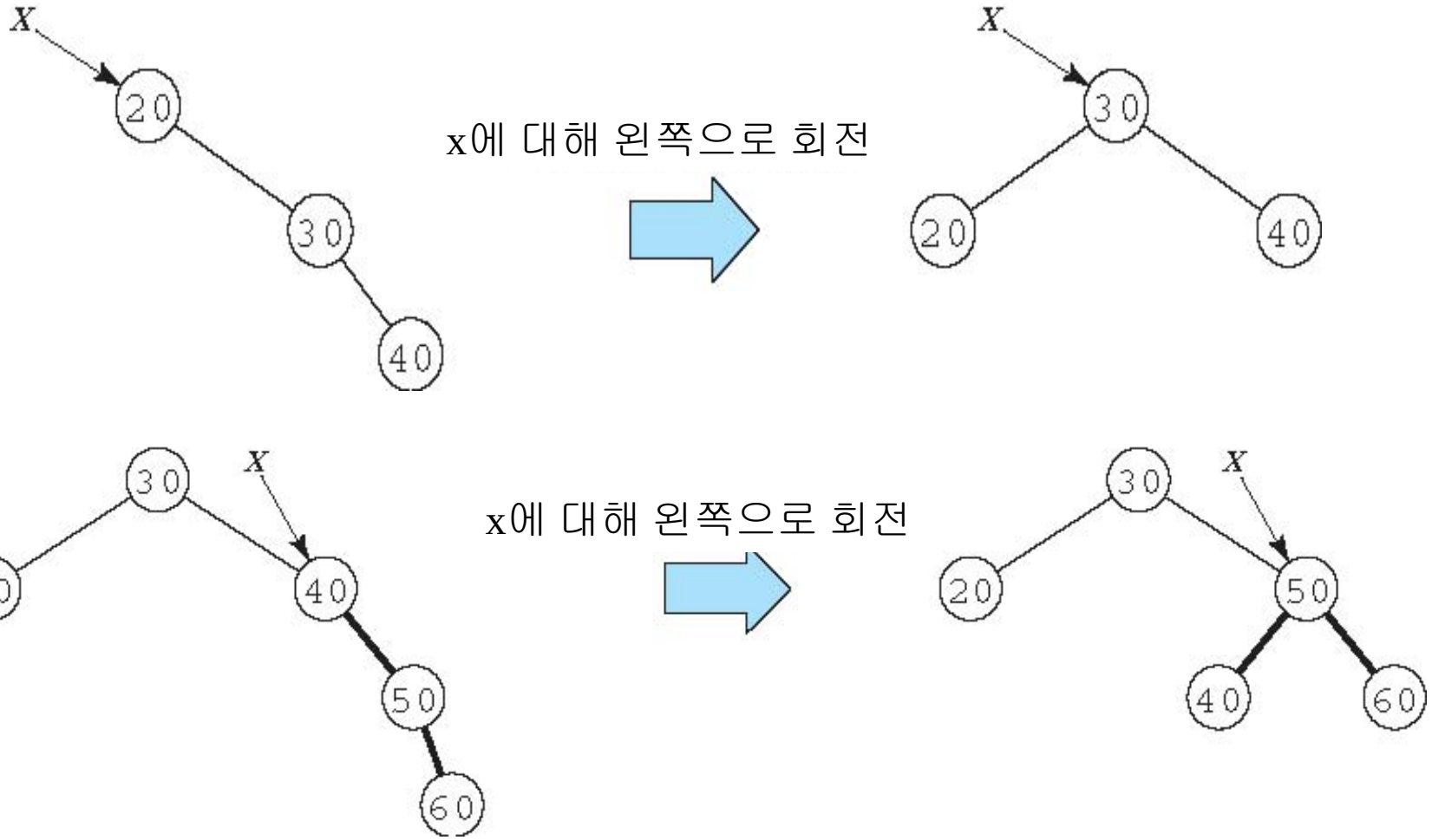
BF(T) = -1, 0, 또는 1

트리가 균형을 잃게 될 때, 트리의 균형을 맞추기 위해 회전(rotation)을 수행한다

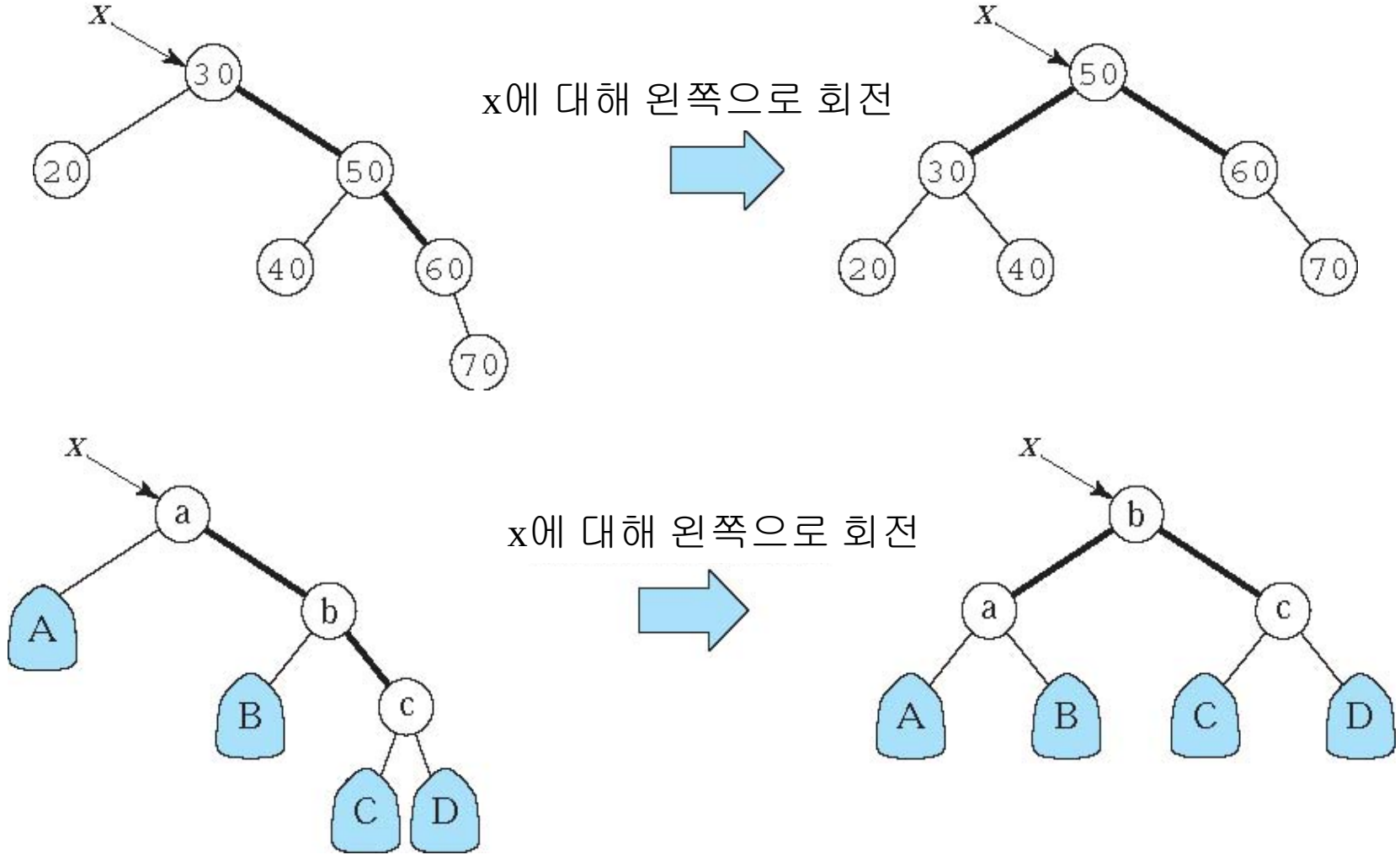
AVL 회전의 패턴


왼쪽 단순 회전
rotateLeft() 호출
오른쪽 단순 회전
rotateRight() 호출
복합 오른쪽-왼쪽 회전
복합 왼쪽-오른쪽 회전

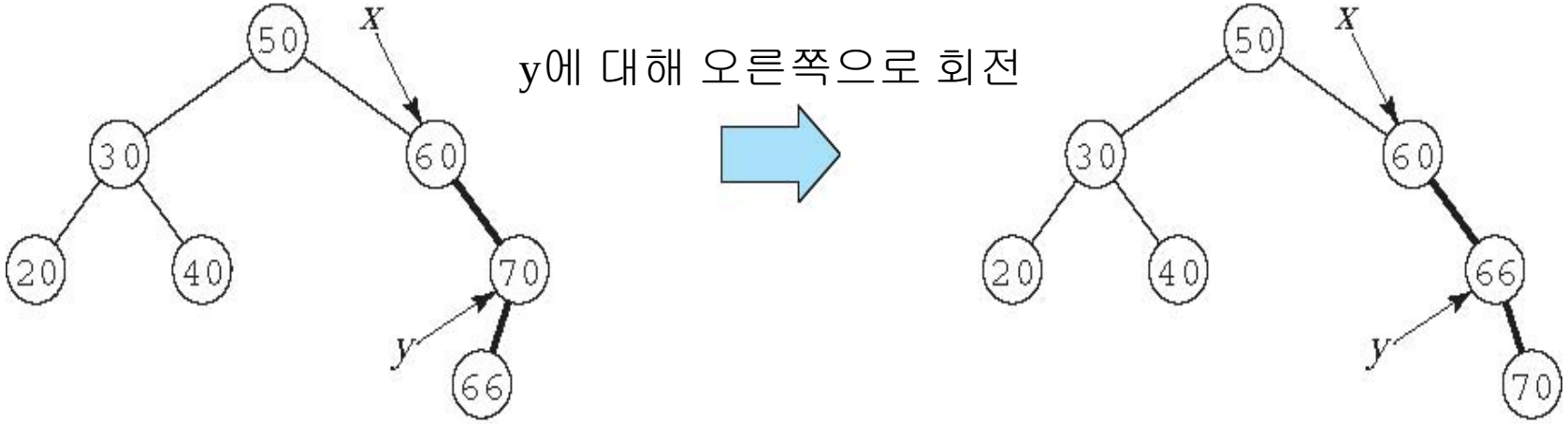
왼쪽 회전 알고리즘 (1)

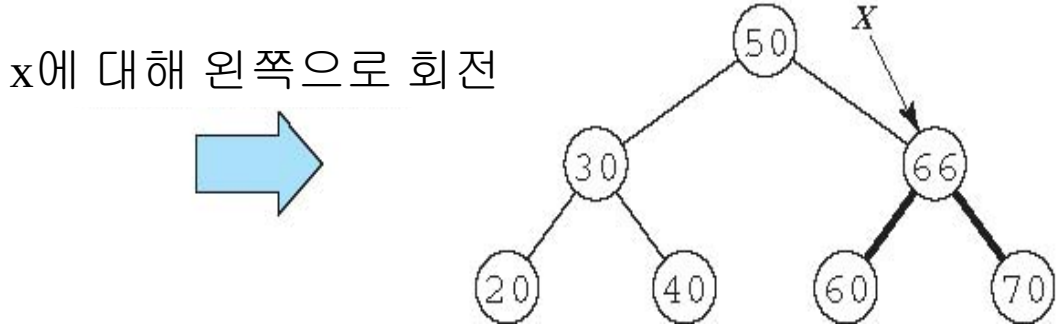

X
20
30
40
x에 대해 왼쪽으로 회전
X
30
20
40
30
X
20
40
50
60
x에 대해 왼쪽으로 회전
30
X
20
50
40
60

왼쪽 회전 알고리즘 (2)

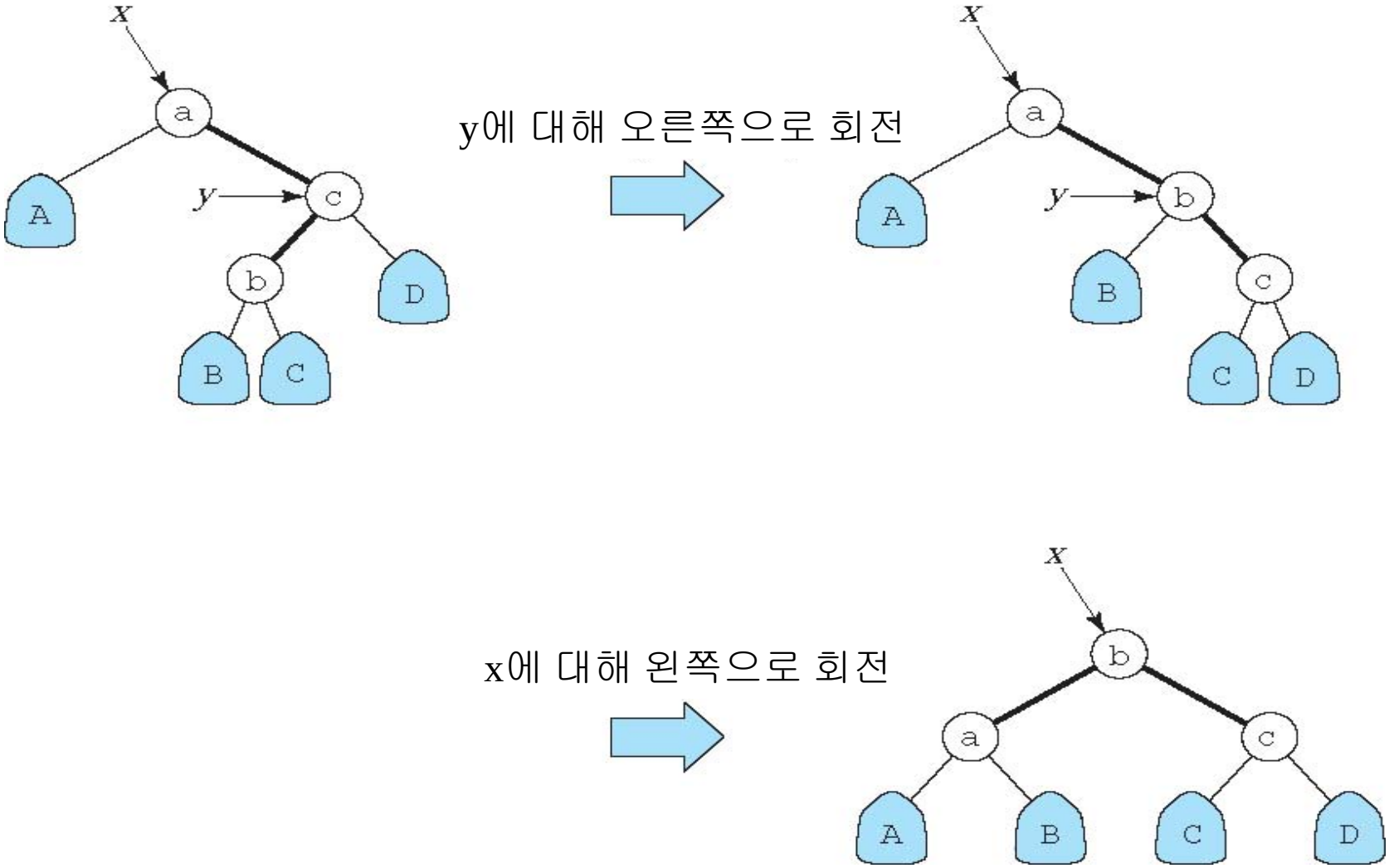

X
30
20
50
40
60
70
x에 대해 왼쪽으로 회전
X
50
30
60
20
40
70
X
a
A
b
B
c
C
D
x에 대해 왼쪽으로 회전
X
b
a
c
A
B
C
D

