COSE213: Data Structure

Lecture 11 - 다원 탐색 트리 (Multiway Search Tree)

Minseok Jeon

2024 Fall

Review : 이진 탐색 트리 (Binary Search Tree)

- 이진 탐색 트리(BST, Binary Search Tree)는 이진트리 기반의 탐색을 위한 자료구조임
- 이진 탐색 트리는 다음과 같이 (재귀적으로) 정의됨

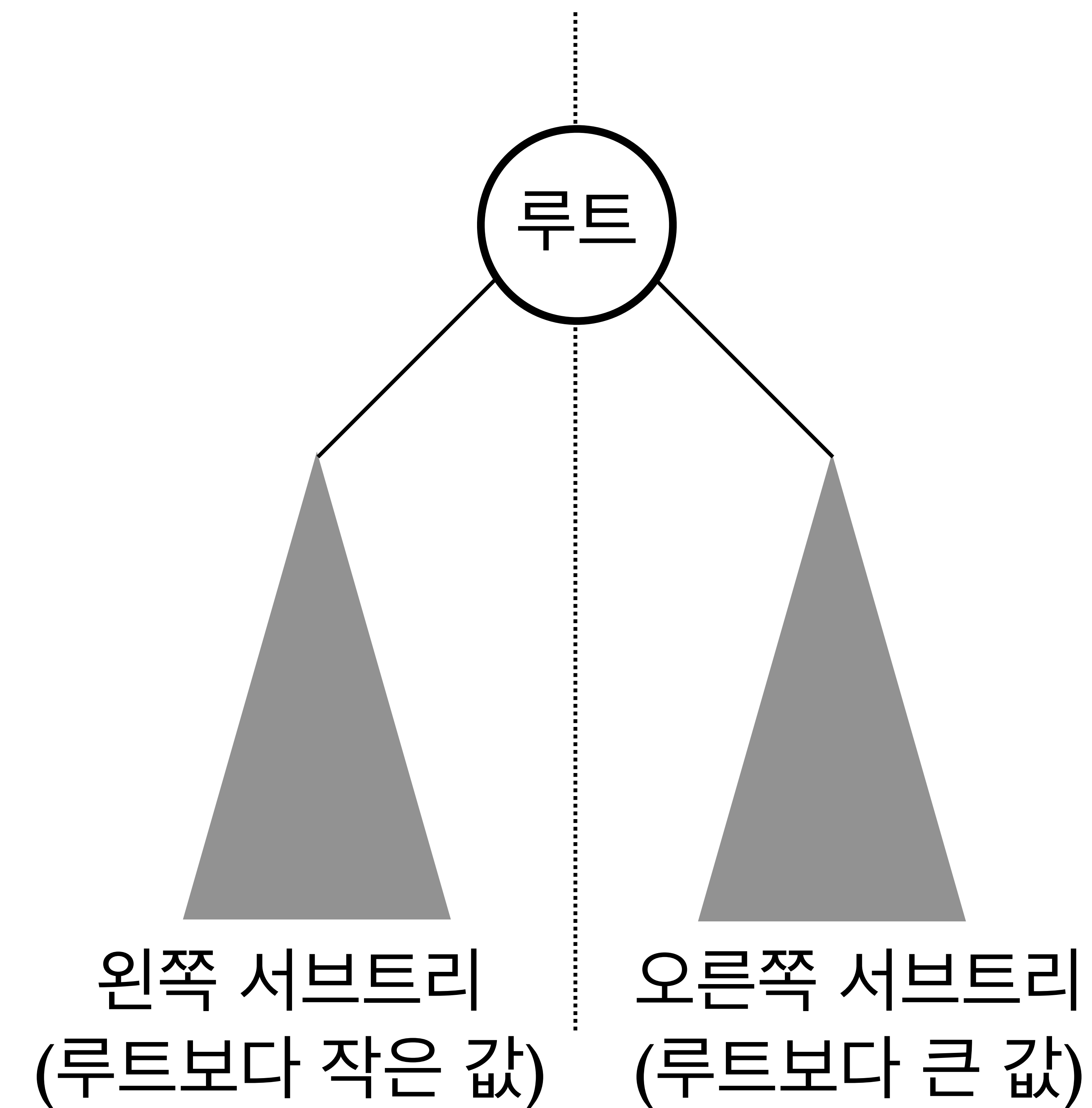
Definition

(1) 모든 노드는 유일한 키(key)를 갖는다.

(2) 왼쪽 서브트리의 키들은 루트의 키보다 작다.

(3) 오른쪽 서브트리의 키들은 루트의 키보다 크다.

(4) 왼쪽과 오른쪽 서브트리도 이진 탐색 트리이다.

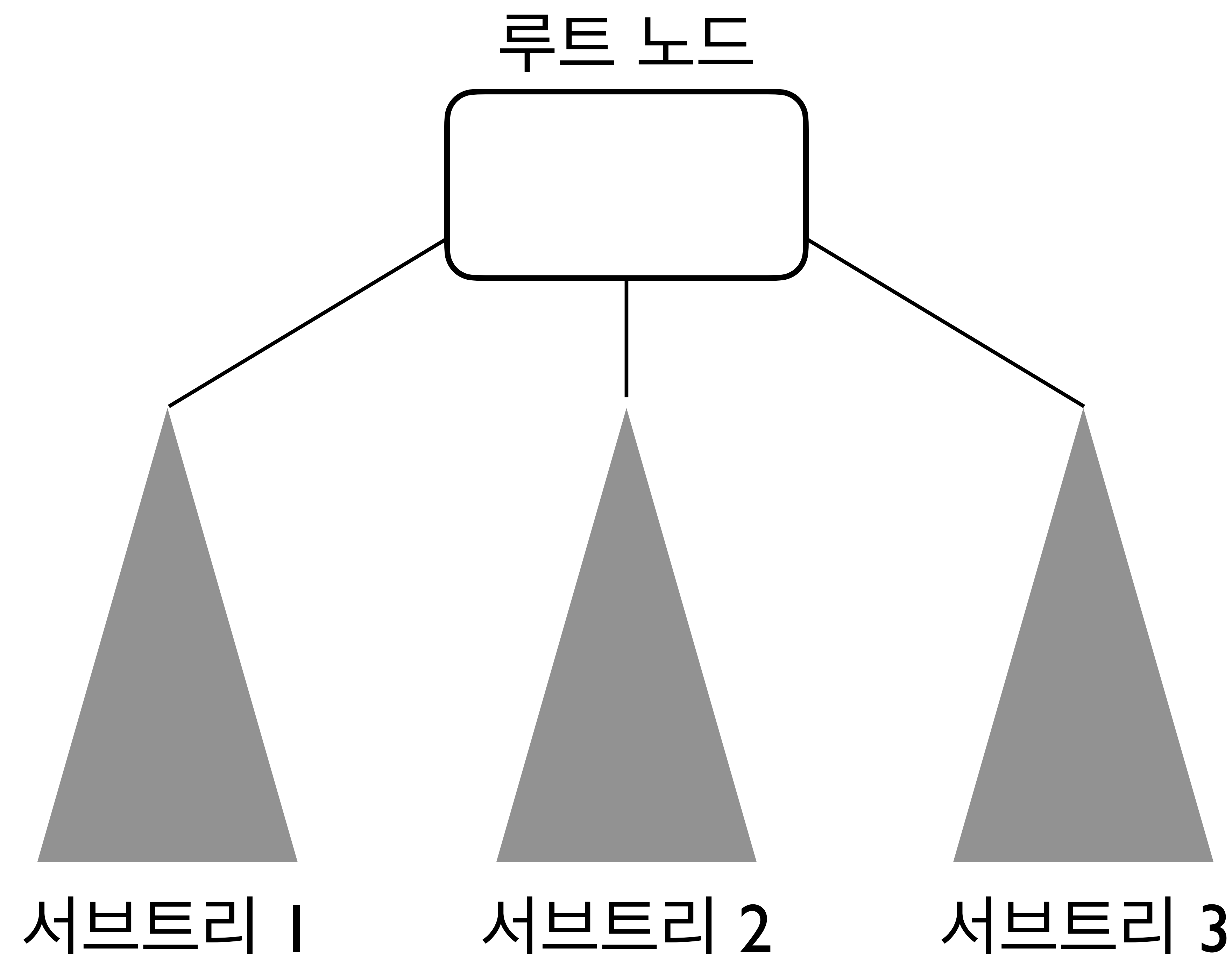


Review : 이진 탐색 트리 (Binary Search Tree)

- 이진 탐색 트리의 노드는 최대 2개의 서브트리를 가질 수 있음
  - 노드의 개수가 n인 이진탐색 트리의 높이는 최소 $\lceil \log_2(n+1) \rceil$임

Review : 이진 탐색 트리 (Binary Search Tree)

- 이진 탐색 트리의 노드는 최대 2개의 서브트리를 가질 수 있음
    - 노드의 개수가 n인 이진탐색 트리의 높이는 최소 $\lceil \log_2(n+1) \rceil$임
- 질문: 노드의 서브트리가 3개 이상인 탐색 트리를 만들어 탐색 트리의 높이를 줄일 수는 없을까?

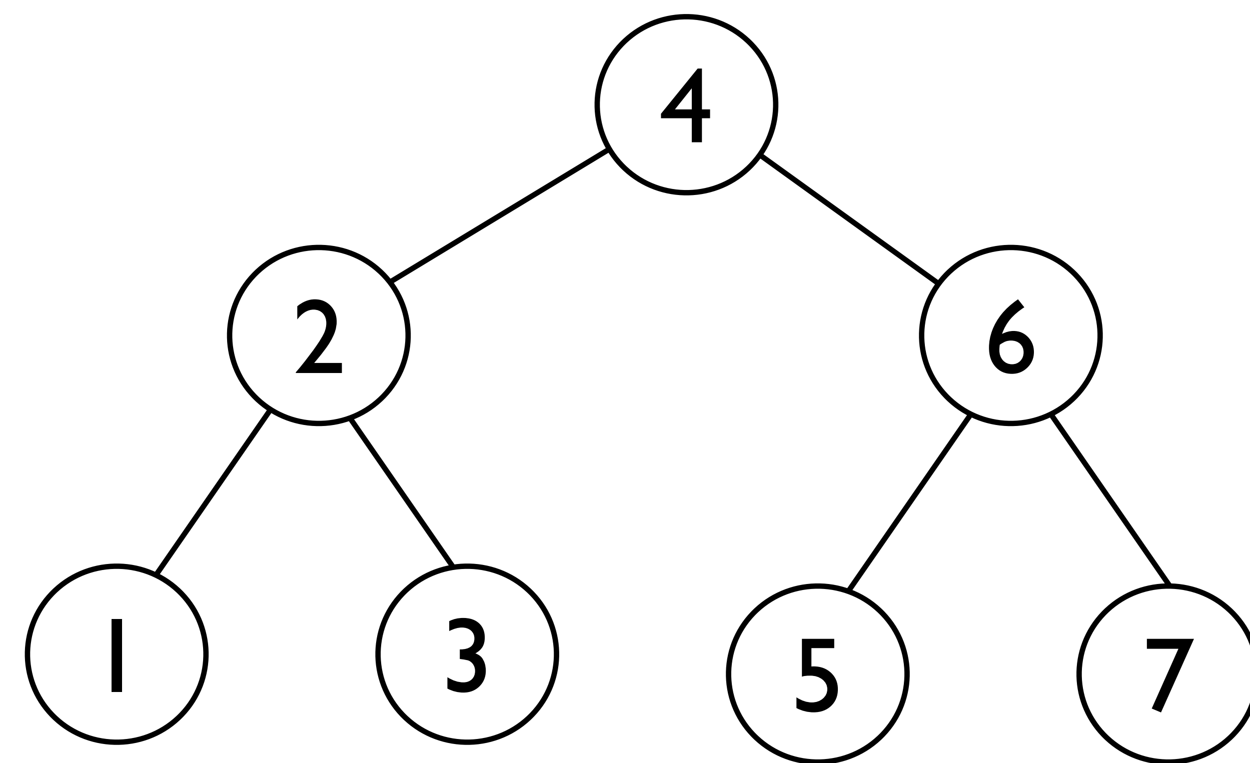


노드별 최대 3개의 서브트리를 가지는 경우

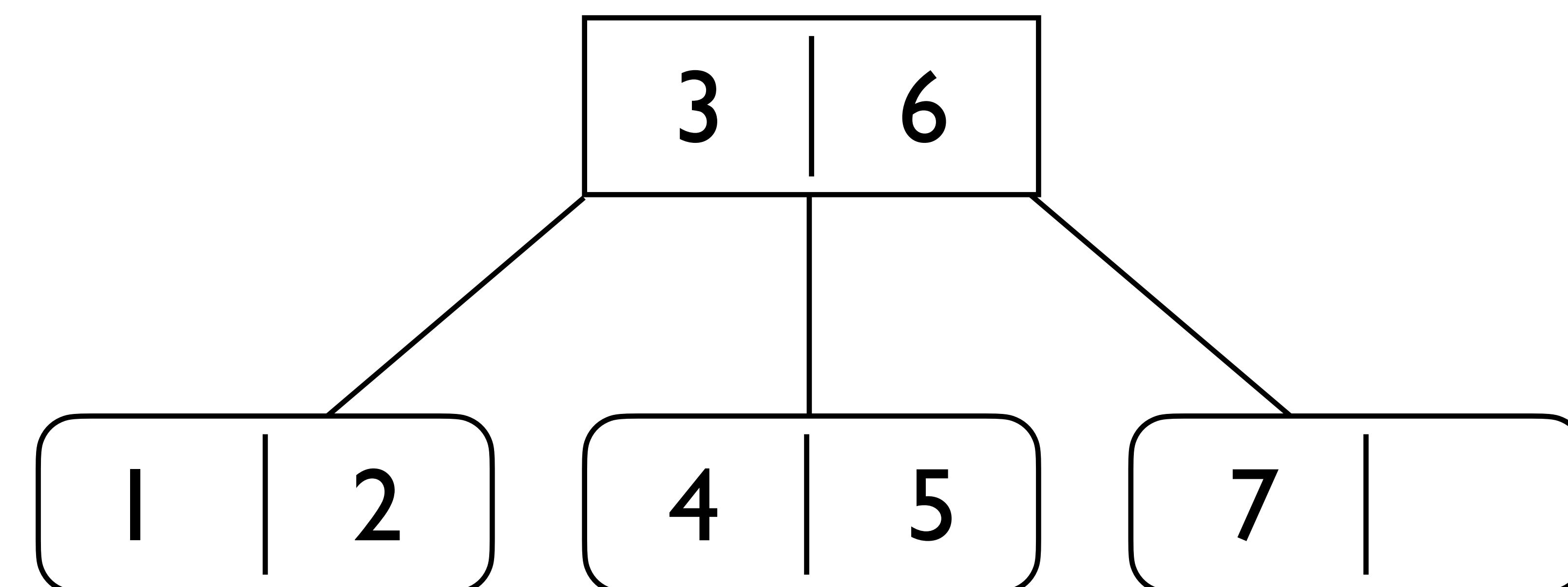
최소 높이 : $\lceil \log_3(n+1) \rceil$

Review : 이진 탐색 트리 (Binary Search Tree)

- 이진 탐색 트리의 노드는 최대 2개의 서브트리를 가질 수 있음
  - 노드의 개수가 n인 이진탐색 트리의 높이는 최소 $\lceil \log_2(n+1) \rceil$임
- 질문: 노드의 서브트리가 3개 이상인 탐색 트리를 만들어 탐색 트리의 높이를 줄일 수는 없을까?