이중 회전 알고리즘 (1)


50
X
30
60
20
40
70
y
66
y에 대해 오른쪽으로 회전
50
X
30
60
20
40
66
y
70


x에 대해 왼쪽으로 회전
50
X
30
66
20
40
60
70

이중 회전 알고리즘 (2)



13.6 AVL 트리의 구현 (1)

- An AVLTree class

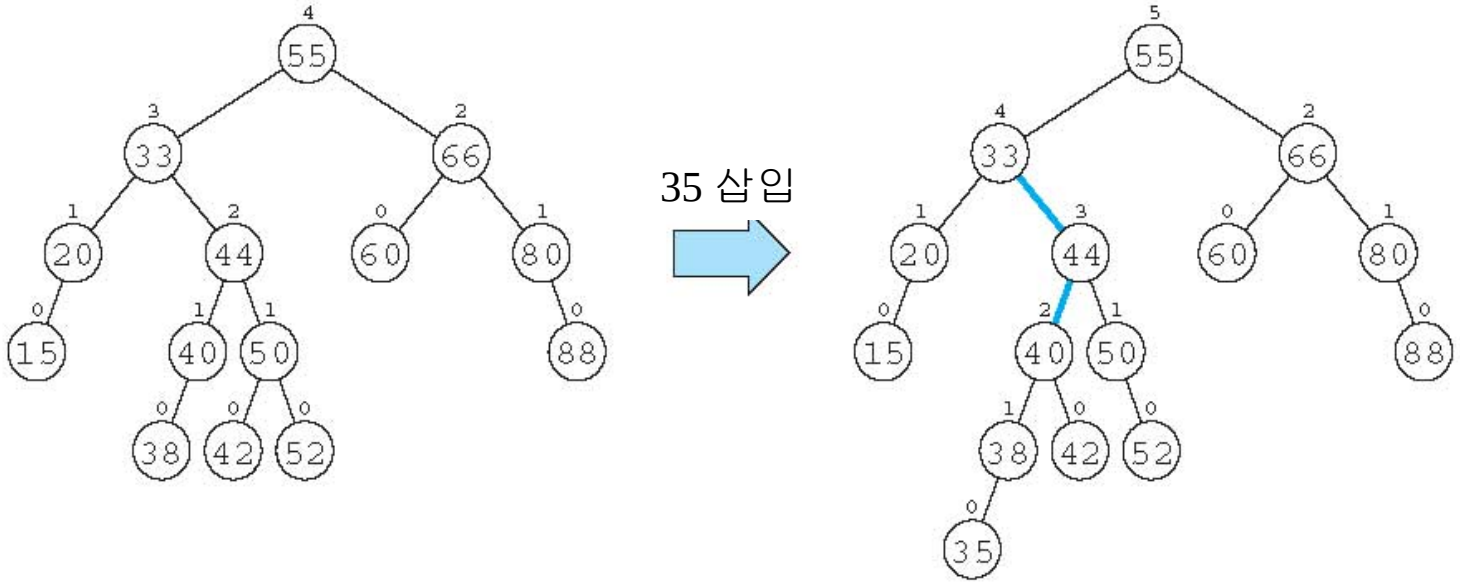
```
1    public class AVLTree {
2        private int key, height;
3        private AVLTree left, right;
5        public static final AVLTree NIL = new AVLTree();

7        public AVLTree(int key){
8            this.key = key;
9            left = right = NIL;
10       }
12       public boolean add(int key) {        //주어진 키를 추가하는데
13           int oldSize = size();            //성공하면 true 리턴
14           grow(key);
15           return size()>oldSize;
16       }
```

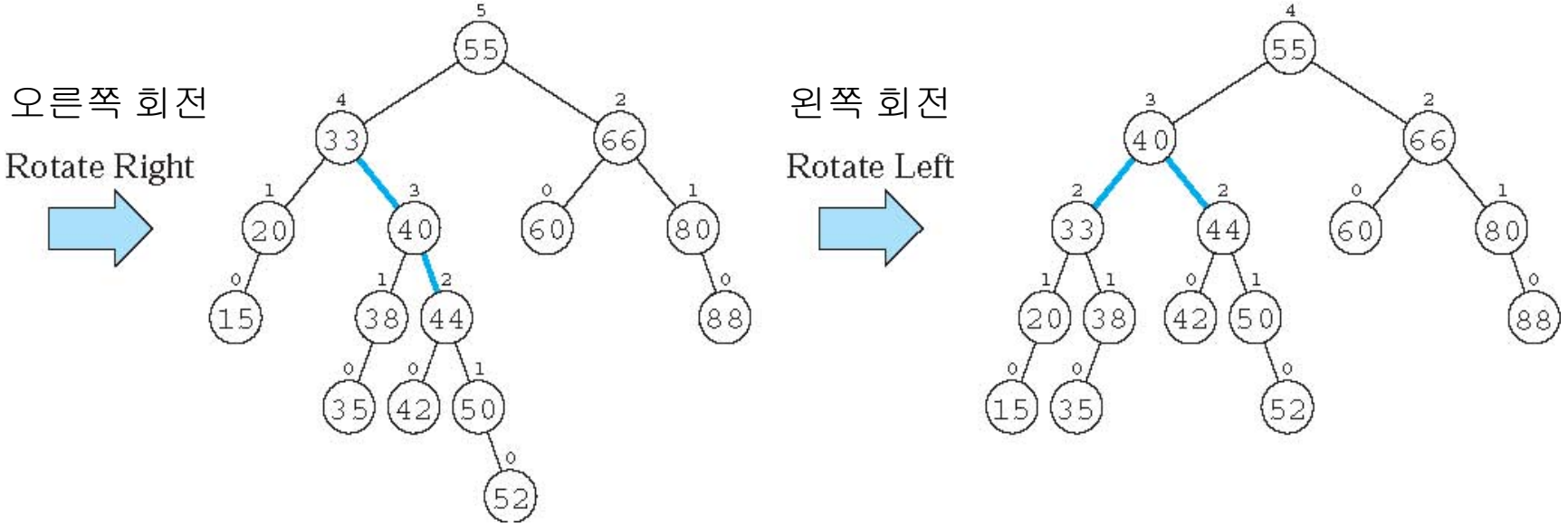
```
18      public AVLTree grow(int key) {
19          if (this == NIL) return new AVLTree(key);
20          if (key == this.key) return this; // prevent key duplication
21          if (key < this.key) left = left.grow(key);
22          else right = right.grow(key);
23          rebalance( );
24          height = 1 + Math.max(left.height,right.height);
25          return this;
26      }
28      public int size() {
29          if (this == NIL) return 0;
30          return 1 + left.size() + right.size();
31    }
33    public String toString() {
34          if (this == NIL) return "";
35          return left + " " + key + " " + right;
36    }
```

```
38      private AVLTree() { // constructs the empty tree
39          left = right = this;
40          height = -1;
41      }
43      private AVLTree(int key, AVLTree left, AVLTree right) {
44          this.key = key;
45          this.left = left;
46          this.right = right;
47          height = 1 + Math.max(left.height, right.height);
48      }
50      private void rebalance() {
51           if (right.height > left.height+1) {
52                if (right.left.height > right.right.height)
                       right.rotateRight();
53                rotateLeft();
54           }
```

```
55          else if (left.height > right.height+1) {
56              if (left.right.height > left.left.height) left.rotateLeft();
57              rotateRight();
58          }
59      }
60
61      private void rotateLeft() {
62          left = new AVLTree(key, left, right.left);
63          key = right.key;
64          right = right.right;
65      }
66
67      private void rotateRight() {
68          right = new AVLTree(key, left.right, right);
69          key = left.key;
70          left = left.left;
71      }
72  }
```

회전이 있는 AVL 삽입



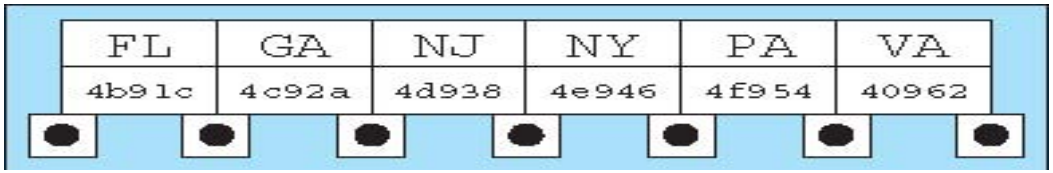


13.7 다원 탐색 트리

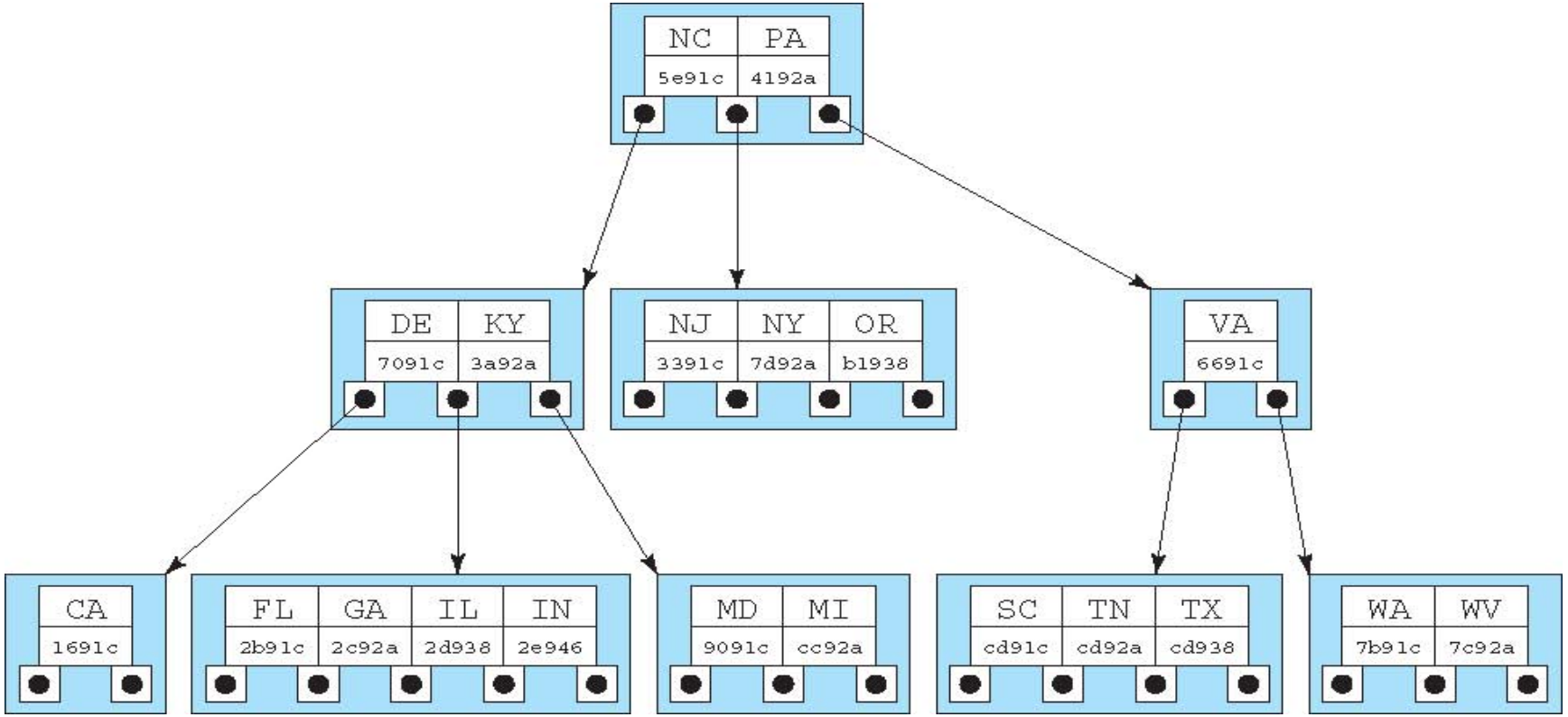
- 차수가 d인 다원 탐색 트리(d-way search tree)
  - 그것의 노드의 차수가 <= d
  - 차수가 d인 노드는 d-1개의 키 $(k_0, k_1, \ldots, k_{d-2})$, d-1개의 주소 $(a_0, a_1, \ldots, a_{d-2})$, d개의 서브트리 $(T_0, T_1, \ldots, T_{d-1})$를 가진다 만일 x_0, x_1, x_2, ..., x_{d-1}을 각 서브트리에 있는 키라고 하면 $x_0 < k_0 < x_1 < k_1 < x_2 < k_2 < \ldots x_{d-2} < k_{d-2} < x_{d-1}$ 임