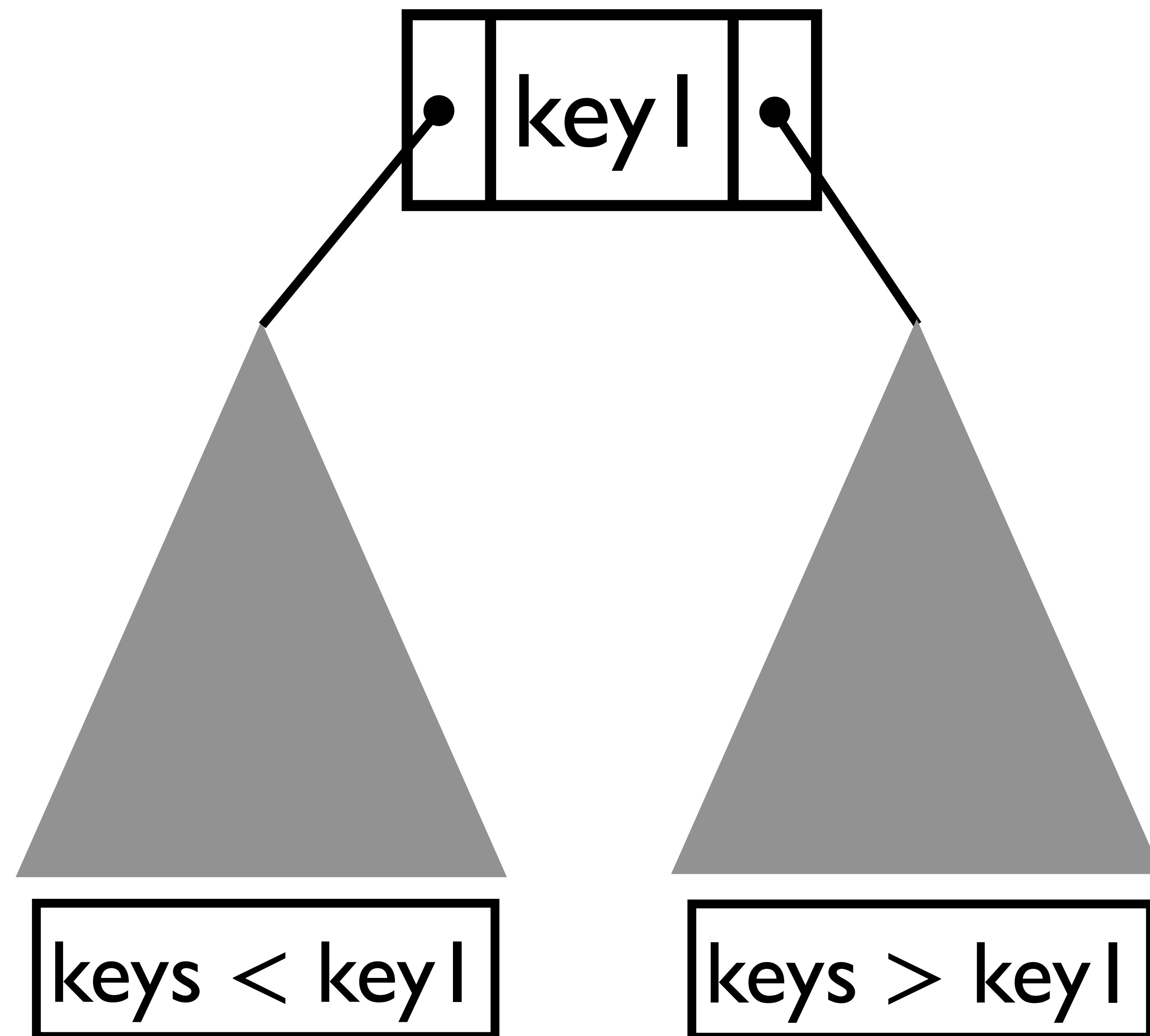
높이가 3인 이진 탐색 트리



높이가 2인 다원 탐색 트리

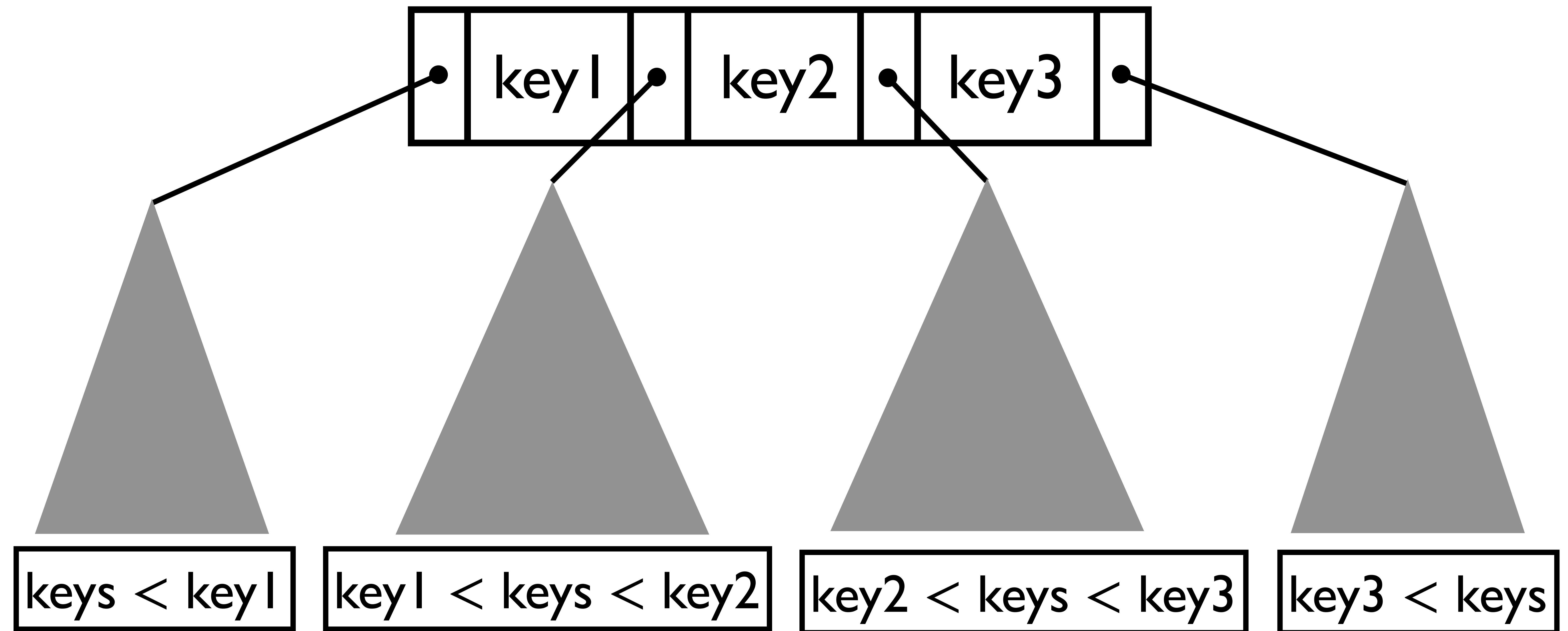
다원 탐색 트리 (Multiway Search Tree)

- 다원 탐색 트리(multi way search tree)는 이진 탐색 트리(binary search tree)의 일반화임
- 다원 탐색 트리의 장점:
  - 한 노드에 여러가지 키를 저장할 수 있어 트리의 높이를 낮출 수 있음 => 트리의 검색 속도를 높일 수 있음



이진 탐색 트리

vs



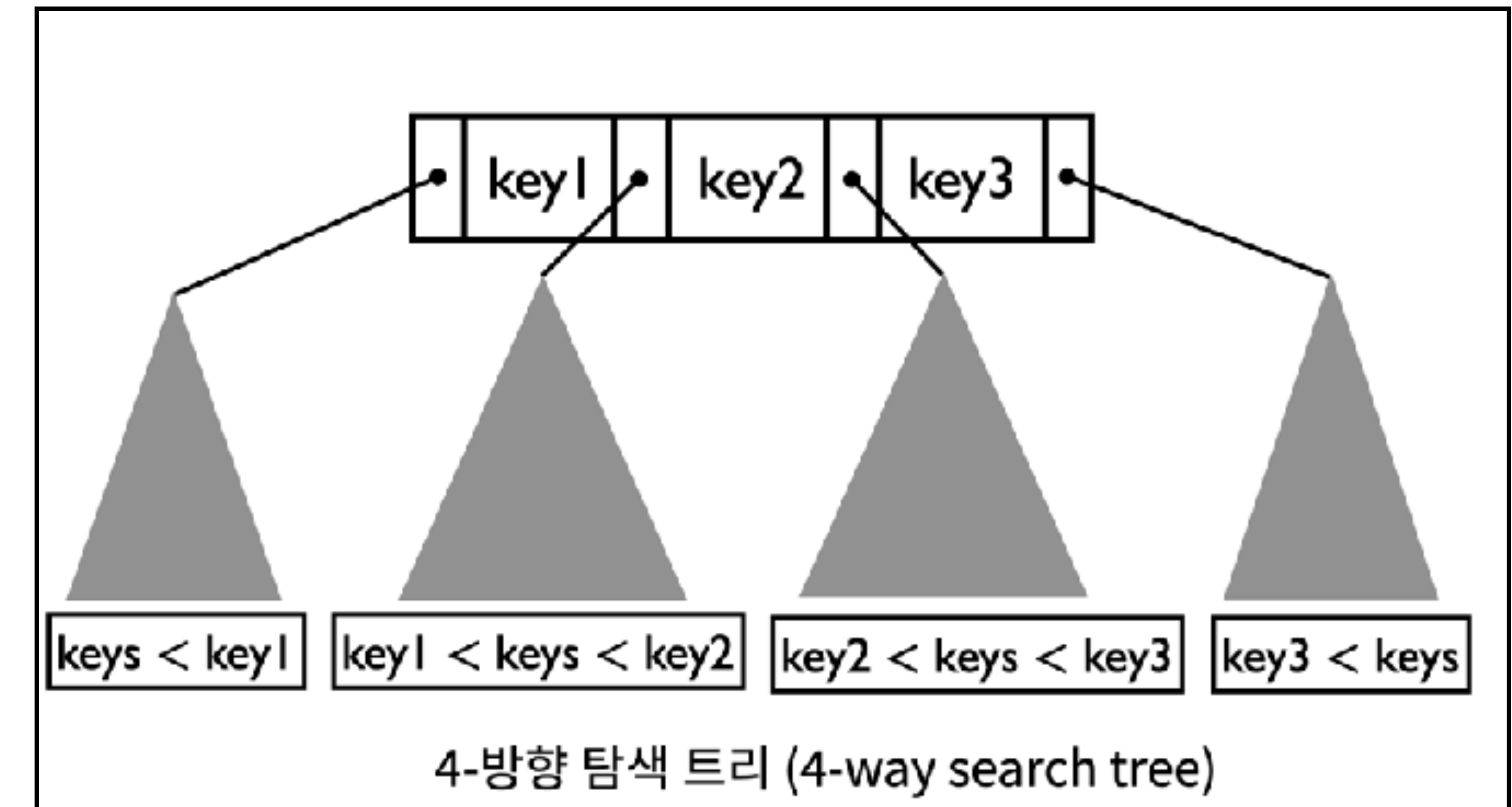
4-방향 탐색 트리 (4-way search tree)

다원 탐색 트리 (Multiway Search Tree)

- m-way 탐색 트리는 다음과 같은 특징을 가진 트리 자료구조임
  - 각 노드가 0에서 m개까지의 서브트리를 가질 수 있음
  - m개의 서브트리를 가지는 노드는 m-1개의 요소(key)를 가짐
  - 각 노드의 키는 오름차순으로 정렬되어 있음
  - 노드의 i번째 키의 값을 key_i, i번째 서브트리의 노드들이 가진 모든 키값들의 집합을 $Skeys_i$라고 할 때

$$\forall i \in \{1,2,\ldots,m-1\} . \forall k \in Skeys_i . \forall k' \in Skeys_{i+1} . k < key_i < k'$$

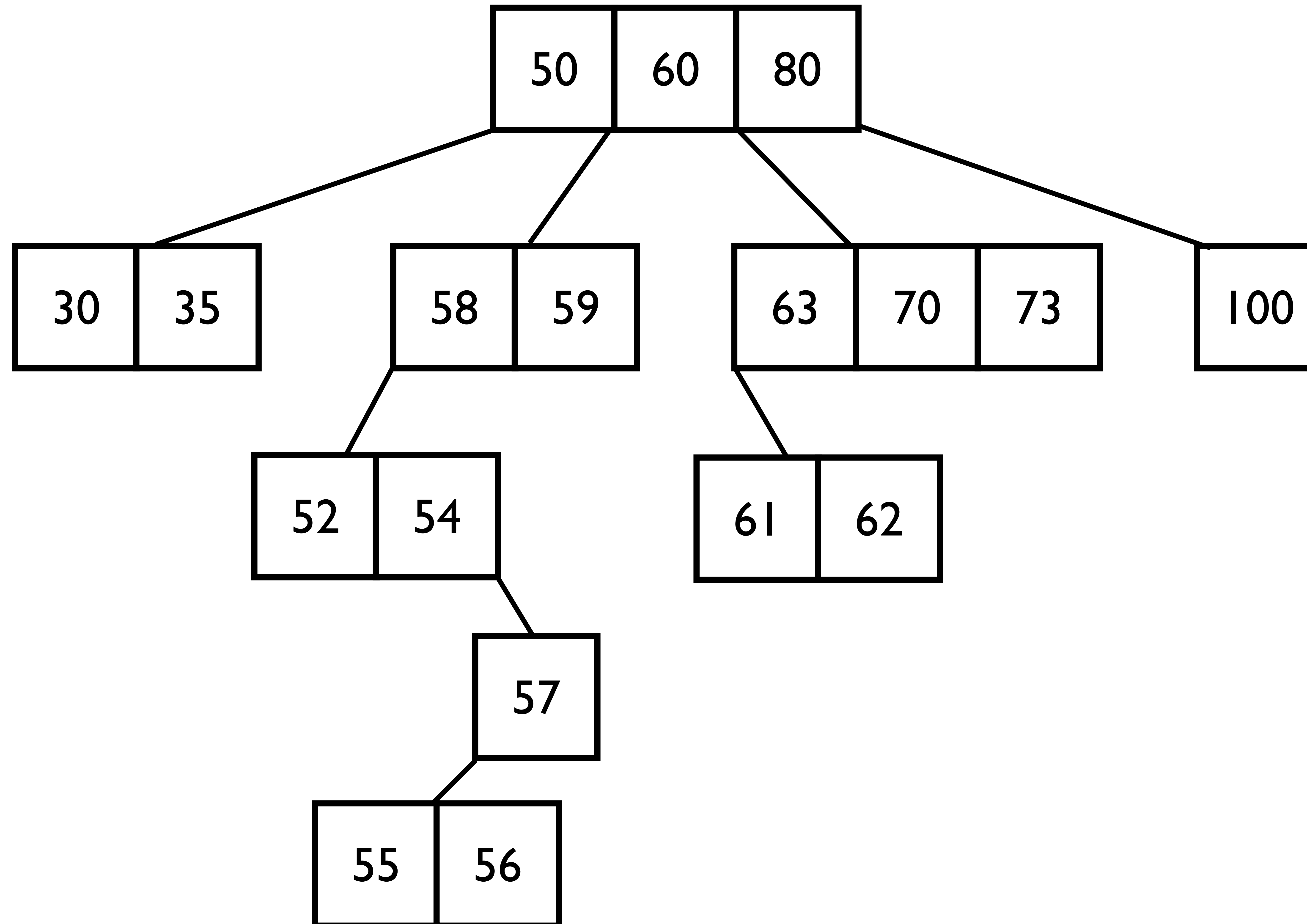
- 이진 탐색 트리는 m의 값이 2인 다원 탐색 트리임



4-방향 탐색 트리 (4-way search tree)

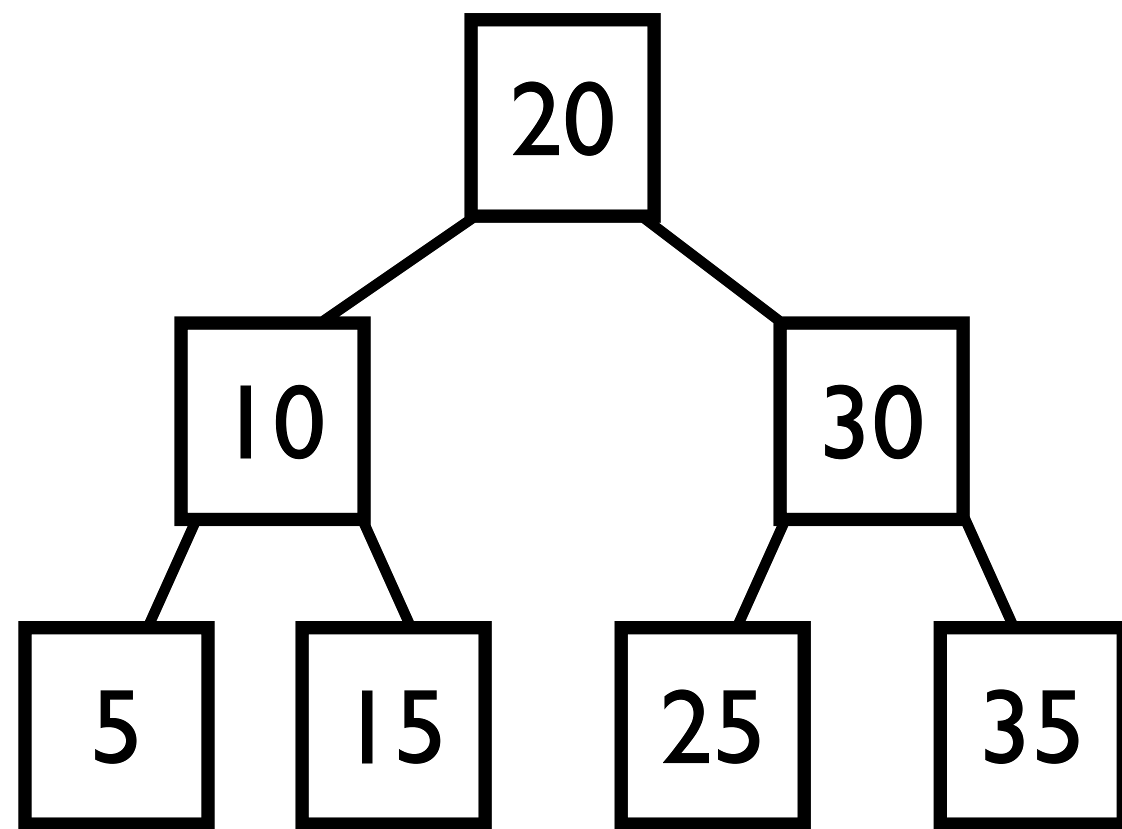
다원 탐색 트리 예시

- 차수가 5(m=5)인 다원 탐색 트리

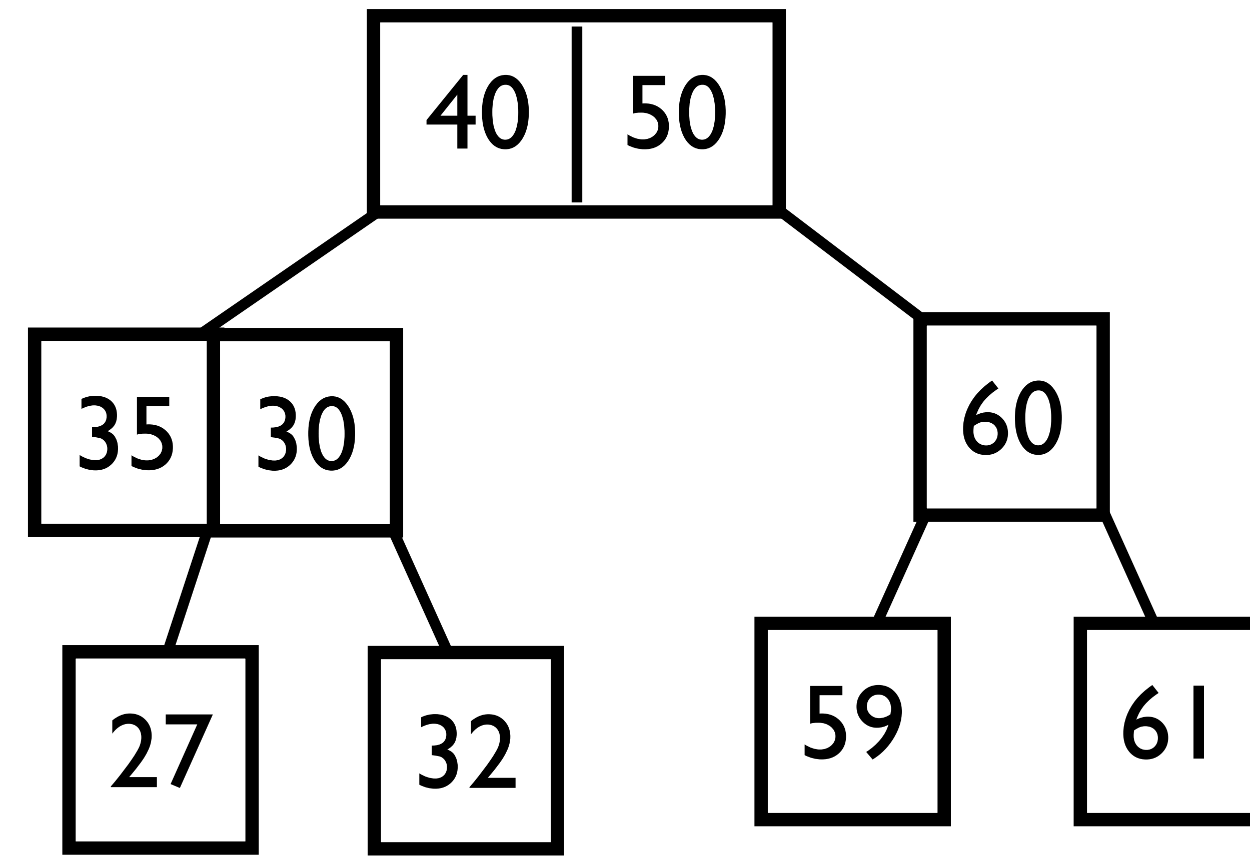


다원 탐색 트리 예시

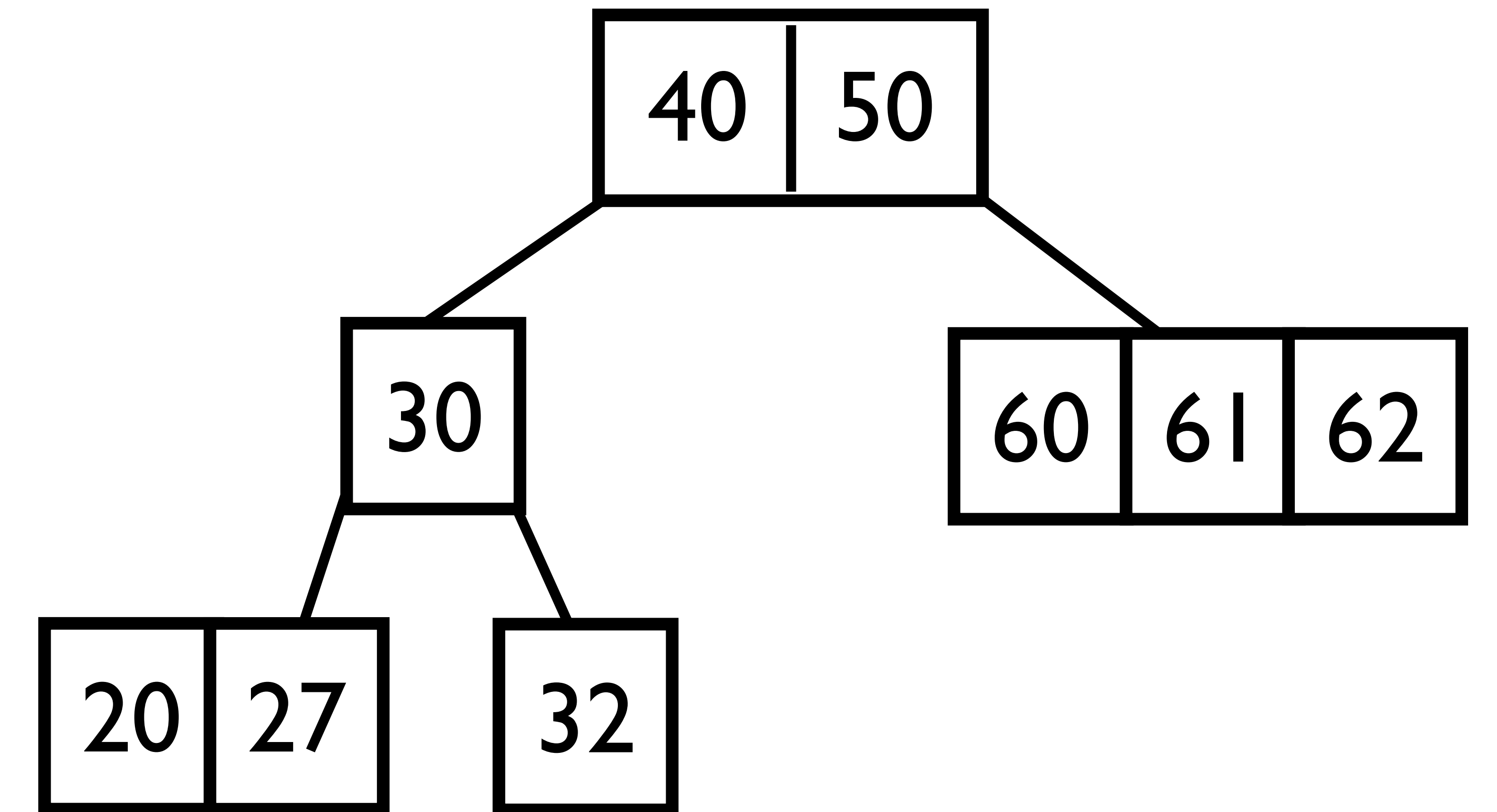
- Example: 다음 중 m이 3인 다원 탐색 트리인 것들과 아닌 것들을 분류하고 이유를 기술하기



(a)



(b)



(c)