다원 탐색 트리의 예

- 차수가 7인 다원 탐색 노드



- 차수가 5인 다원 탐색 트리



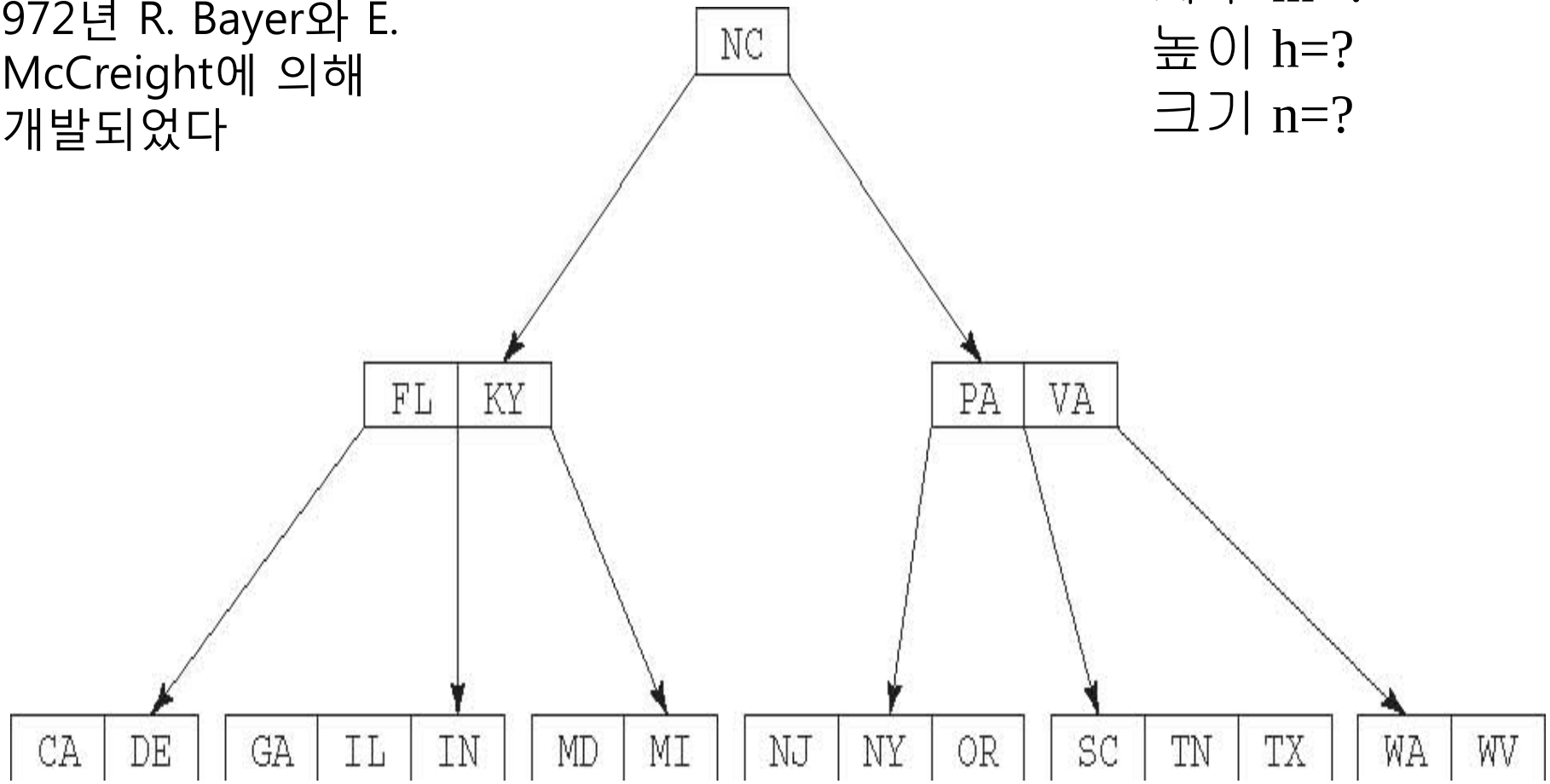
13.9 B-트리

- 차수가 *m*인 B-트리
  - 차수가 *m*인 다원 탐색 트리(MST-multiway search tree):
    그것의 노드의 차수가 <= m
  - 모든 리프 노드는 동일한 레벨에 있음
  - 루트가 아닌 모든 내부 노드는 최소한 $\lceil m/2 \rceil$ 의 차수를 가짐

B-트리의 예

972년 R. Bayer와 E. McCreight에 의해 개발되었다

차수 m=?
높이 h=?
크기 n=?



B-트리(2)

- 높이가 h이고 차수가 m인 B-트리의 크기에 대한 범위

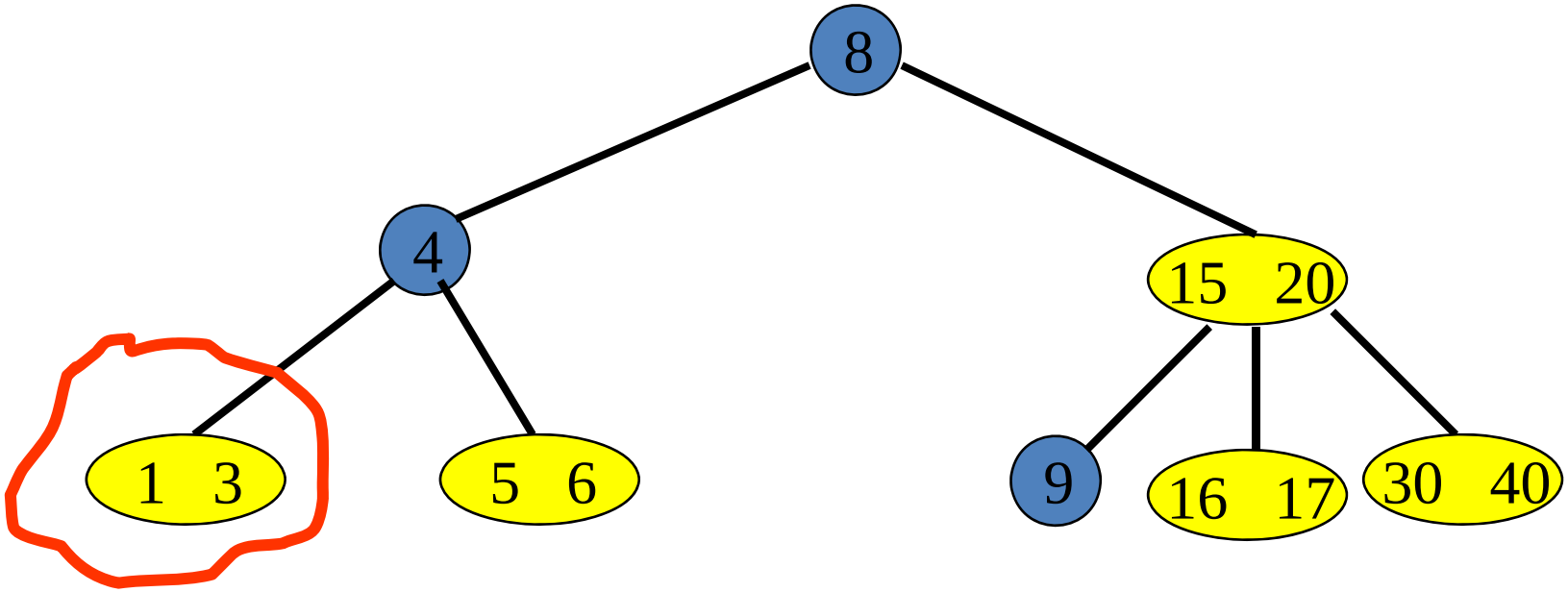
$$2\lceil m/2 \rceil^h - 1 \le n \le m^{h+1} - 1$$

- 크기가 n이고 차수가 m인 B-트리의 높이에 대한 범위

$$h \le \log_{m/2} n = \Theta(\lg n)$$

- B-트리는 데이터베이스 테이블을 위한 외부 인덱스를 구현하는데 사용되는 표준 자료 구조이다. 각각의 키는 레코드에 대한 디스크주소를 가지고 있다. B-트리의 차수는 각 노드가 하나의 디스크 블럭에 저장될 수 있는 값으로 선택된다. 따라서 어떤 레코드를 접근하기 위한 디스크 판독 횟수는 h+2를 넘지 않는다.

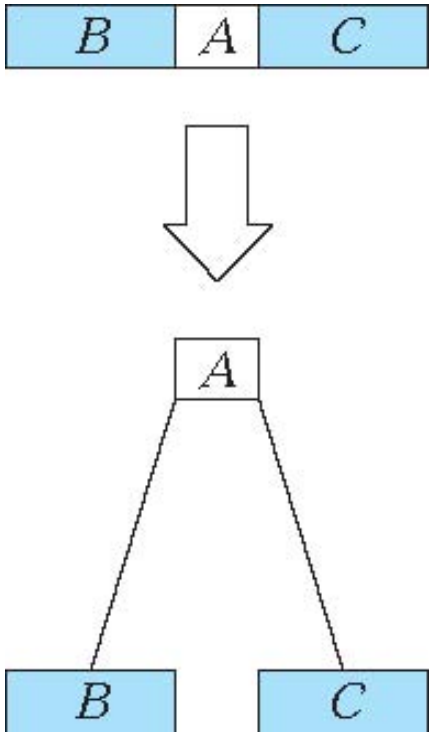
차수가 3인 B-트리로의 삽입



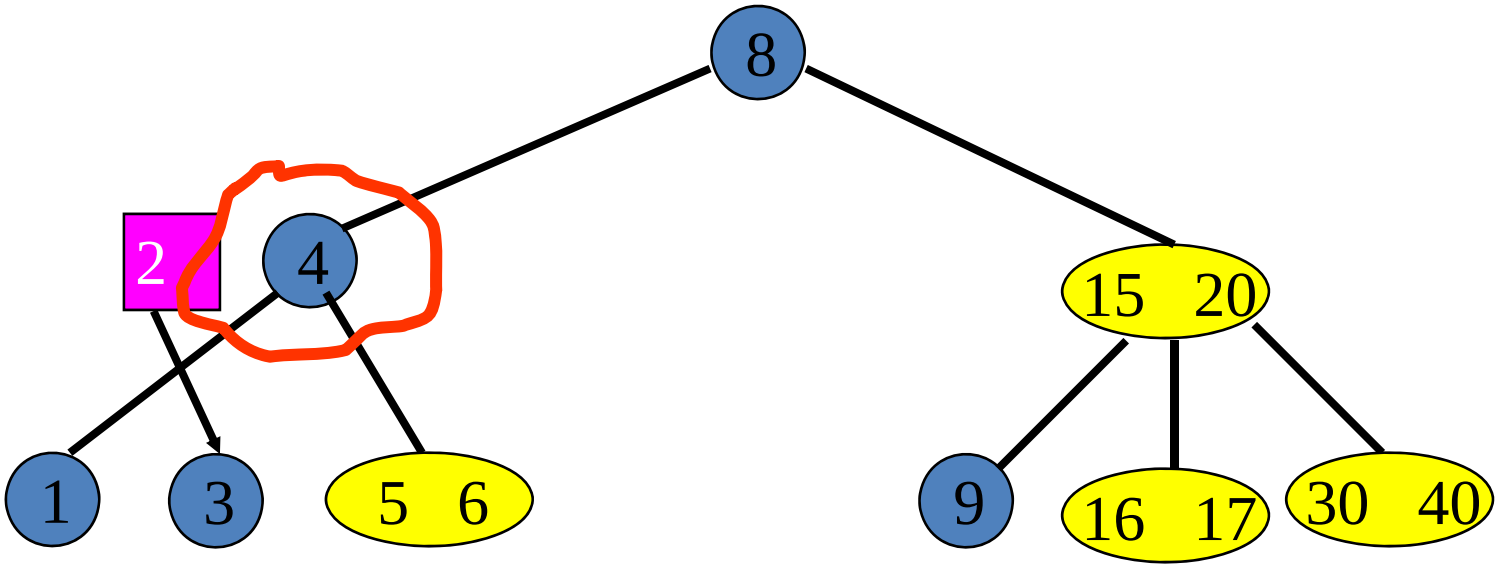
- Insert a pair with key = 2.
- New pair goes into a 3-node.

B-트리 분할 연산

- B-트리 분할 알고리즘
  - 포화 노드는 하나의 단독 노드 A와 이것의 두 자식이 되는 두 개의 반-포화(half-full) 노드 B와 C로 교체됨
  - A에 있는 하나의 원소는 원래 노드에 있는 키의 중앙값이다.