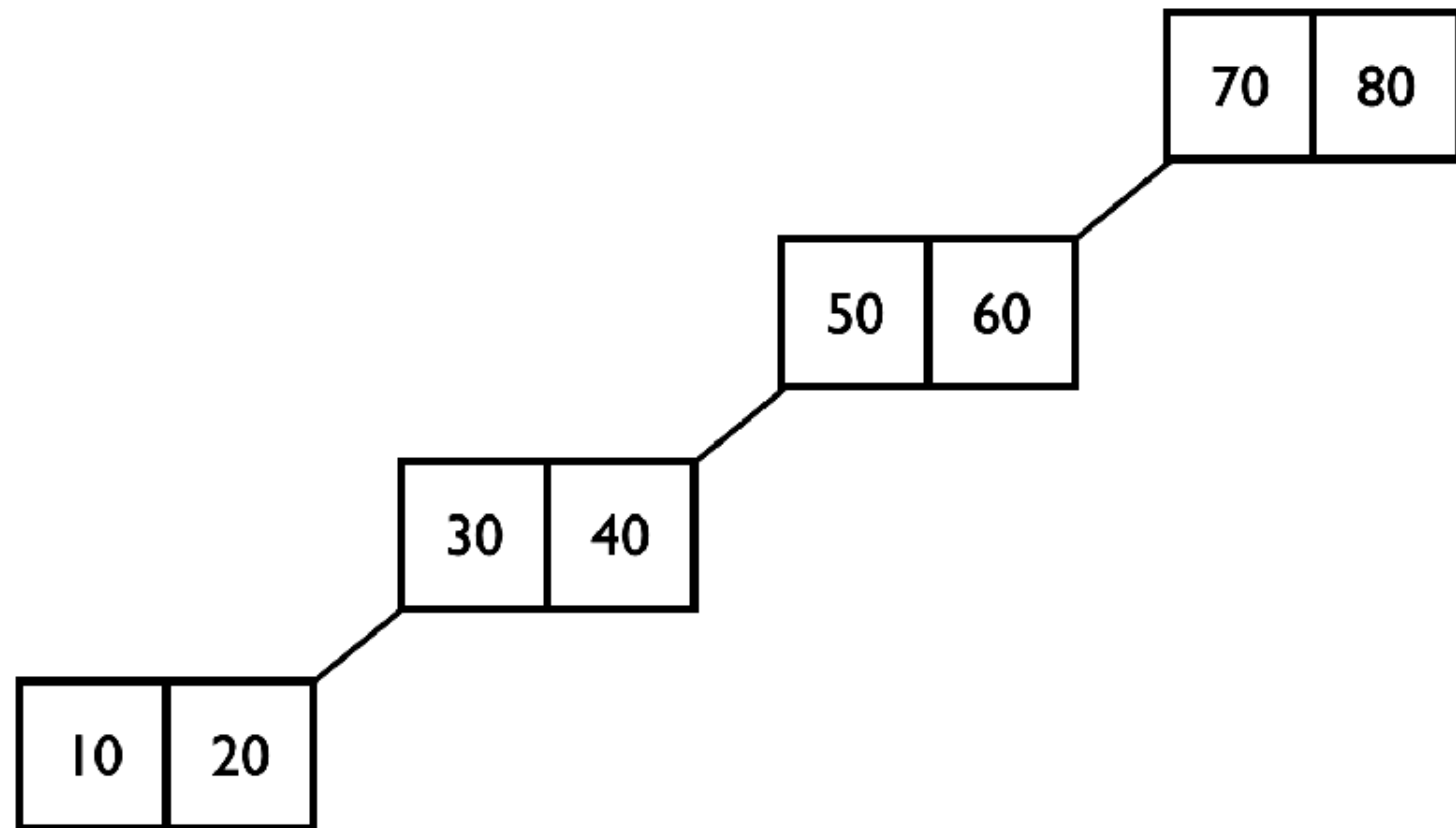
다원 탐색 트리 예시

- Example 1: 타원 탐색 트리의 차수가 3이고 키 값으로 1, 2, 3, 4, 5, 6, 7, 8을 가질 때 높이가 최소가 되는 다원 탐색 트리 그리기

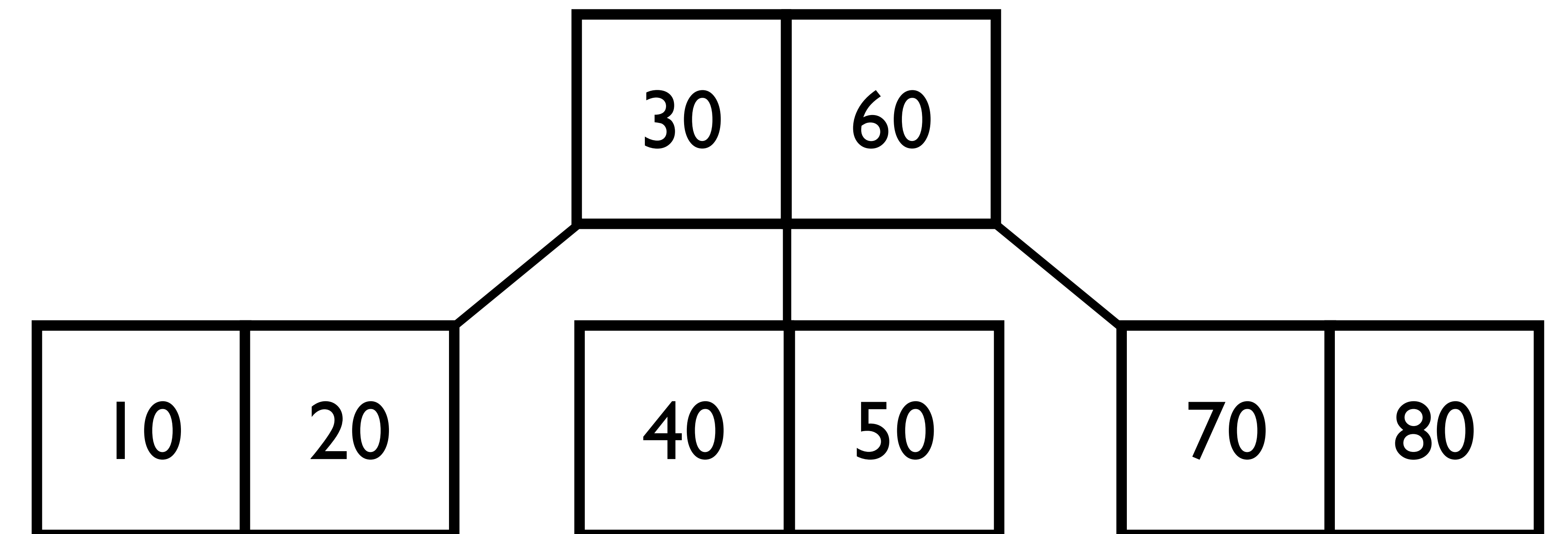
- Example 2: 타원 탐색 트리의 차수가 3이고 키 값으로 1, 2, 3, 4, 5, 6, 7, 8을 가질 때 높이가 최대가 되는 다원 탐색 트리 그리기

다원 탐색 트리의 문제점

- 다원 탐색 트리도 이진 탐색 트리가 가지는 문제점을 똑같이 가지고 있음:
    - 트리의 밸런스가 무너져있는 경우 탐색이 오래 걸릴 수 있음

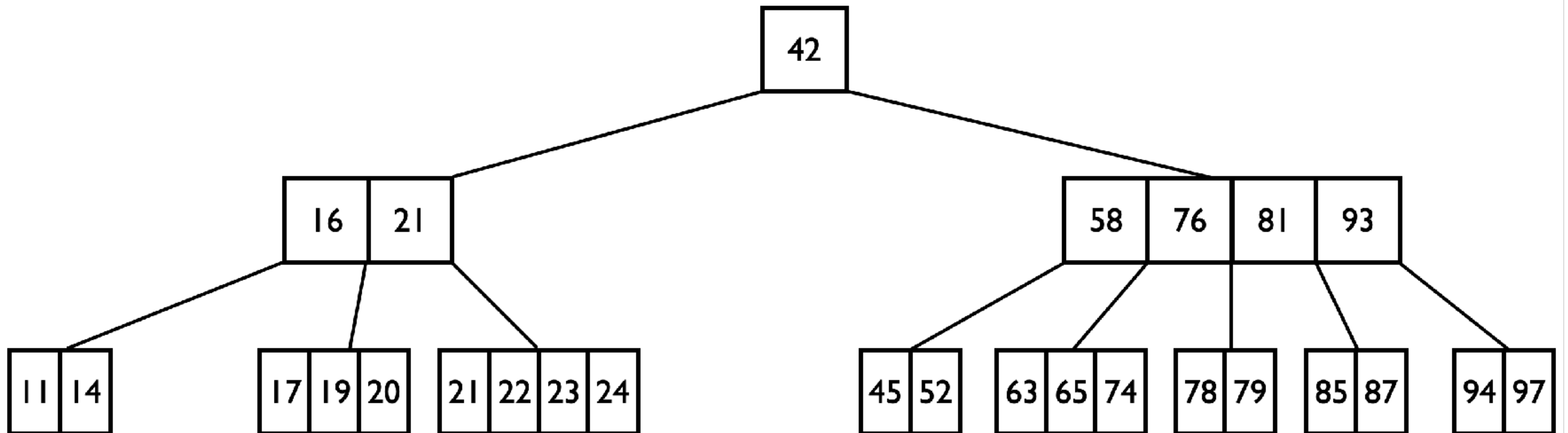


VS



해결책 : B-Tree

- B-Tree는 항상 균형을 유지하는 다원 탐색 트리임



차수가 5(m = 5)인 B-tree 예시

해결책 : B-Tree

- B-Tree는 다음의 조건들을 만족하는 다원 탐색 트리(M-way search tree)임

(1) 루트(root) 노드가 리프 노드가 아닌 경우 항상 2개 이상의 서브트리를 가짐

(2) 차수가 m인 B-tree의 경우 루트(root)와 리프(leaf) 노드를 제외한 내부(internal) 노드들은 최소 ⌈m/2⌉개 최대 m개의 서브트리를 가짐

(3) 루트 노드를 제외한 모든 노드들은 최소 ⌈m/2⌉-1개 최대 m-1 개의 키값을 가짐

(4) B-tree는 균형이 잡혀있으며 모든 리프 노드가 같은 레벨에 위치함

B-Tree (Balanced Tree)

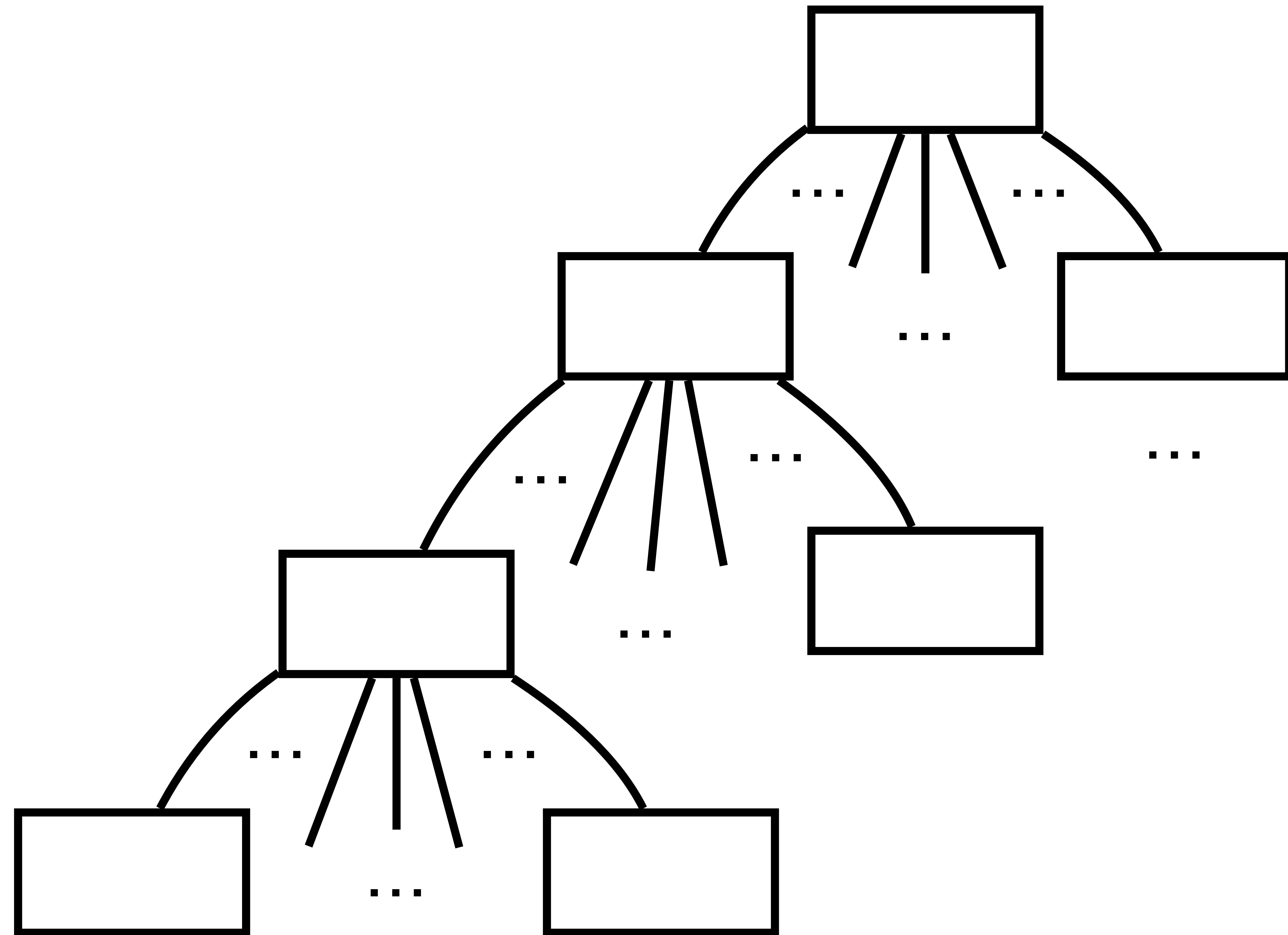
- Example: 차수가 4이고 높이가 2인 B-tree중 키값의 개수가 최대가 되는 B-tree를 그리시오.
    - 기술한 B-tree의 노드의 개수:
    - 기술한 B-tree에서 키값의 개수:

B-Tree (Balanced Tree)

- Example: 차수가 4이고 높이가 3인 B-tree중 키값의 개수가 최소가 되는 B-tree를 기술하시오.
  - 기술한 B-tree의 노드의 개수:
  - 기술한 B-tree에서 키값의 개수:

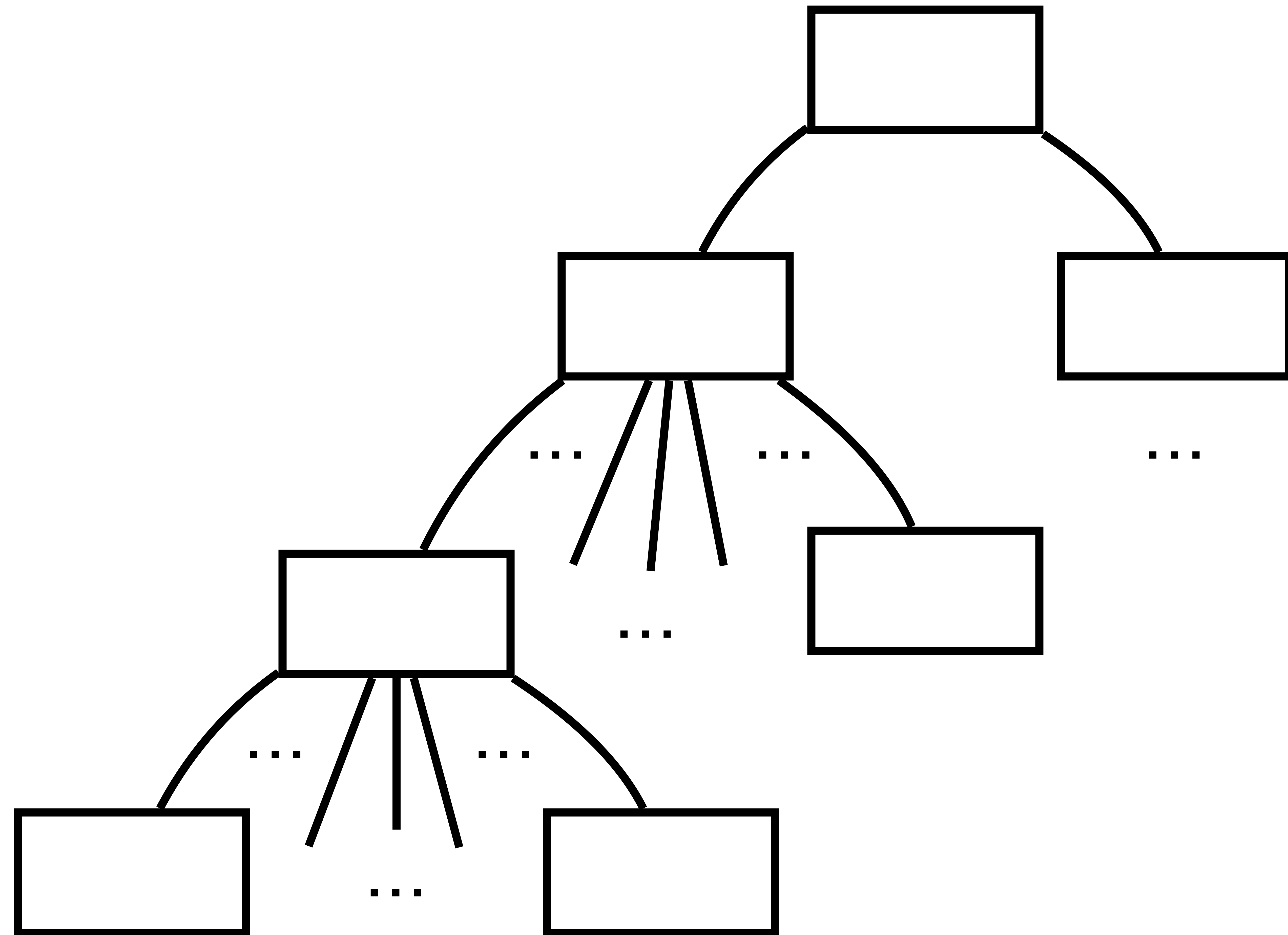
B-Tree (Balanced Tree)

- 차수가 101일때(m=101), 높이가 4인 B-tree가 가질 수 있는 키값의 최대 개수는?



B-Tree (Balanced Tree)

- 차수가 101일때(m=101), 높이가 4인 B-tree가 가질 수 있는 키값의 최소 개수는?



B-Tree의 추상 자료형

- B-tree는 다음과 같은 기능들을 제공하는 자료구조임
  - insert(tree, key): B-tree의 특성을 유지하면서 주어진 key를 tree에 삽입함
  - delete(tree, key): B-tree의 특성을 유지하면서 주어진 key를 tree에서 삭제함
  - search(tree, key): B-tree가 주어진 key를 가지고 있는 노드를 반환함
  - traversal(tree): B-tree가 가지고 있는 key값들을 순회함

B-Tree에서의 노드

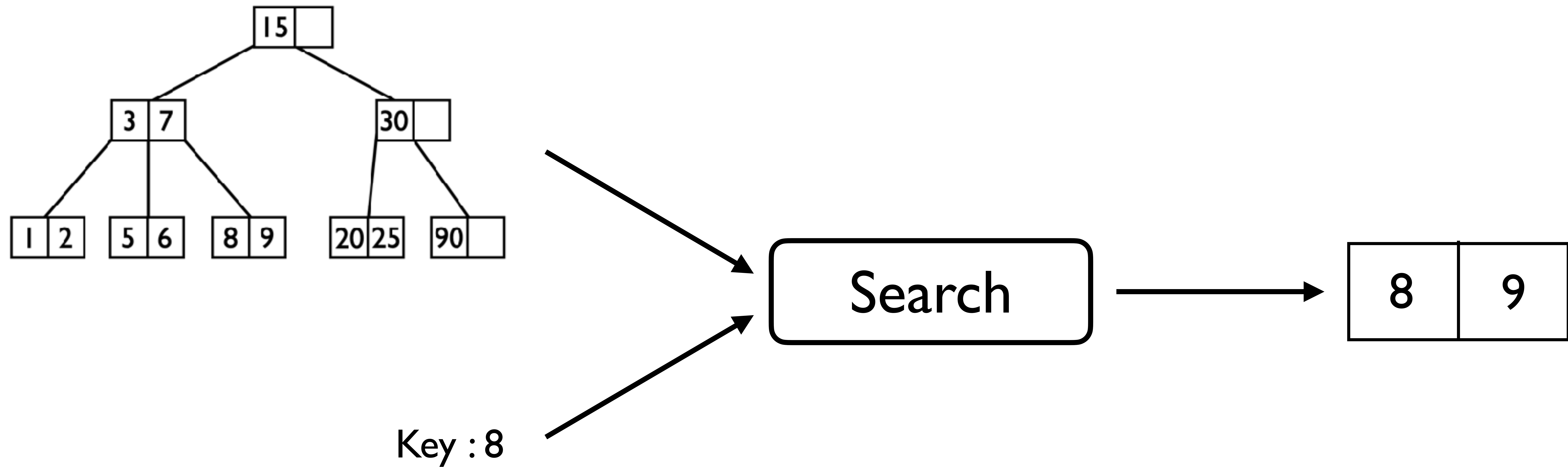
- B-Tree에서의 노드

```
#define M 3 // Order of the m-way tree

typedef struct Node {
    int keys[M - 1]; // Max number of keys in a node is M-1
    struct Node* children[M]; // Max number of children is M
    int num_keys; // Current number of keys
    bool is_leaf; // True if node is a leaf
} Node;
```

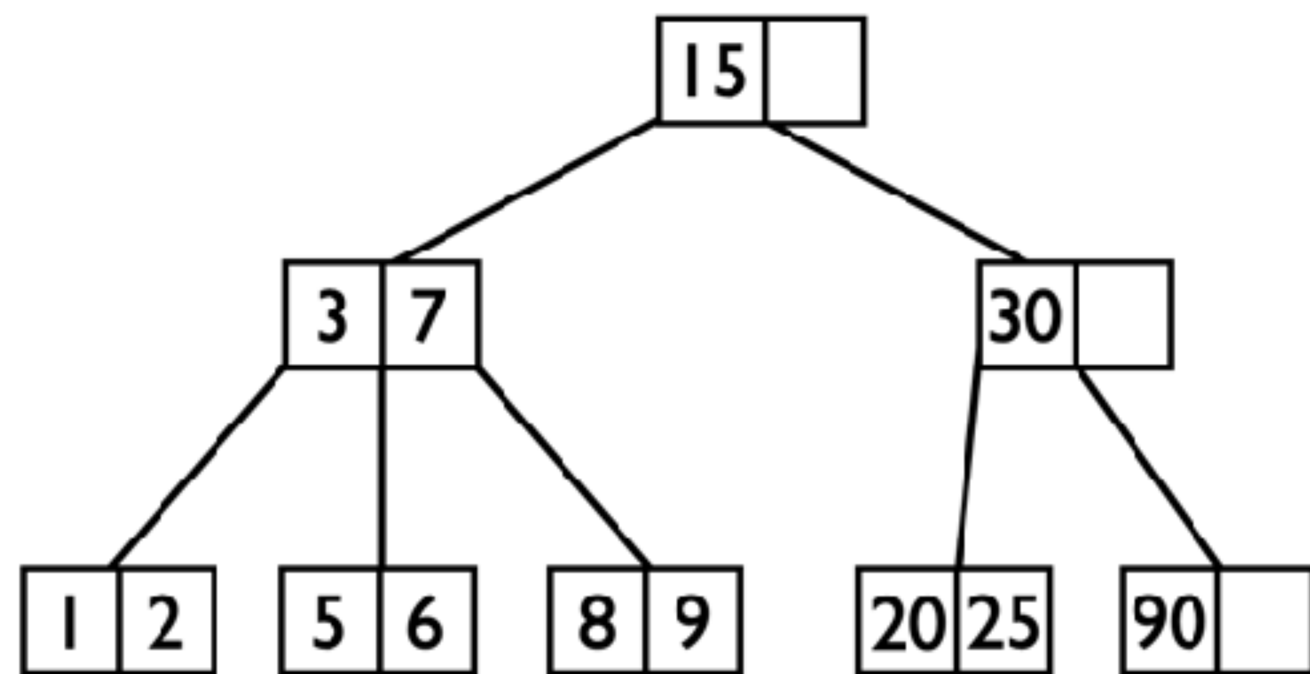
Search

- search(tree, key): B-tree가 주어진 key를 가지고 노드를 반환



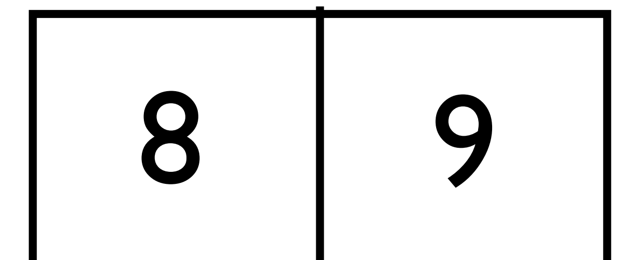
Search

- search(tree, key): B-tree가 주어진 key를 가지고 노드를 반환



Key : 8

```
procedure search(root, key)
  i ← 0
  while i < root.num_keys and key > root.keys[i] do
    i ← i + 1
  end while
  if key = root.keys[i] then
    return root
  end if
  if root.isleaf = true then
    return NULL
  end if
  return search(root.children[i], key)
end procedure
```



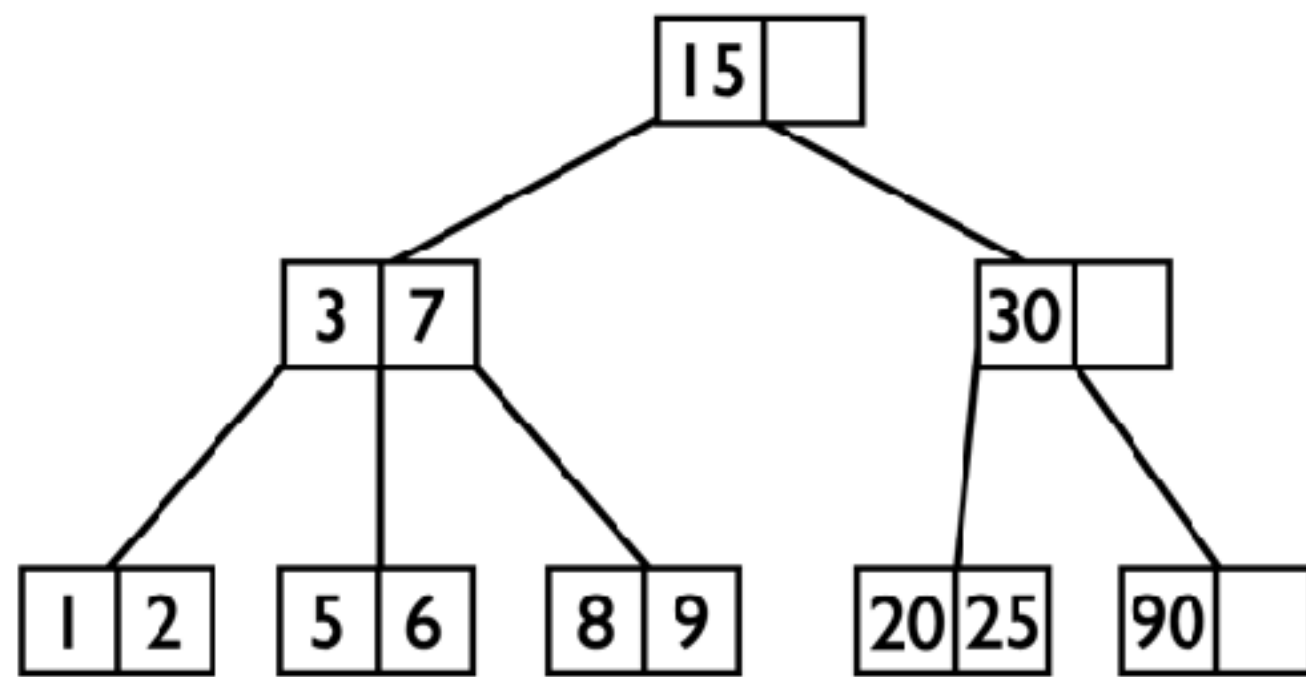
Search

- search(tree, key): B-tree가 주어진 key를 가지고 있는 노드를 반환함

```
procedure search(root, key)
  i ← 0
  while i < root.num_keys and key > root.keys[i] do
    i ← i + 1
  end while
  if key = root.keys[i] then
    return root
  end if
  if root.isleaf = true then
    return NULL
  end if
  return search(root.children[i], key)
end procedure
```