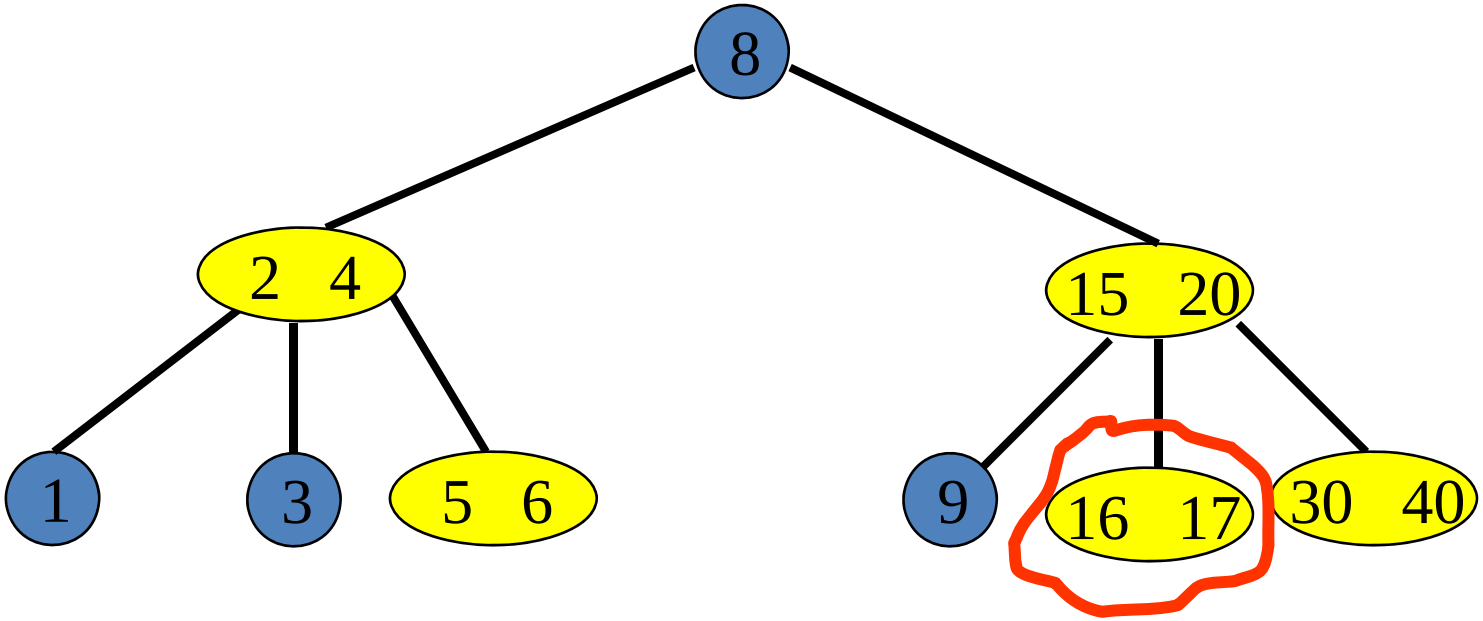


Insert



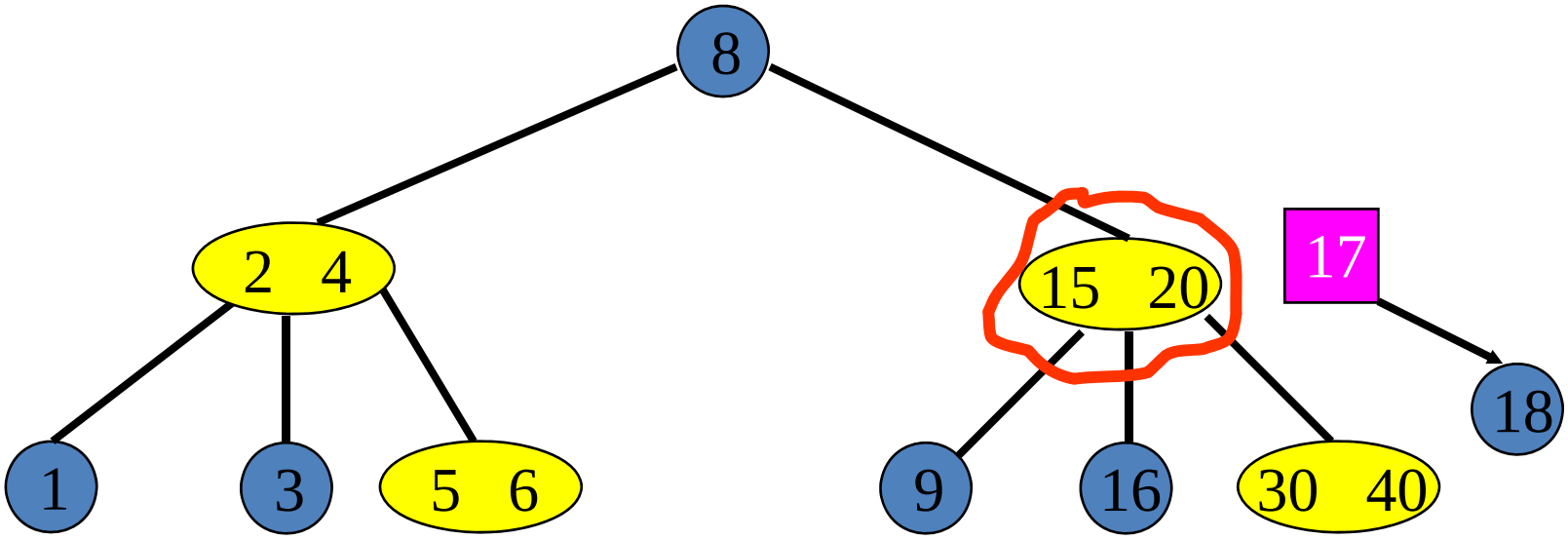
- Insert a pair with key = 2 plus a pointer into parent.

Insert



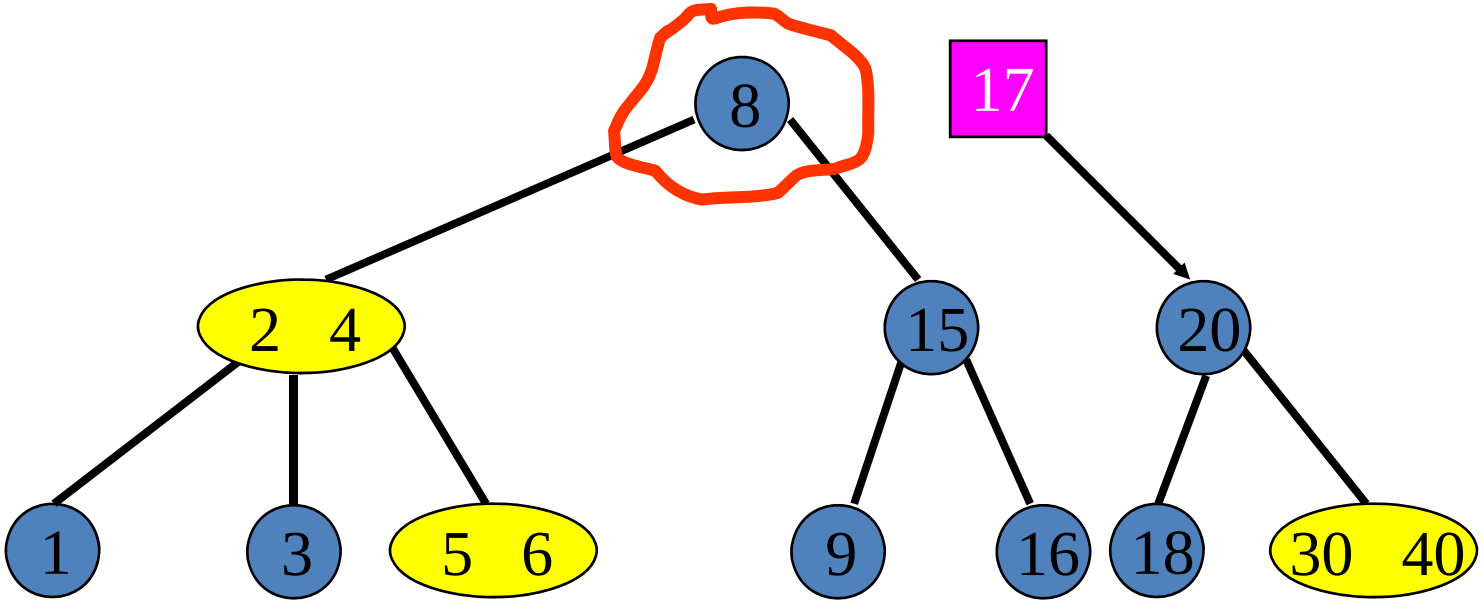
- Now, insert a pair with key = 18.

Insert



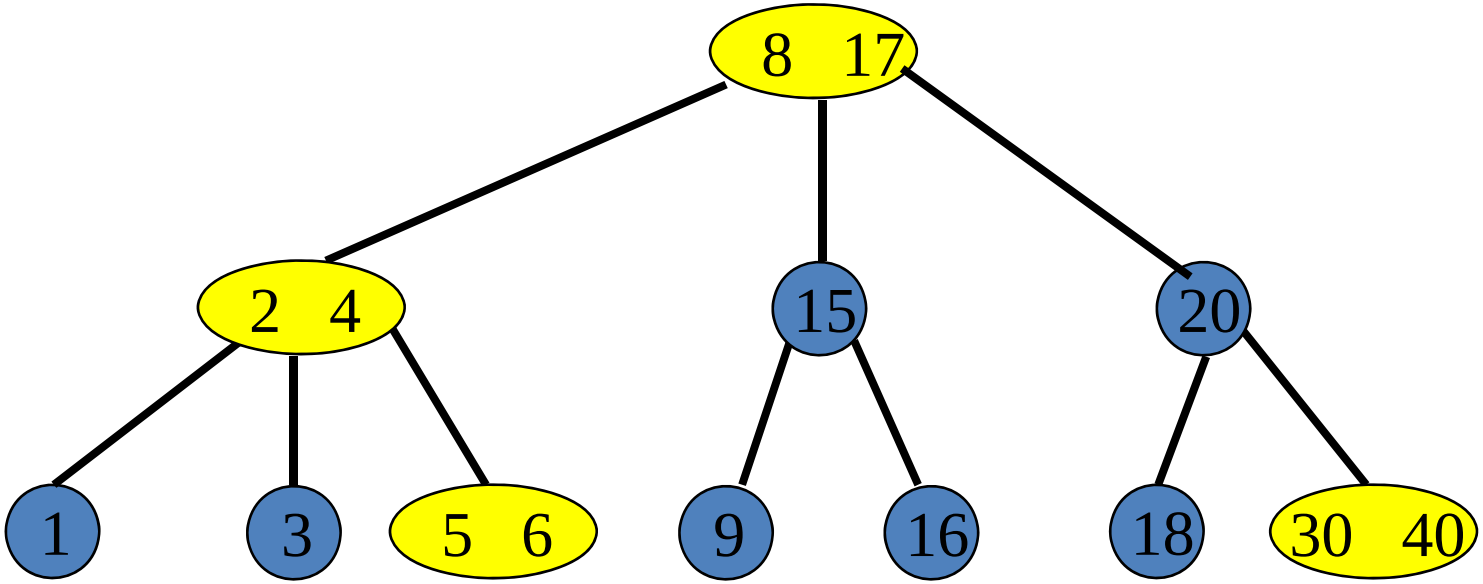
- Insert a pair with key = 17 plus a pointer into parent.

Insert



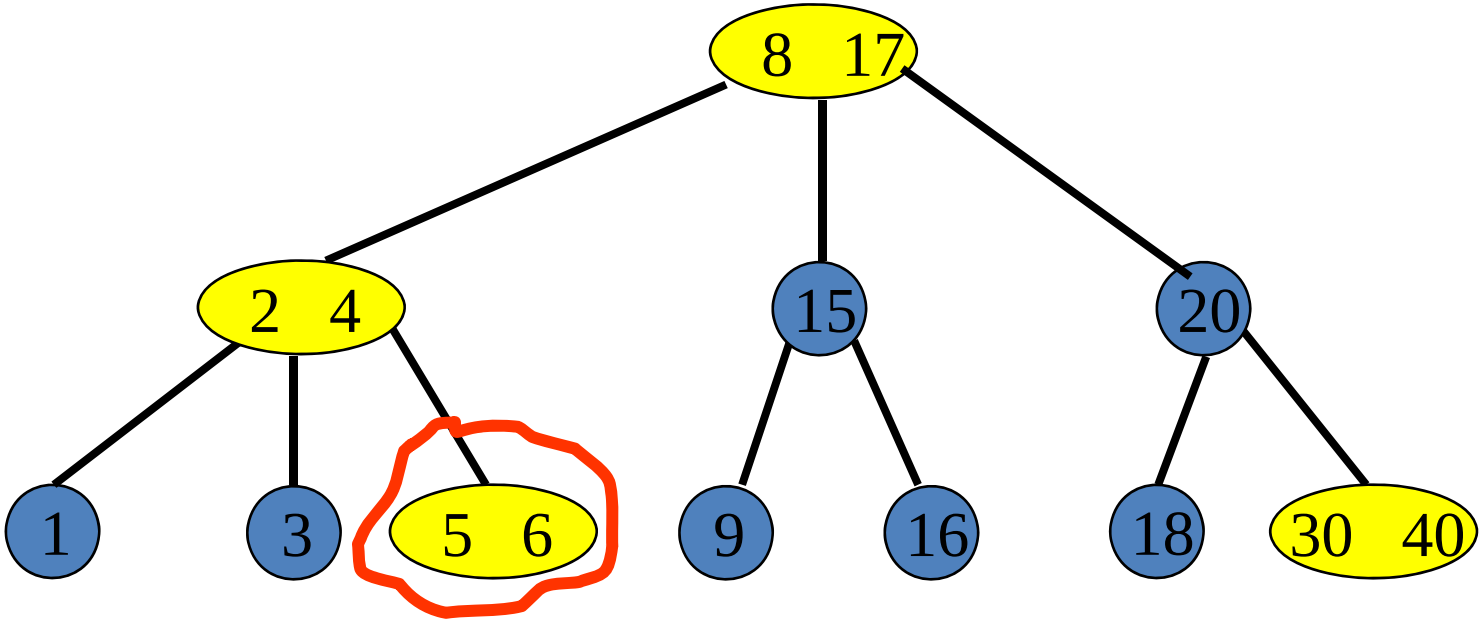
- Insert a pair with key = 17 plus a pointer into parent.