traversal

- traversal(tree): B-tree가 가지고 있는 key값들을 순회함

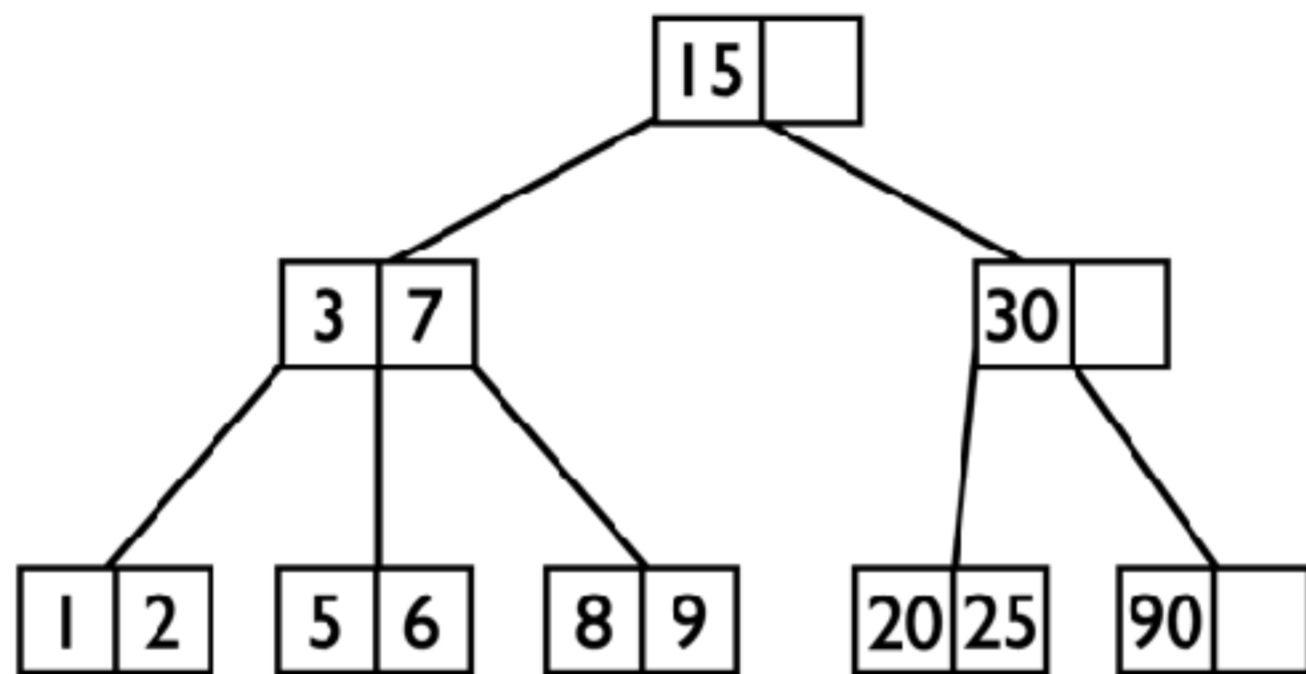


traversal

```
Keys in B-tree:
1, 2, 3, 5, 6, 7, 8,
9, 15, 20, 25, 30, 90
```

traversal

- traversal(tree): B-tree가 가지고 있는 key값들을 순회함

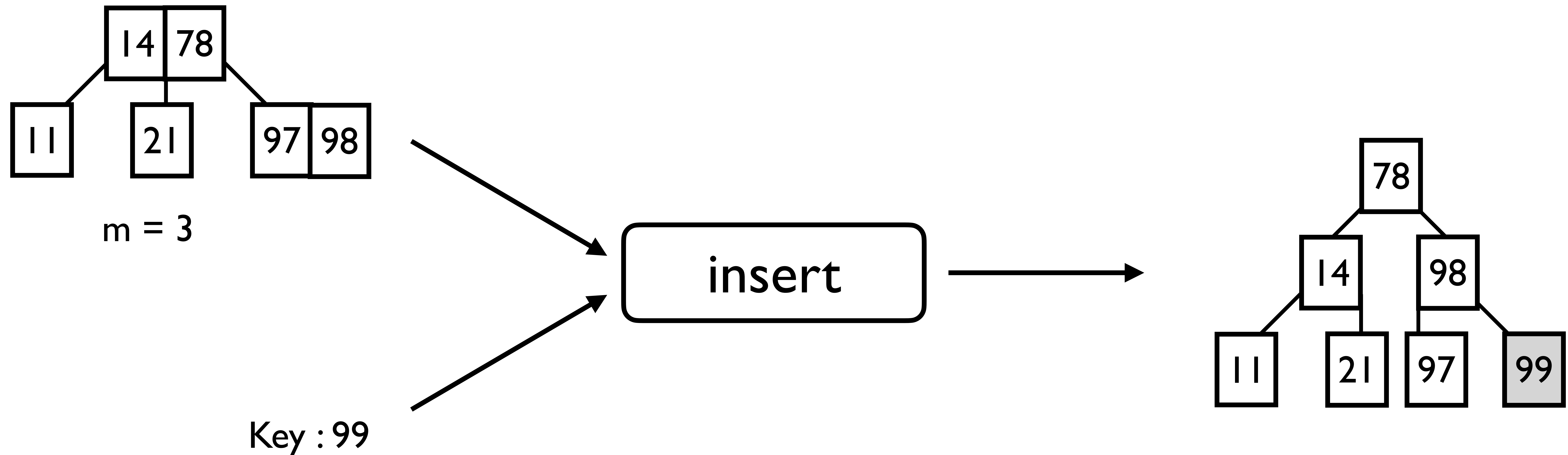



```
procedure traversal(root)
  i ← 0
  while i < root.num_keys-1 do
    if root.is_leaf = false then
      traversal(node.children[i])
    end if
    print(node.keys[i])
    i ← i + 1
  end while
  if root.is_leaf = false then
    traversal(node.children[i])
  end if
end procedure
```

```
Keys in B-tree:
1, 2, 3, 5, 6, 7, 8,
9, 15, 20, 25, 30, 90
```

insert

- insert(tree, key): B-tree의 특성을 유지하면서 주어진 key를 tree에 삽입함



insert

- B-tree에서 삽입(insert)은 항상 리프 노드에서 발생함

(1) B-tree에서 주어진 키를 삽입할 리프 노드를 찾아내어 키를 삽입함

(2) 노드가 넘치는(overflow) 경우 중간값(median)을 기준으로 좌우 key들을 분할(split)하고 가운데 키는 부모로 올림

(3) 부모노드에서도 넘침이 발생하면 재귀적으로 노드를 중간값을 기준으로 분할하고 중간값은 부모로 올림

insert

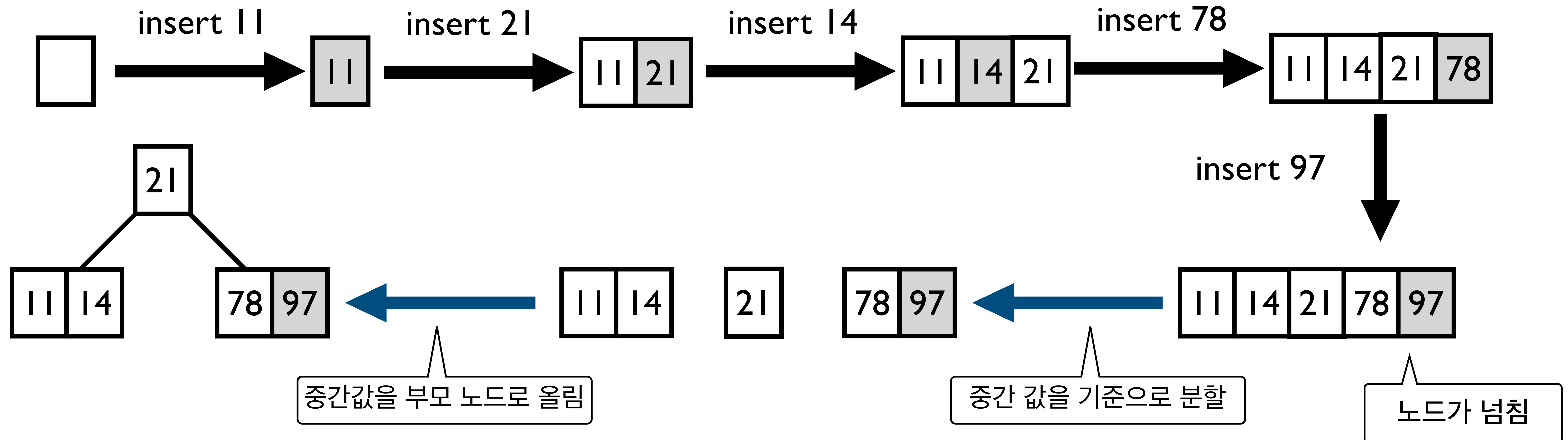
- B-tree에서 삽입(insert)은 항상 리프 노드에서 발생함

(1) B-tree에서 주어진 키를 삽입할 리프 노드를 찾아내어 키를 삽입함

(2) 노드가 넘치는(overflow) 경우 중간값(median)을 기준으로 좌우 key들을 분할(split)하고 가운데 키는 부모로 올림

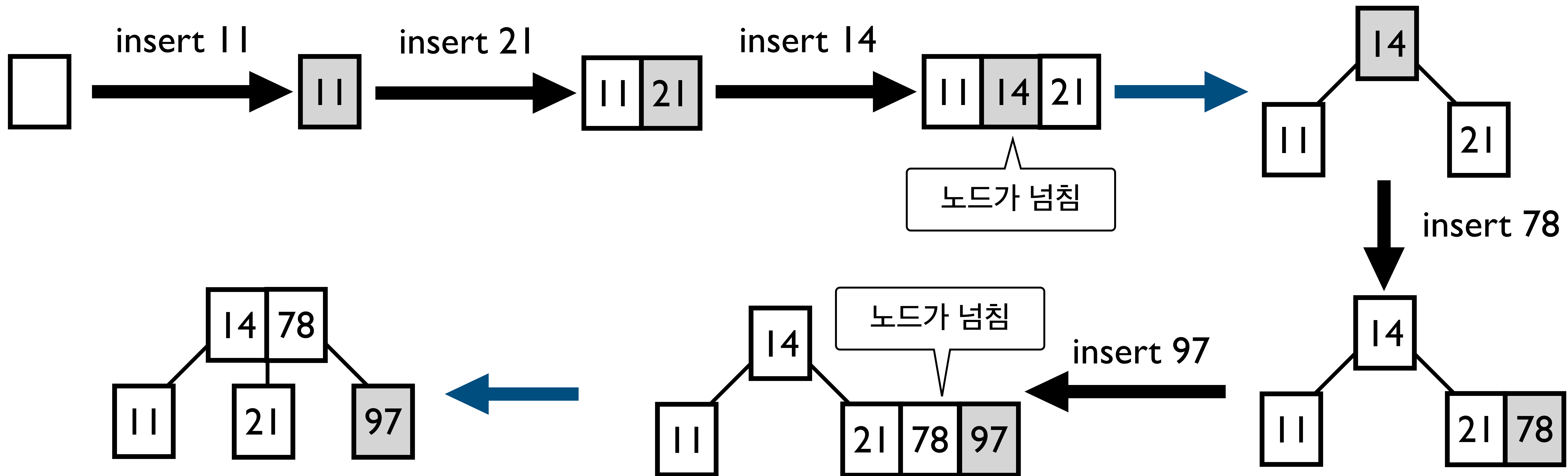
(3) 부모노드에서도 넘침이 발생하면 재귀적으로 노드를 중간값을 기준으로 분할하고 중간값은 부모로 올림

- Example: 차수가 5인 비어있는 B-tree에 11, 21, 14, 78, 97을 순차적으로 삽입하는 경우



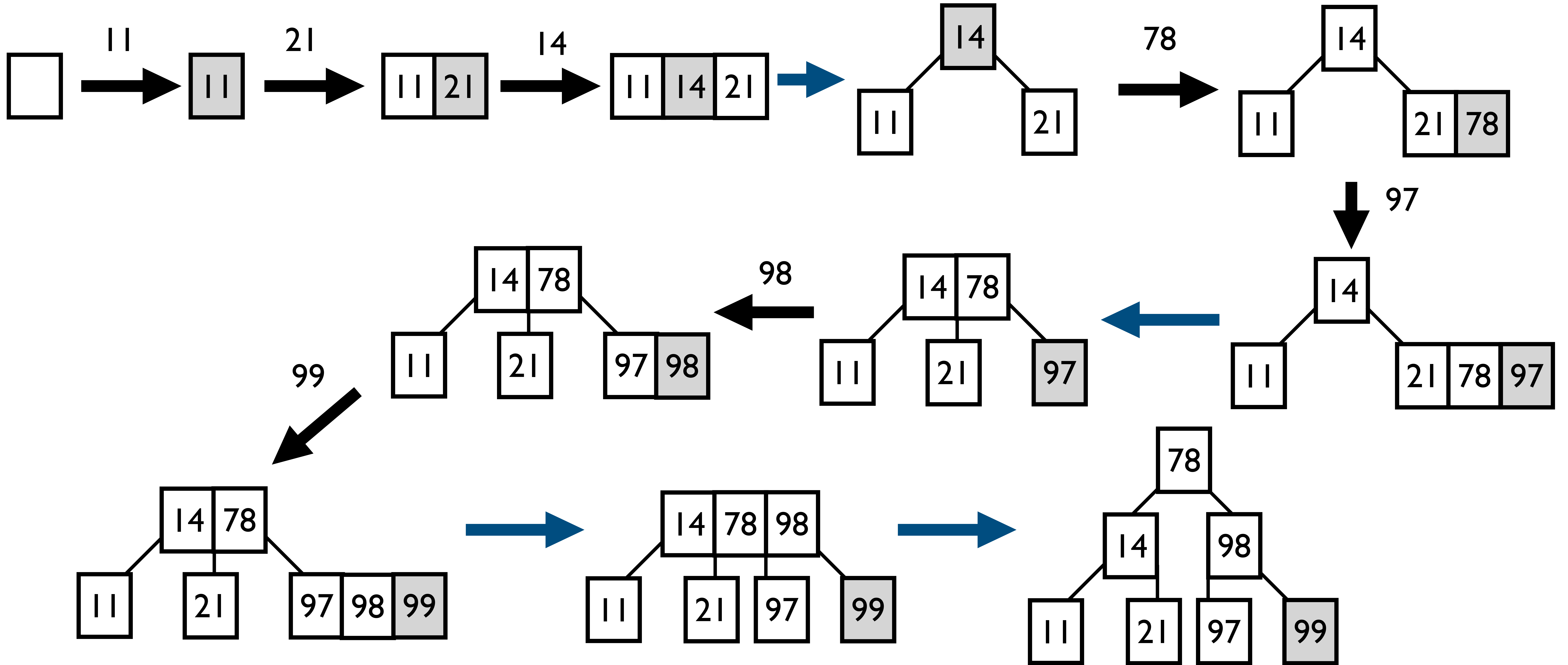
insert

- insert(tree, key): B-tree의 특성을 유지하면서 주어진 key를 tree에 삽입함
- 차수가 3인 비어있는 B-tree에 11, 21, 14, 78, 97을 순차적으로 삽입하는 예시



insert

- 차수가 3인 비어있는 B-tree에 11, 21, 14, 78, 97, 98, 99을 순차적으로 삽입하는 예시

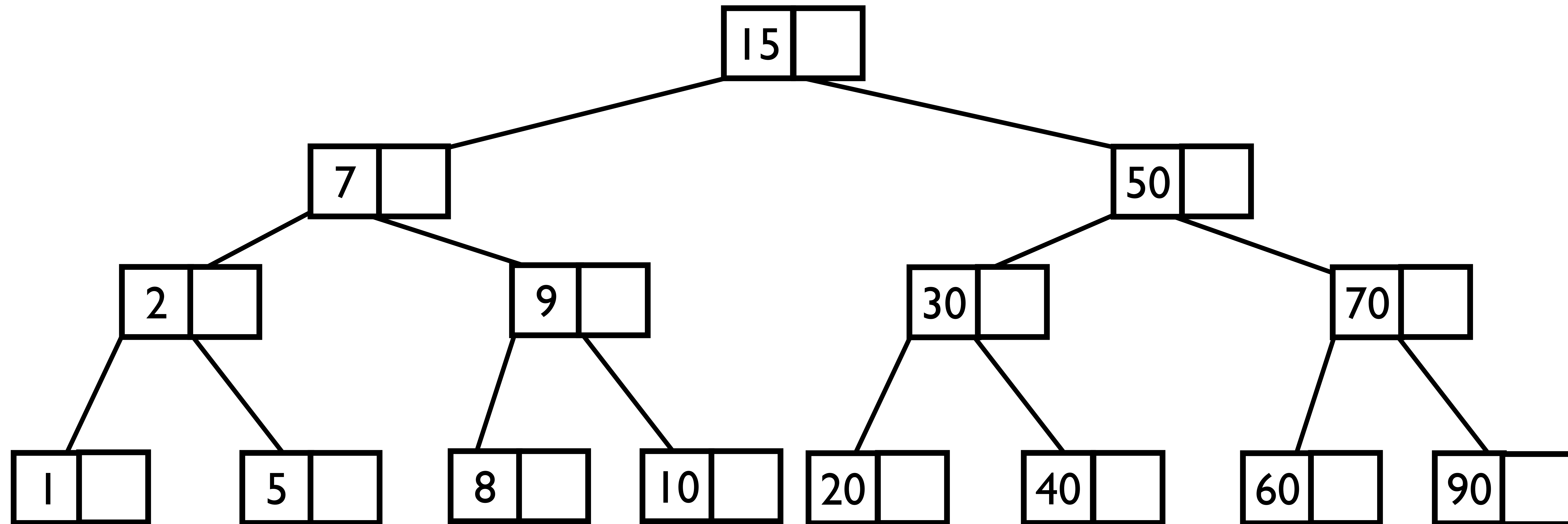


insert

- 차수가 3인 비어있는 B-tree에 1, 15, 2, 5, 30, 90, 20, 7, 9, 8, 10, 50, 70, 60, 40을 순차적으로 추가했을 때 최종 B-트리의 형태는?