Insert



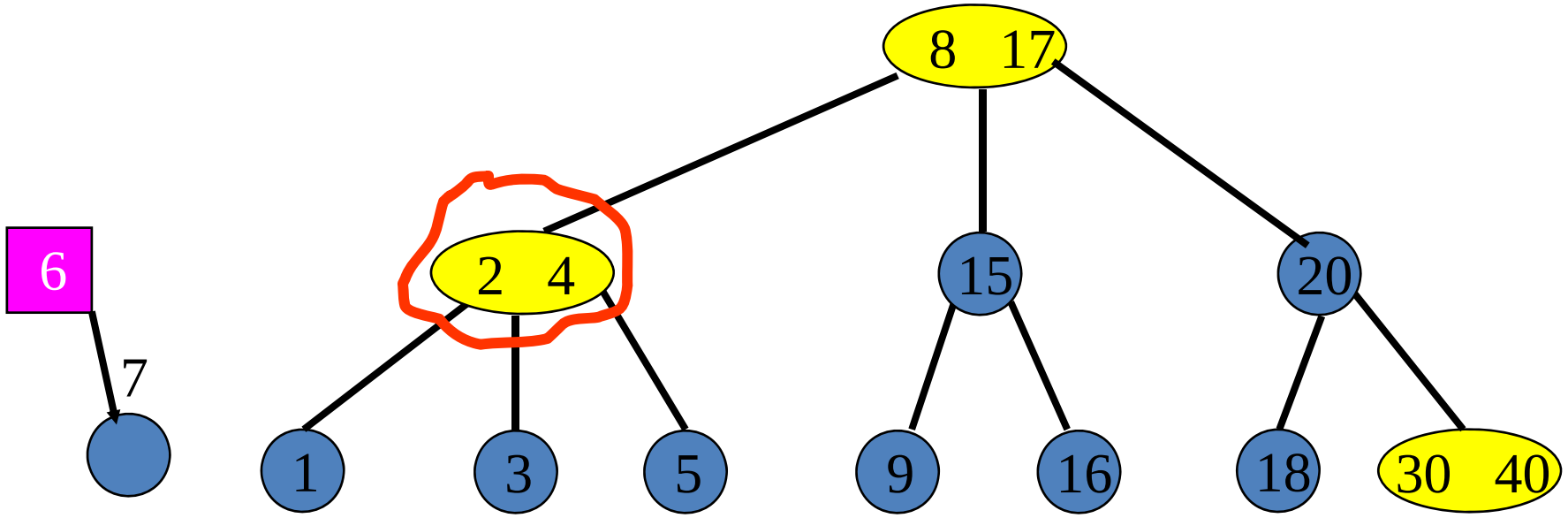
- Now, insert a pair with key = 7.

Insert



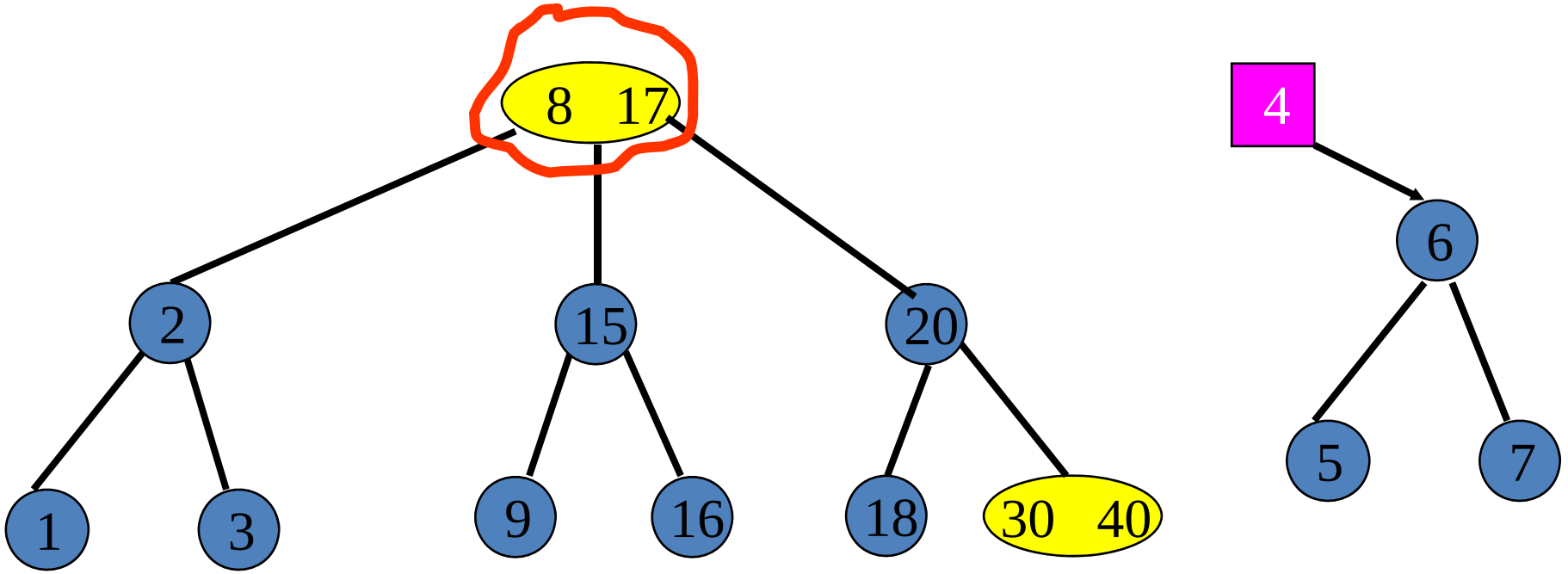
- Now, insert a pair with key = 7.

Insert



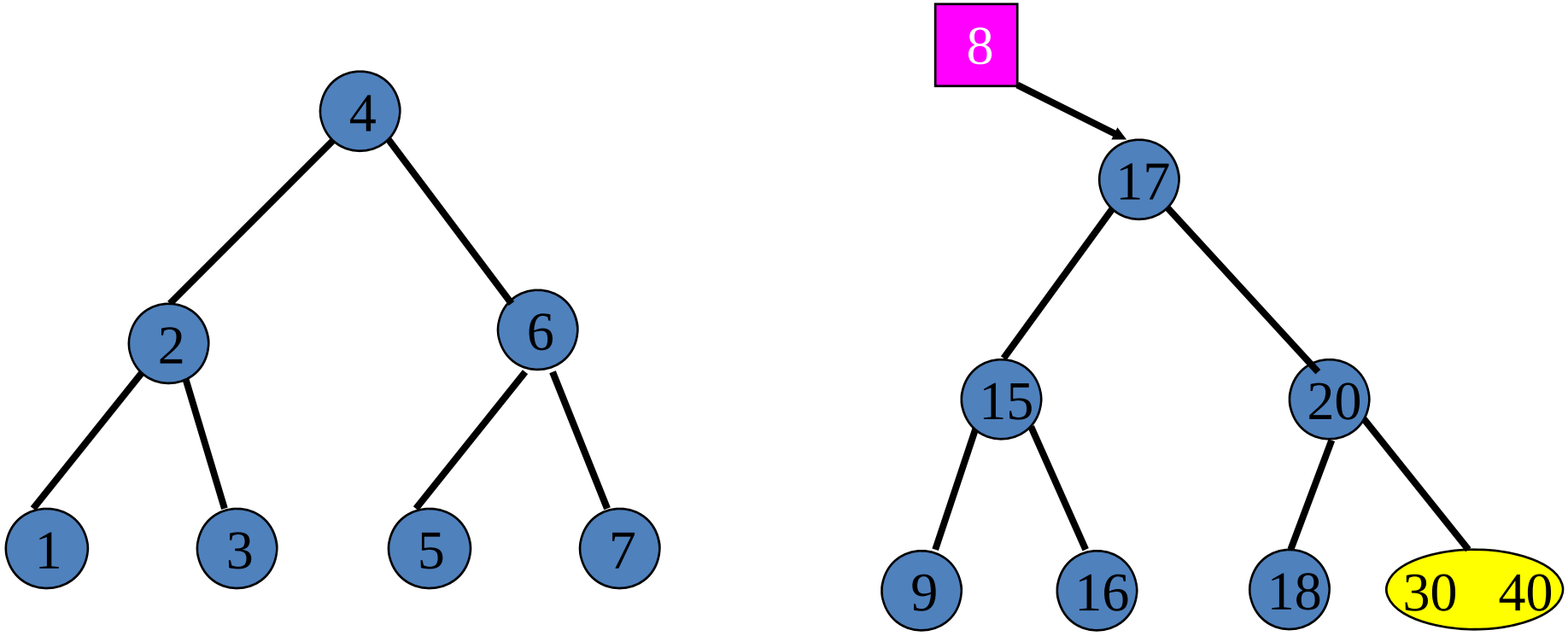
- Insert a pair with key = 6 plus a pointer into parent.

Insert



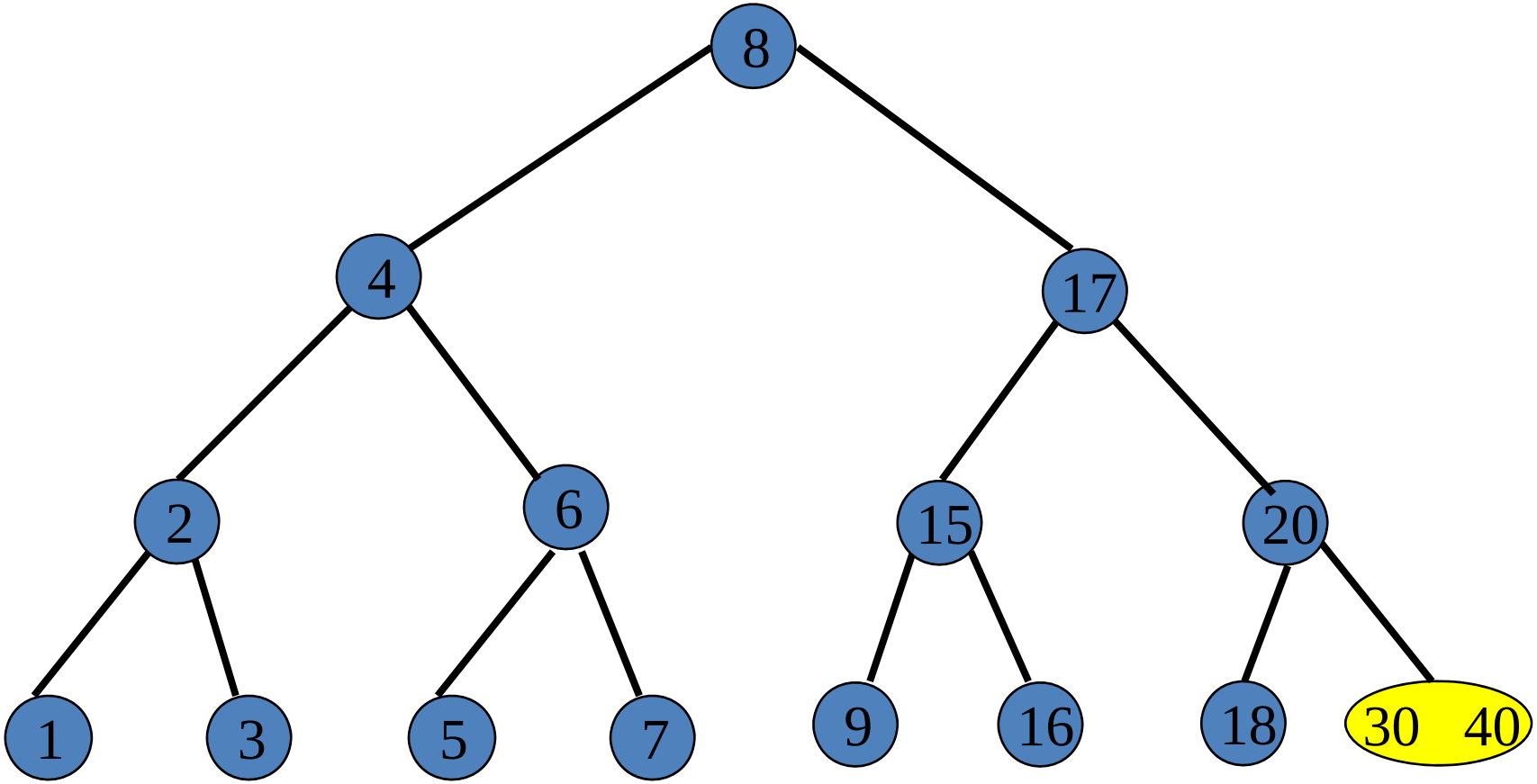
- Insert a pair with key = 4 plus a pointer into parent.

Insert



- Insert a pair with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.

Insert



- Height increases by 1.

B-트리 삽입 알고리즘

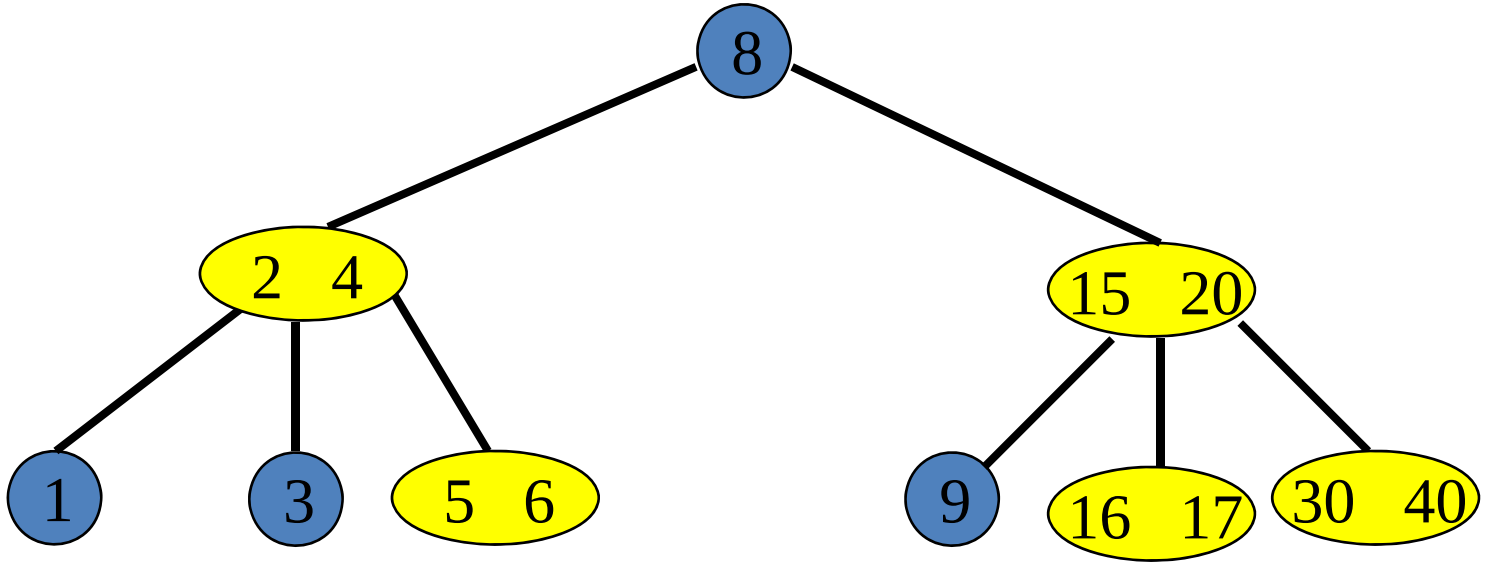
입력 : B-트리 T와 키-주소 쌍 (x,y).

– 출력 : 만일 T가 변경되었으면 true, 아니면 false.

– 후조건 : 키-주소 쌍 (x,y)는 트리 T에 존재.

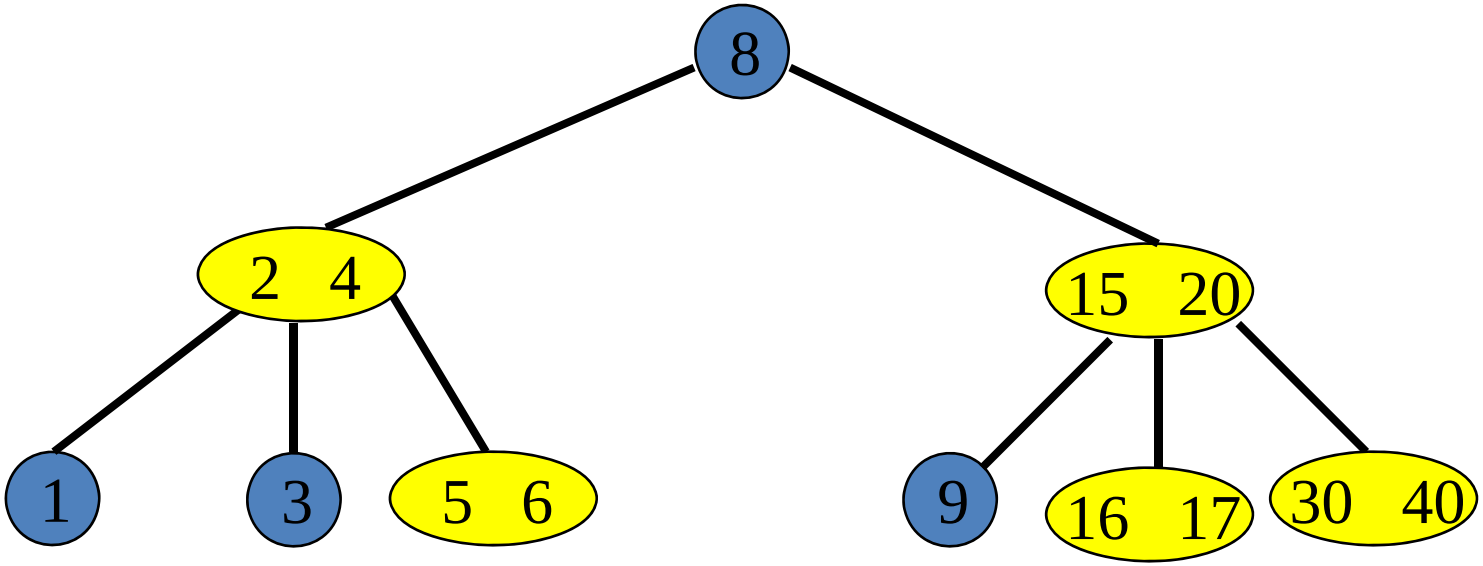
1. 만일 *T*가 공백이면, 키-주소 쌍 (*x,y*)를 포함하는 단독 트리로 교체하고 true를 리턴.
2. *x*를 가지고 있는 리프 노드 *p*를 찾기 위해 다원 탐색 알고리즘을 적용.
3. 만일 *x*가 *p*에 있고 그것의 주소가 *y*이면, false를 리턴.
4. 만일 *x*가 *p*에 있고 그것의 주소가 *y*가 아니면, 그 주소를 *y*로 교체하고 true를 리턴.
5. 만일 *x*가 *p*에 없으면, 키-주소 쌍 (*x,y*)를 삽입.
6. 만일 *p*가 오버플로되면, 그것을 분할.
7. true를 리턴.

차수가 3인 B-트리에서 Delete



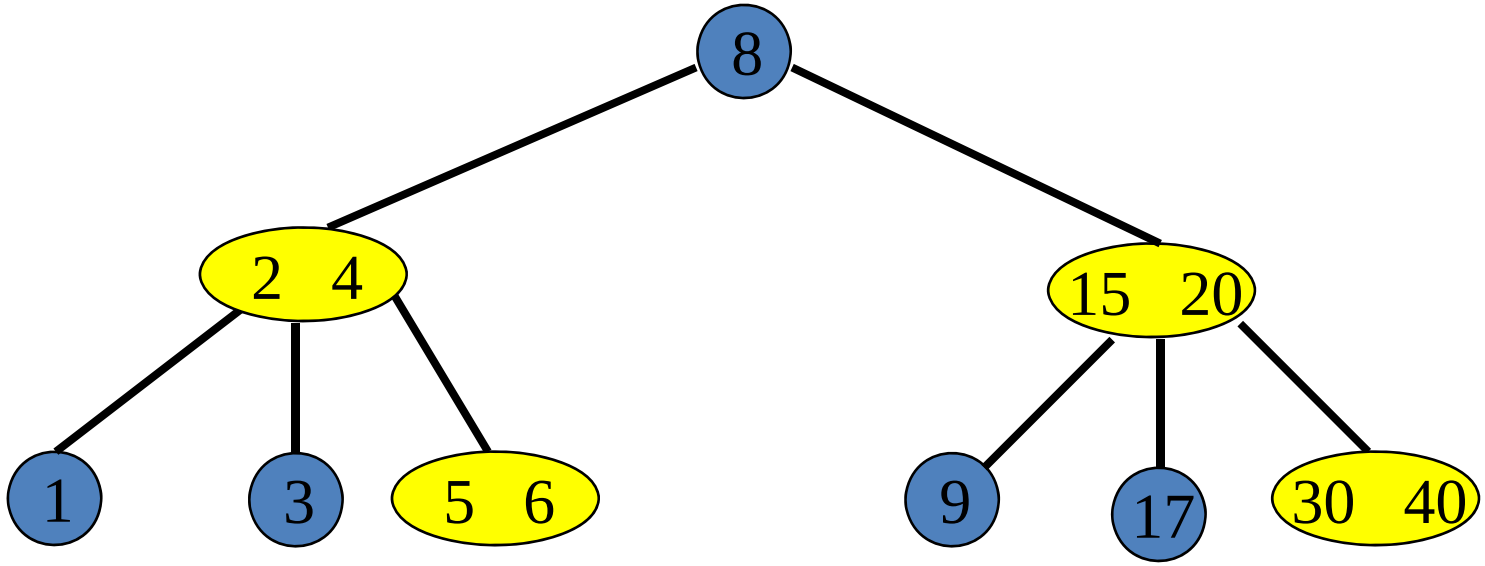
- Delete the pair with key = 8.
- Transform deletion from interior into deletion from a leaf.
- Replace by the smallest in right subtree.

Delete From A Leaf

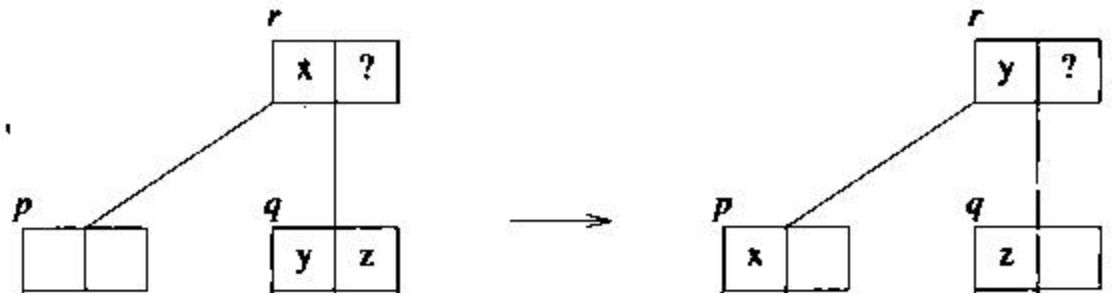


- Delete the pair with key = 16.
- 3-node (node with degree 3) becomes 2-node.

Delete From A Leaf

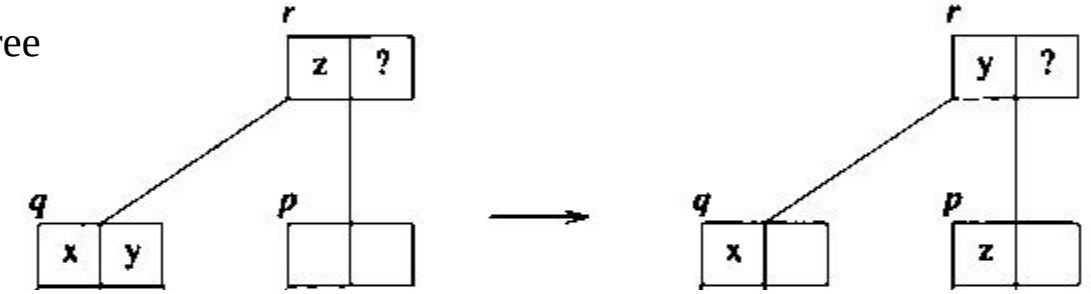


- Delete the pair with key = 17.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If so, borrow a pair via parent node.

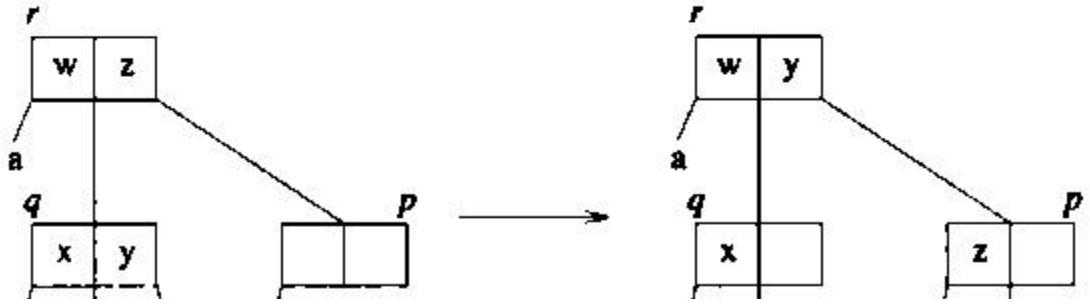


(a) *p* is the left child of *r*

Figure 11.7
Rotation in a 2-3 tree

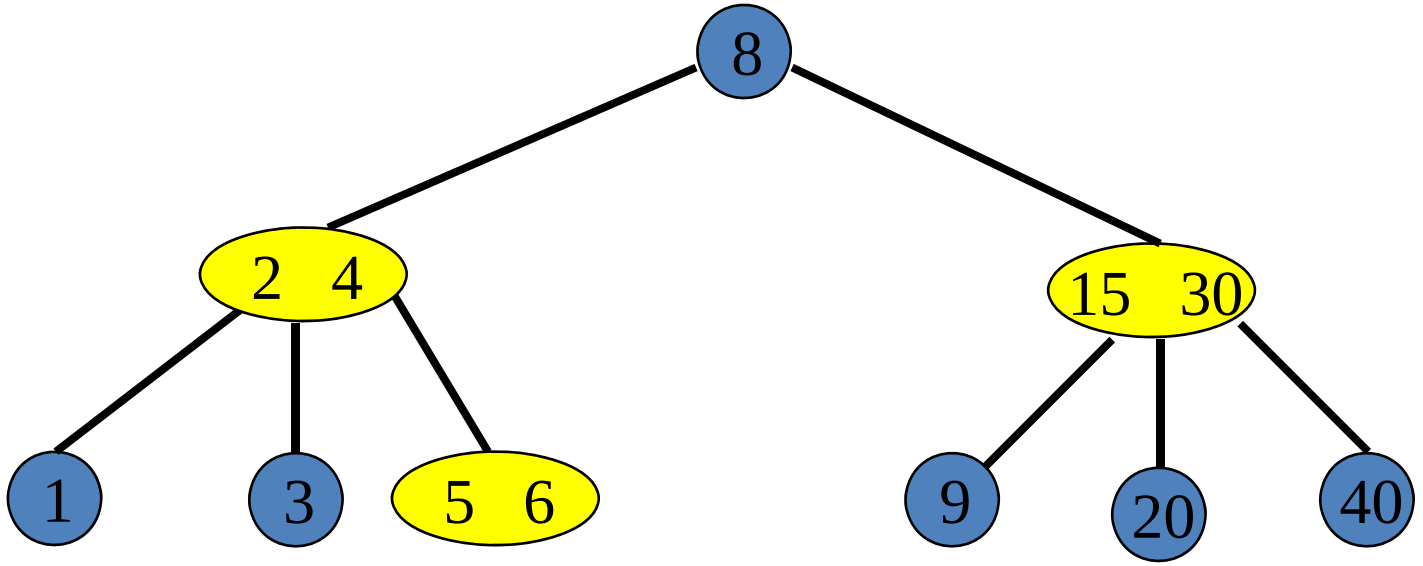


(b) *p* is the middle child of *r*



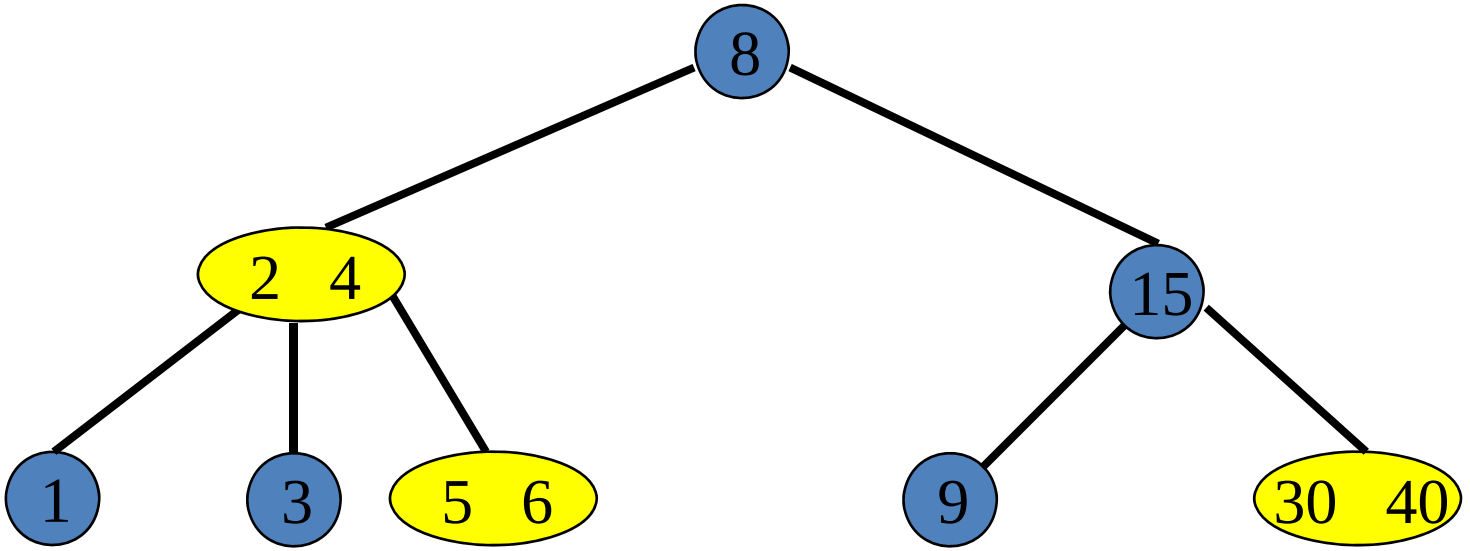
(c) *p* is the right child of *r*

Delete From A Leaf



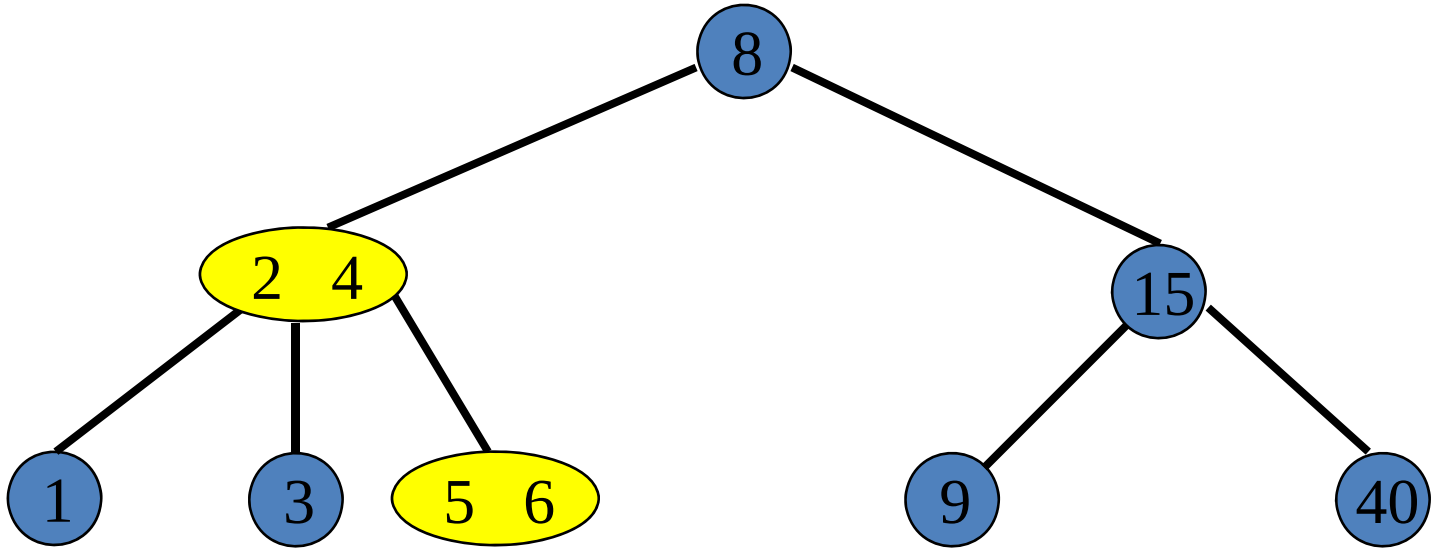
- Delete the pair with key = 20.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine sibling and parent pair.

Delete From A Leaf



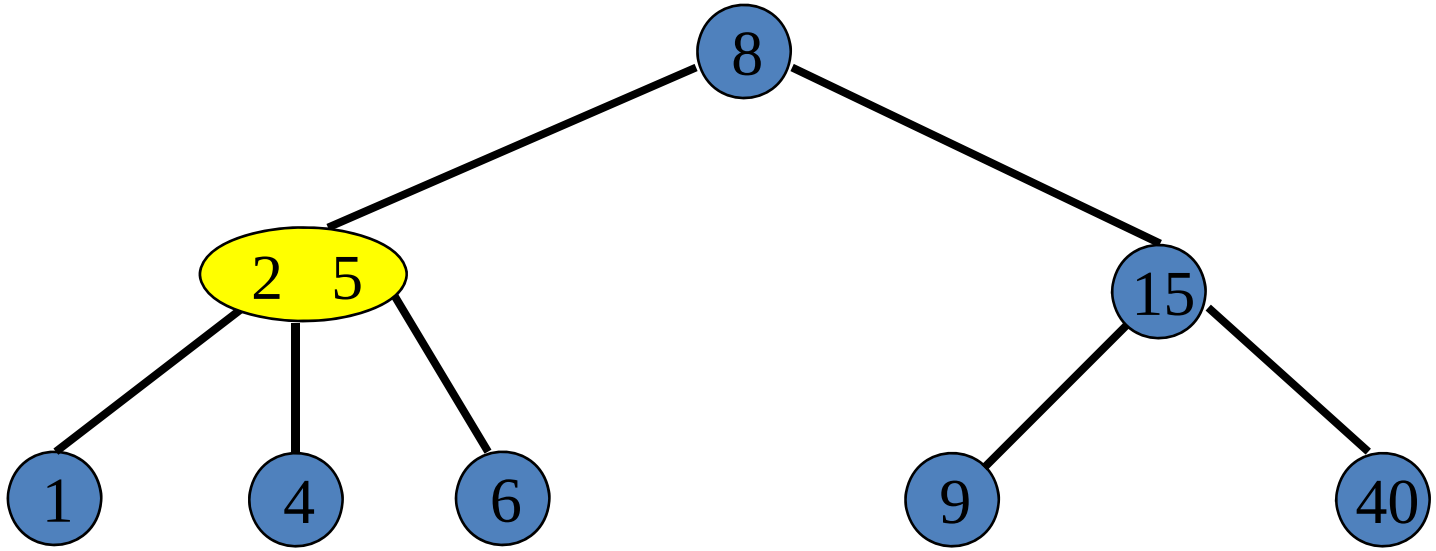
- Delete the pair with key = 30.
- Deletion from a 3-node.
- 3-node becomes 2-node.

Delete From A Leaf



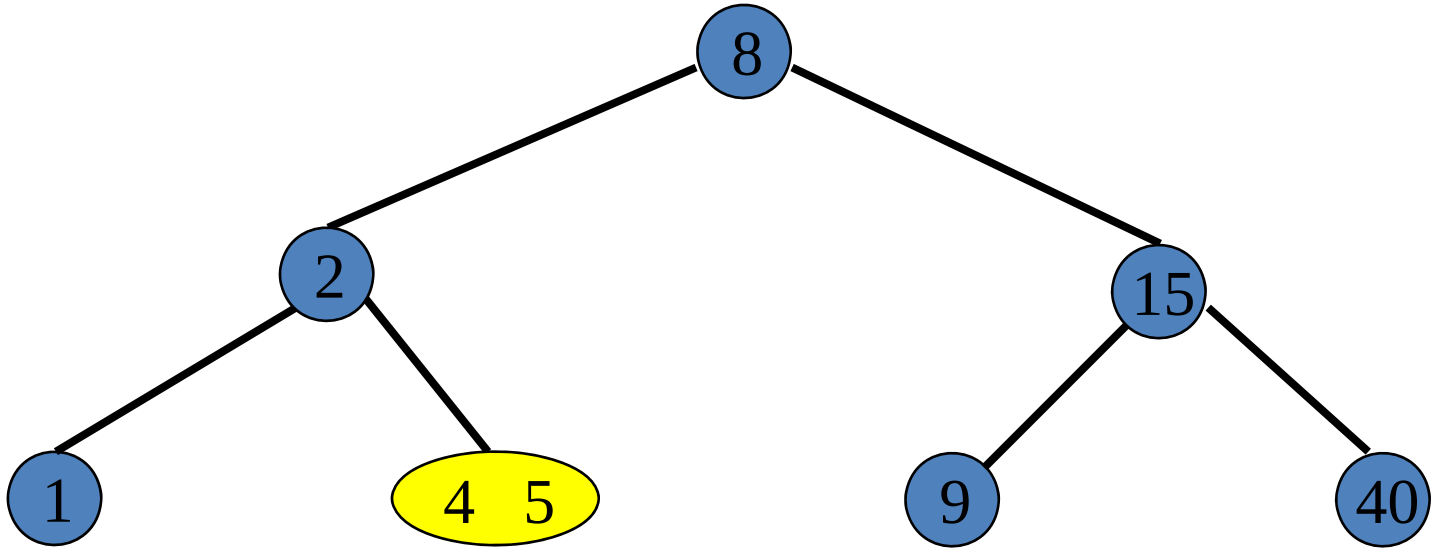
- Delete the pair with key = 3.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If so, borrow a pair via parent node.

Delete From A Leaf



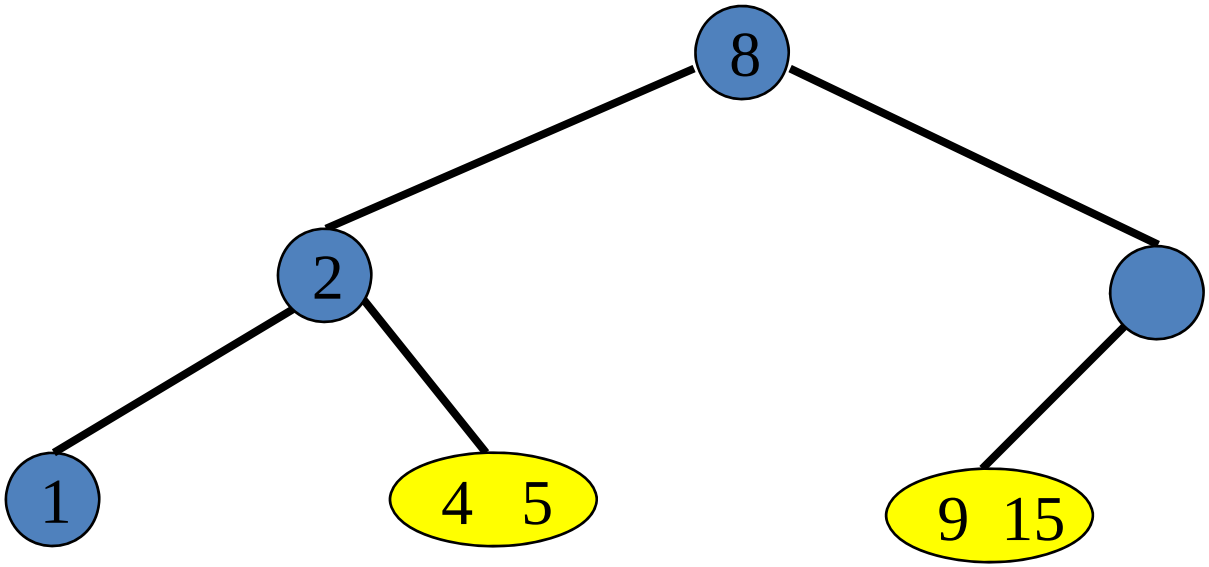
- Delete the pair with key = 6.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine sibling and parent pair.

Delete From A Leaf



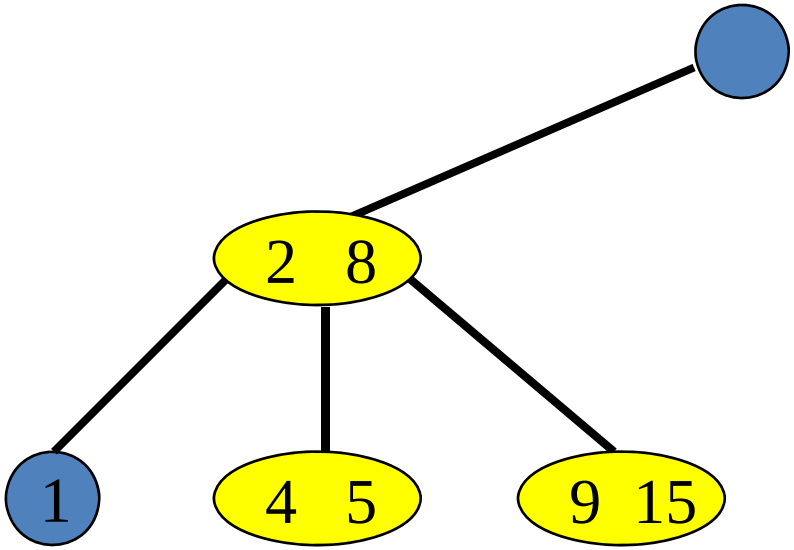
- Delete the pair with key = 40.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine sibling and parent pair.

Delete From A Leaf



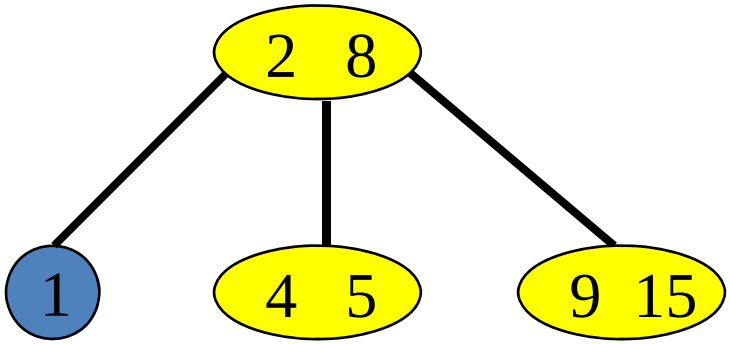
- Parent pair was from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



- Parent pair was from a 2-node.
- Check one sibling and determine if it is a 3-node.
- No sibling, so must be the root.
- Discard root. Left child becomes new root.

Delete From A Leaf



- Height reduces by 1.

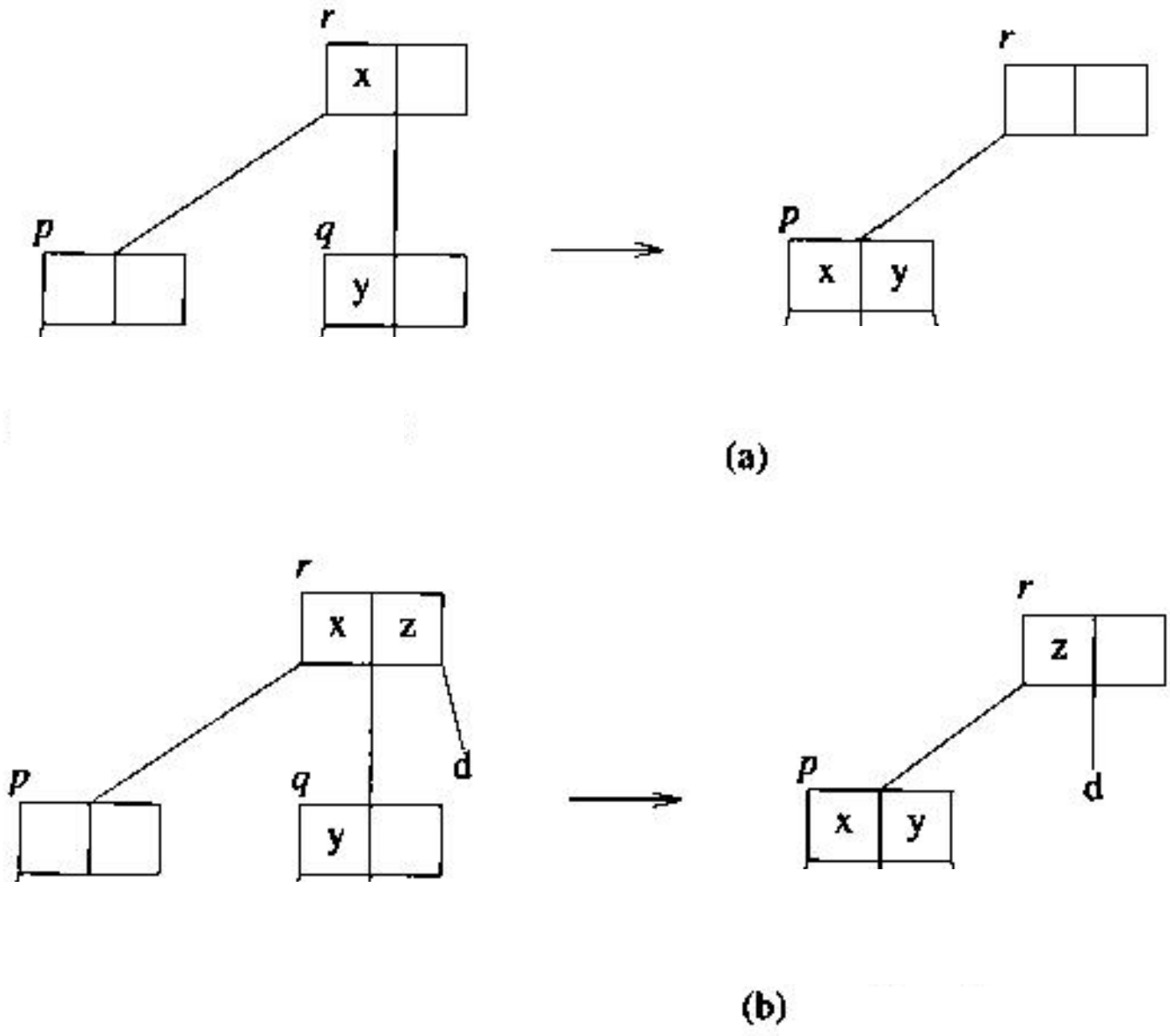


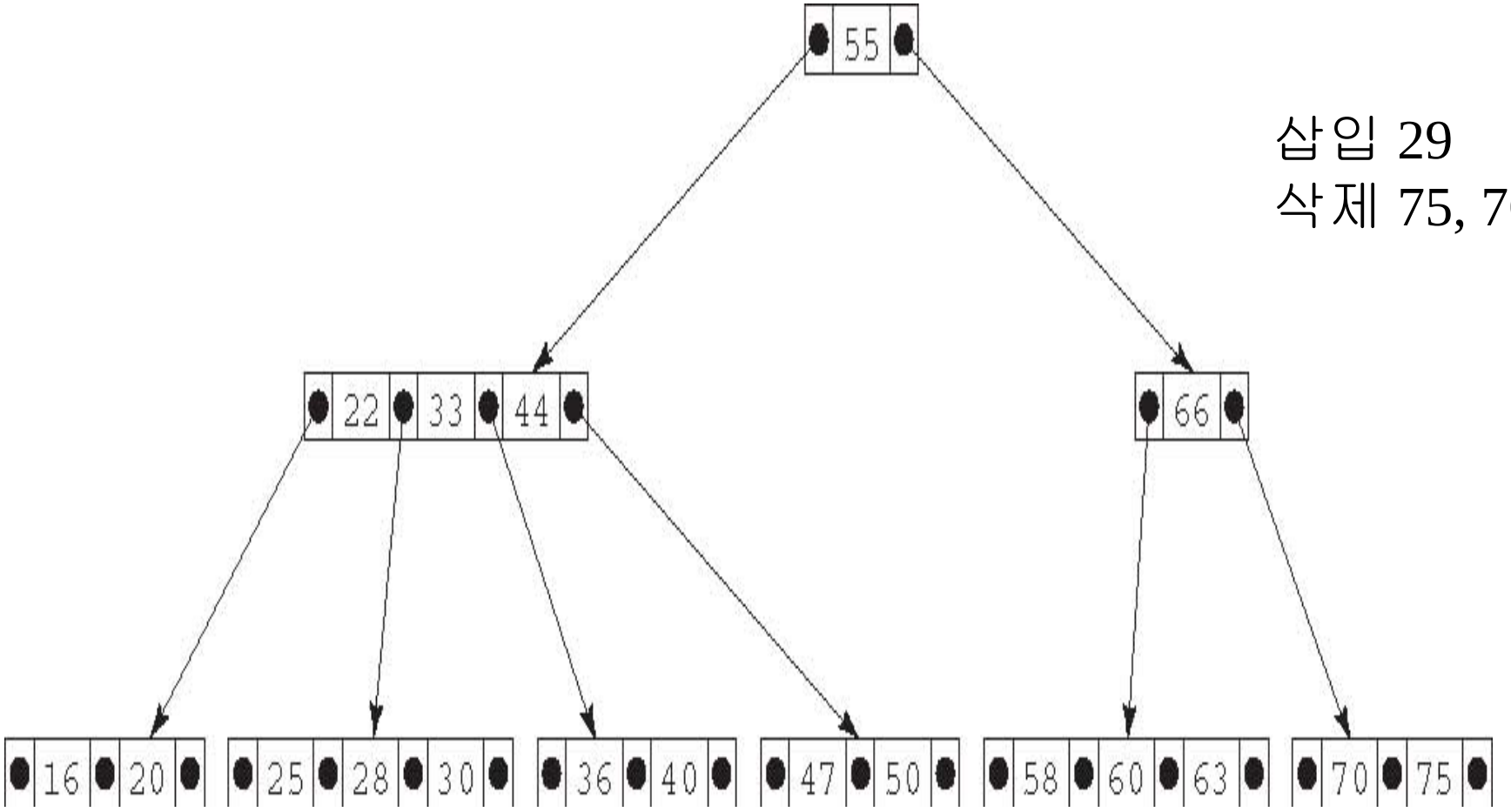
Figure 11.8: Combining in a 2-3 tree when p is the left child of r

B-트리 삭제 알고리즘

- 입력 : B-트리 T와 키 x.
- 출력 : 만일 T가 변경되었으면 true, 아니면 false.
- 후조건 : 키 x가 트리 T에 없음.

1. *x*를 가지고 있는 노드 *p*를 찾기 위해 다원 탐색 알고리즘을 적용.
2. 만일 *x*를 찾을 수 없으면, false를 리턴.
3. 만일 *p*가 리프가 아니면, *x*를 중위 후속자로 교체; 그러면 *x*는 그 후속자가 되고, *p*는 그것의 노드가 됨.
4. 만일 *p*가 최소가 아니면, *x*를 삭제하고 true를 리턴.
5. 만일 *p*의 형제들 중의 하나가 최소가 아니면, 그것을 *x*로 회전하고 true를 리턴.
6. *p*를 그것의 형제들 중의 하나 및 그들의 부모와 결합(join)한 다음, *x*를 삭제하고 true를 리턴.

높이가 2이고 차수가 4인 B-트리



삽입 29
삭제 75, 70

B-트리 검색

- B-트리 검색 알고리즘
  - 입력 : 다원 탐색 트리 T와 키 x.
  - 출력 : x가 T에 존재하는지 여부.

1. 만일 T가 NIL이라면, false를 리턴.
2. T의 루트에 있는 키 시이퀀스를 이진 탐색; i를 $k_{i-1} < x \leq k_i$에 대한 인덱스로 설정.
3. 만일 $x = k_i$이면, true를 리턴.
4. T를 서브트리 T_i로 설정.
5. x에 대해 T_i를 탐색한 결과 반환되는 값을 리턴.

2-3 And 2-3-4 Trees

- 차수가 *m*인 B-트리에서 루트가 아닌 모든 내부 노드는 최소한 ceil(m/2)의 차수를 가짐

- 2-3 tree is B-tree of order 3.
- 2-3-4 tree is B-tree of order 4.
- B-tree of order 5 is 3-4-5 tree (root may be 2-node though).

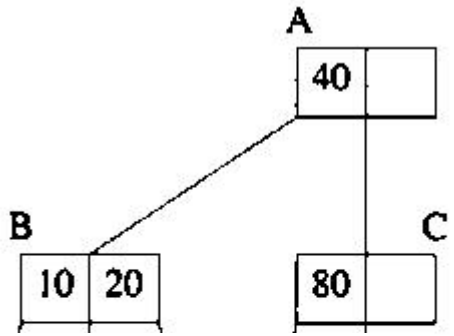


Figure 11.2: Example of a 2-3 tree

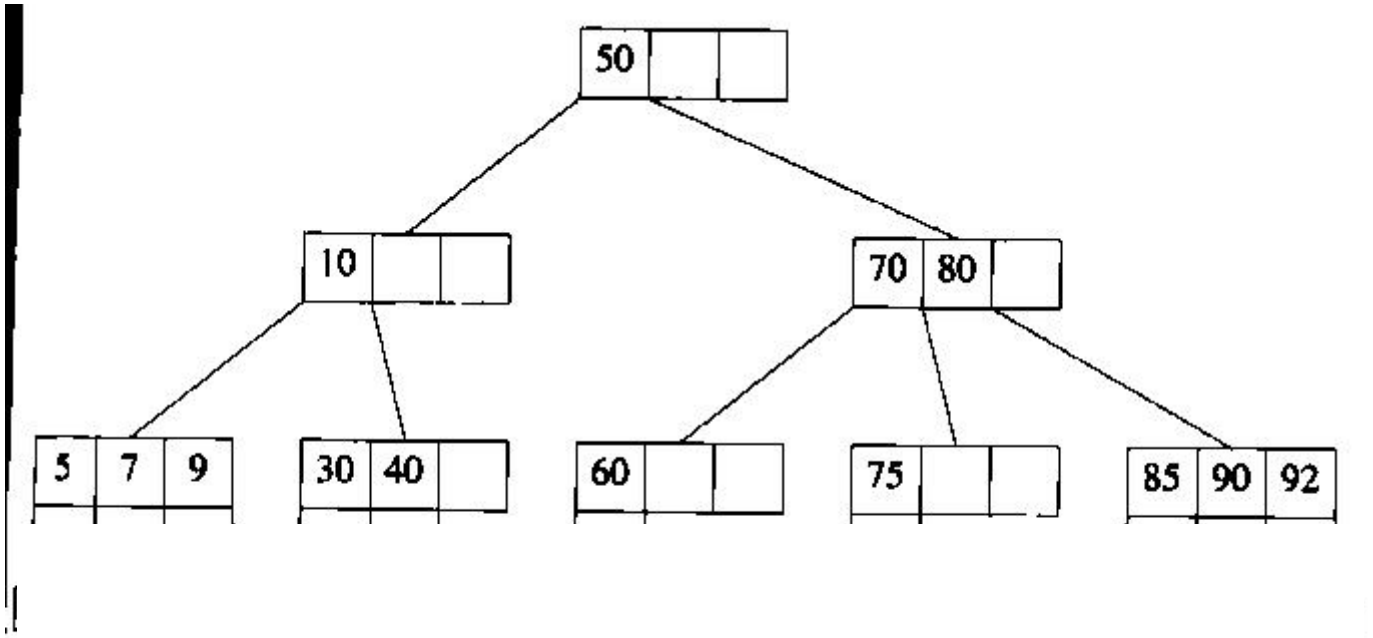


Figure 11.3: Example of a 2-3-4 tree