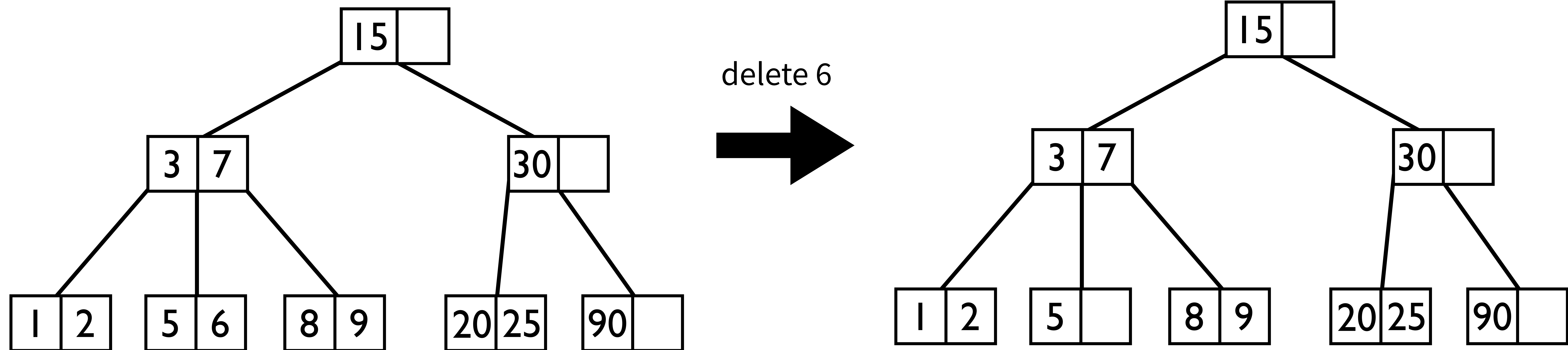
insert

- 차수가 3인 비어있는 B-tree에 1, 15, 2, 5, 30, 90, 20, 7, 9, 8, 10, 50, 70, 60, 40을 순차적으로 추가했을 때 최종 트리의 형태는?
- 모든 leaf 노드들은 같은 레벨에 있음
  - 검색의 시간 복잡도 = $O(\log n)$



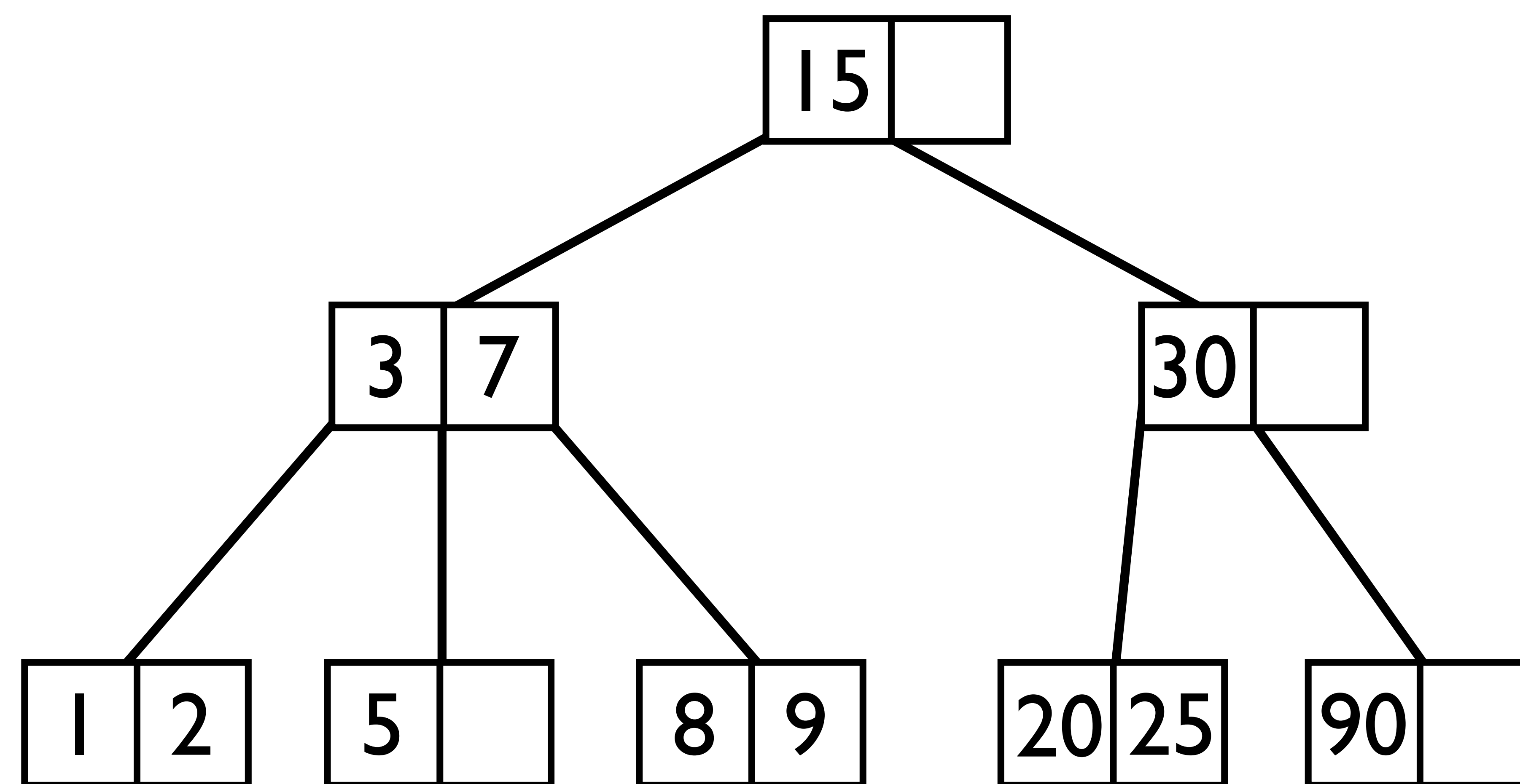
delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
- Example: m=3인 B-tree에서의 삭제

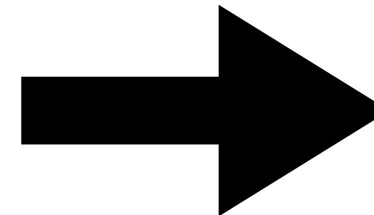


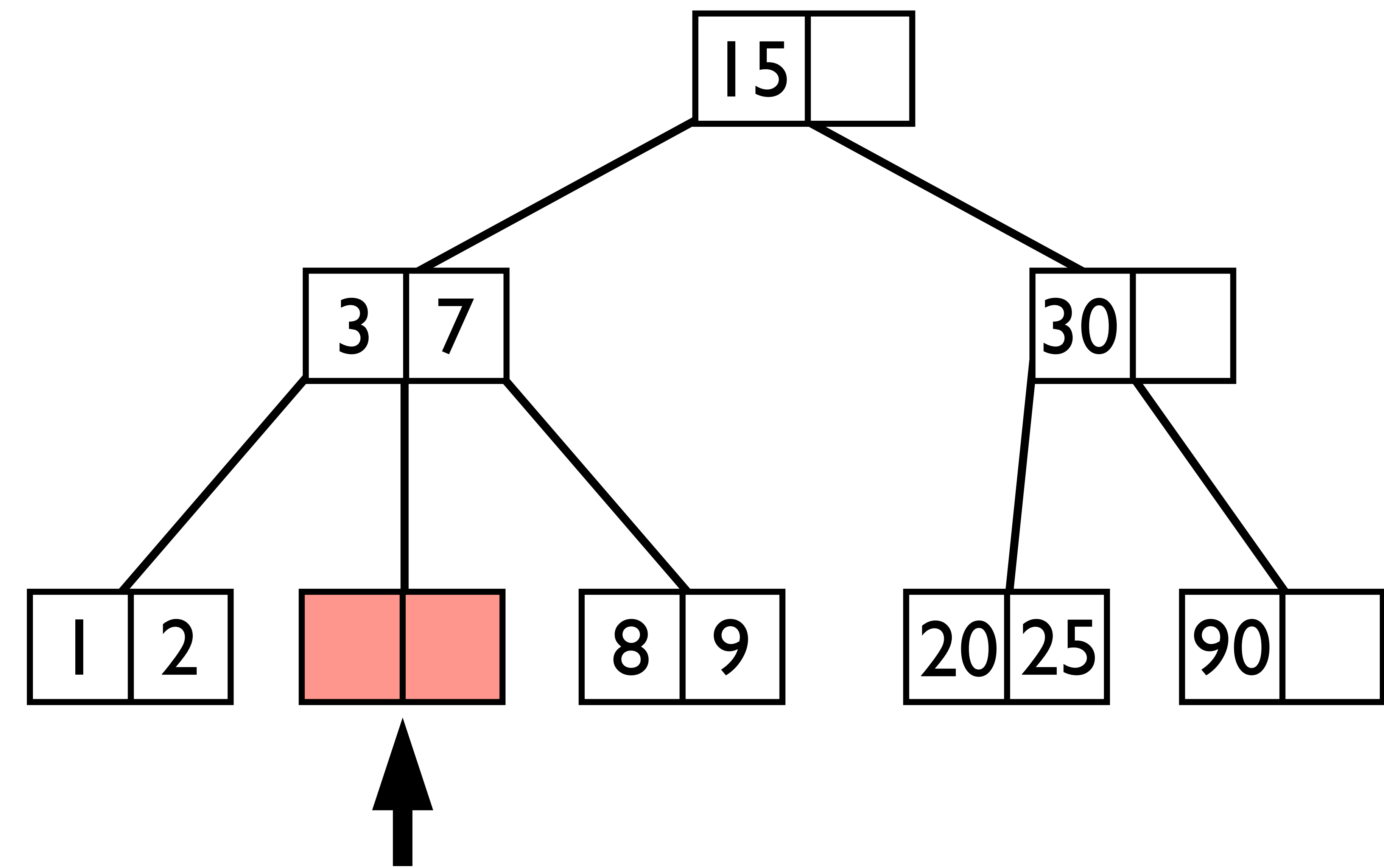
delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
- Example: m=3인 B-tree에서의 삭제



delete 5

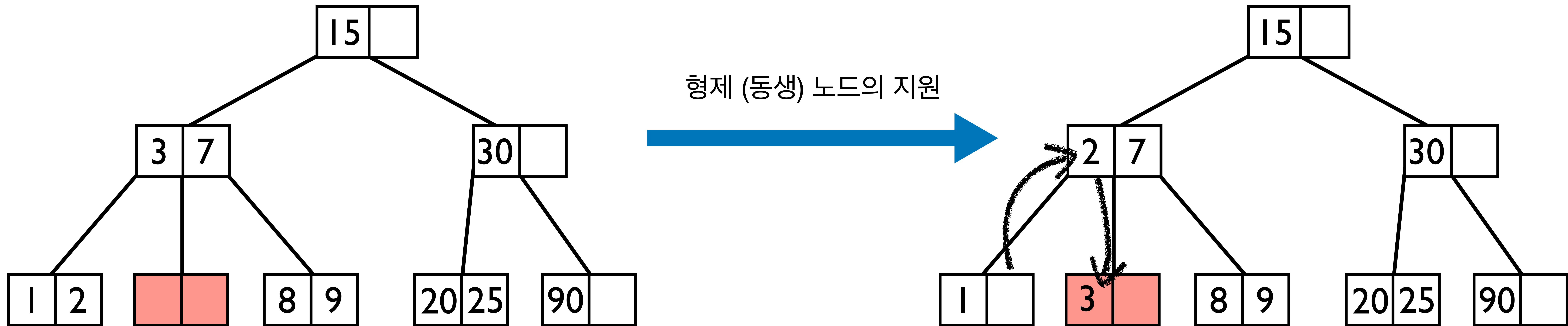




delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
  - 재조정이 필요한 경우

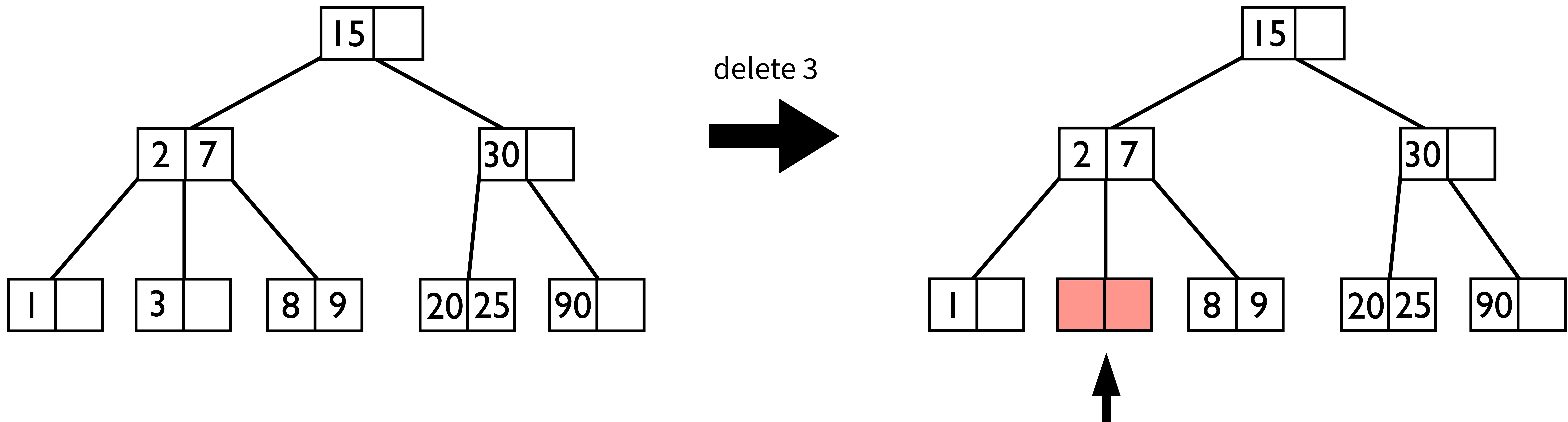
    (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.



delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
    - 재조정이 필요한 경우

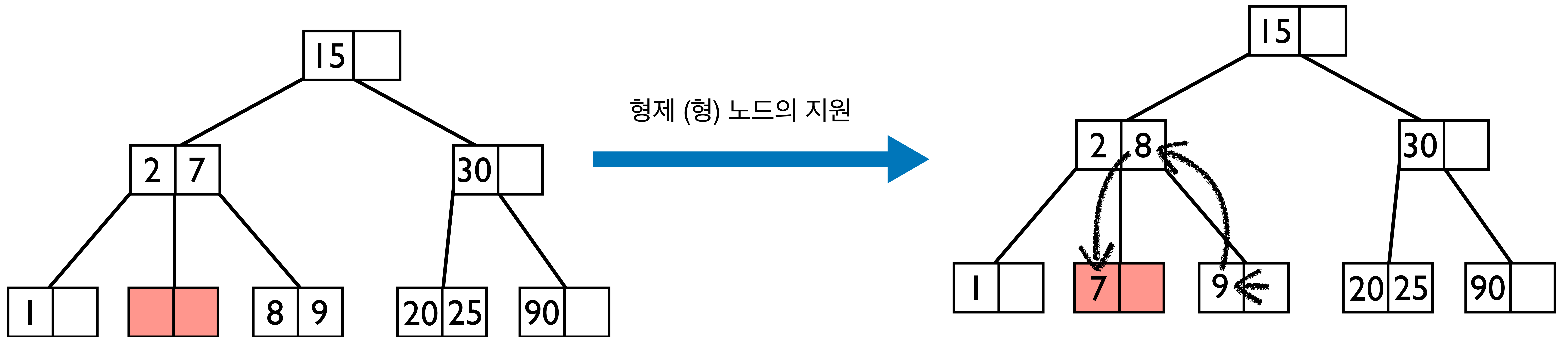
      (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.



delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
  - 재조정이 필요한 경우

    (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.

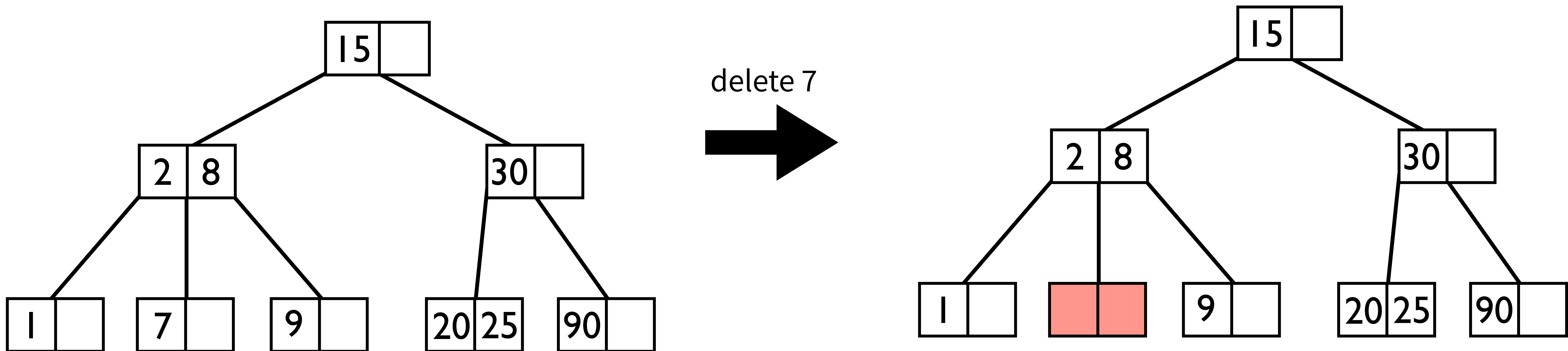


delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
  - 재조정이 필요한 경우

    (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.

    (2) 형제노드들에게 여유가 없을 경우 부모의 지원을 받고 형제와 합친다.

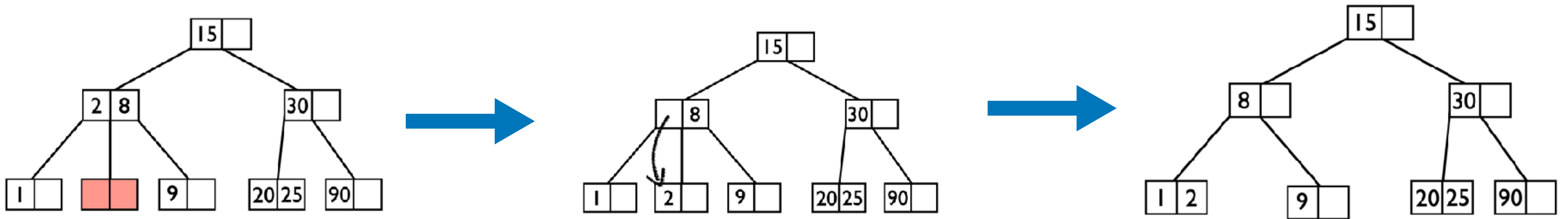


delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
  - 재조정이 필요한 경우

    (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.

    (2) 형제노드들에게 여유가 없을 경우 부모의 지원을 받고 형제와 합친다.

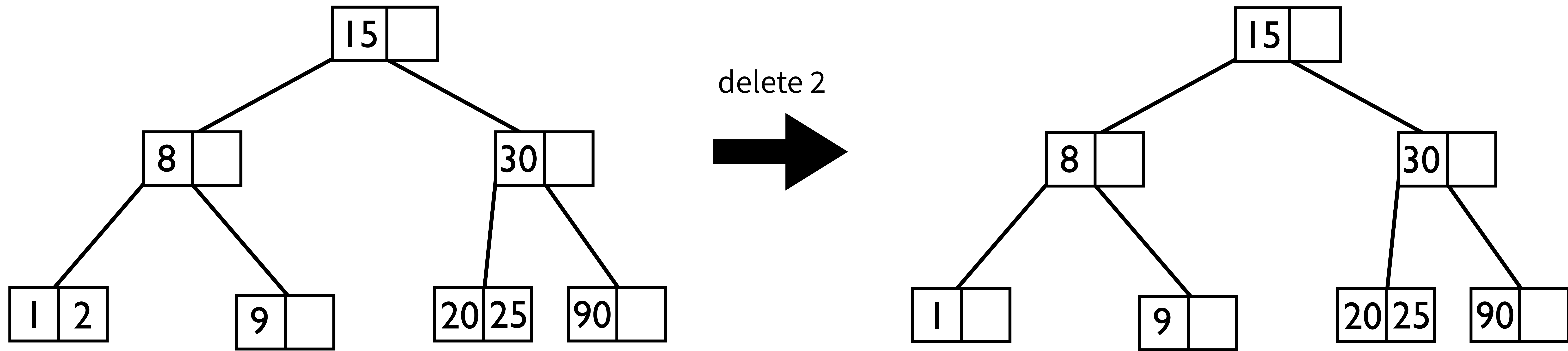


delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
  - 재조정이 필요한 경우

    (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.

    (2) 형제노드들에게 여유가 없을 경우 부모의 지원을 받고 형제와 합친다.

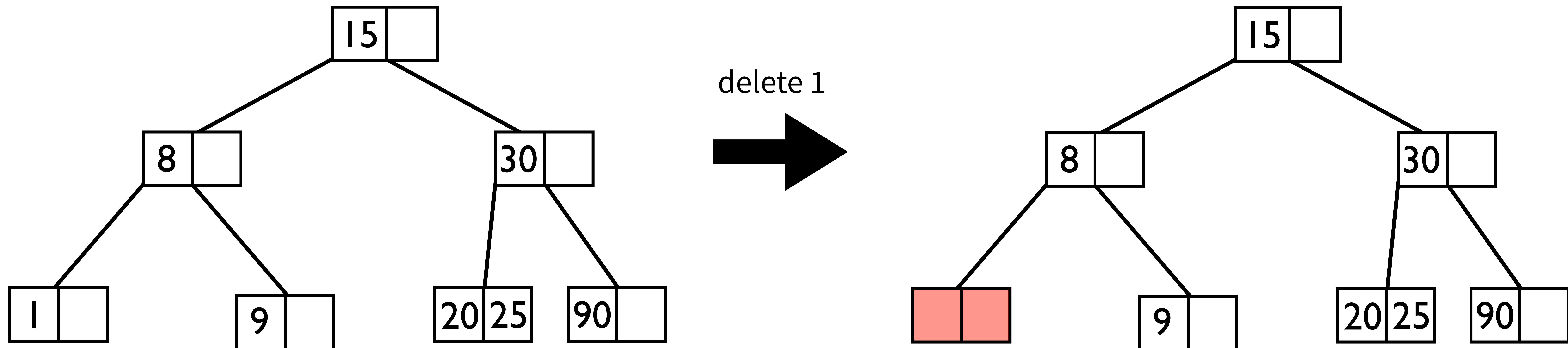


delete

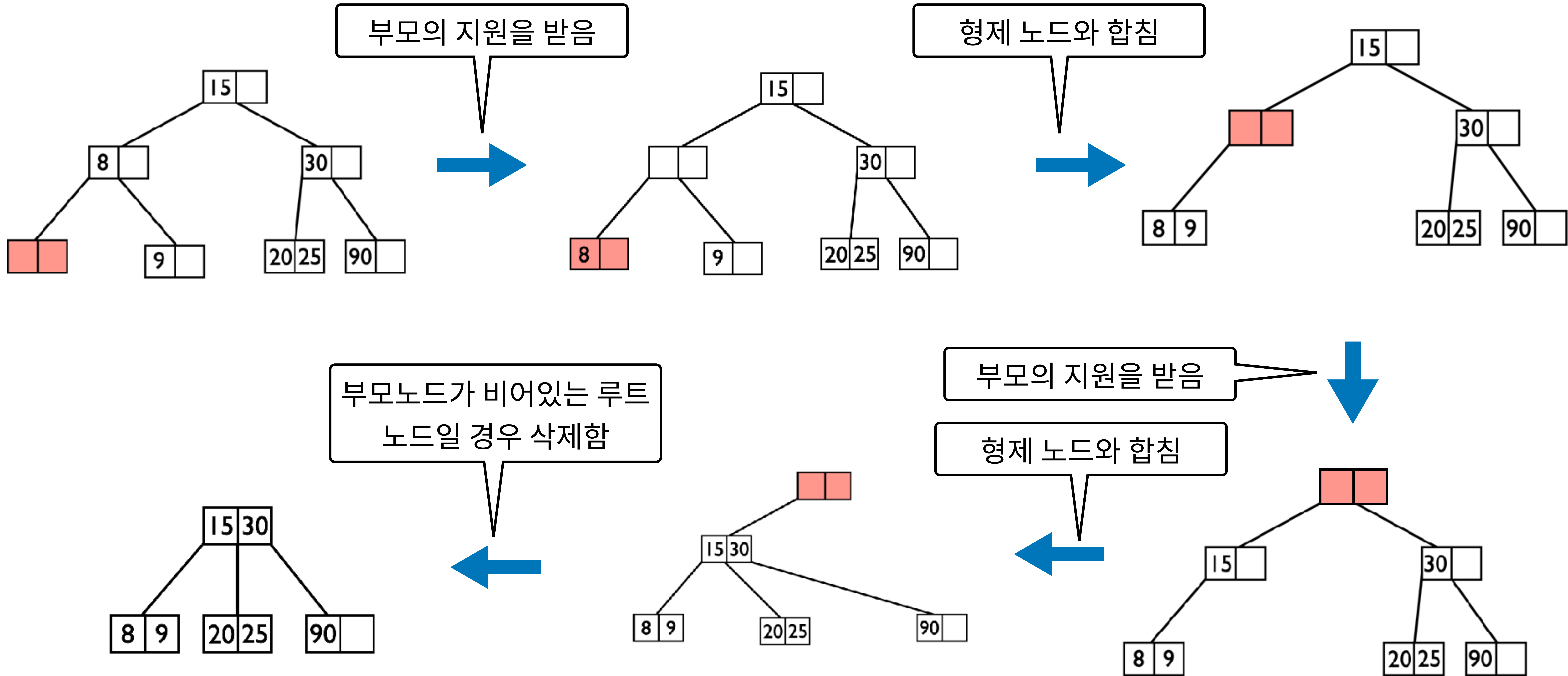
- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 재조정함
  - 재조정이 필요한 경우

    (1) key의 개수가 여유 있는 형제 노드의 지원을 받는다.

    (2) 형제노드들에게 여유가 없을 경우 부모의 지원을 받고 형제와 합친다.



delete



delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
  - 리프 노드가 아닌 노드에 있는 key를 삭제해야할 경우 리프 노드에 있는 키와 위치를 바꾼 후 삭제함
- 키를 삭제 후 노드의 key의 개수가 최소 키의 개수(⌈m/2⌉-1개)보다 적어졌다면 B-tree를 아래와 같이 재조정함

(1) key의 개수가 여유 있는 형제 노드가 있을 경우 지원을 받는다.

(1.1) 동생(왼쪽 형제)이 여유가 있는 경우

- 동생의 가장 큰 key를 부모 노드의 동생과 나 사이의 키로 둔다

- 부모 노드에 있던 키는 delete가 발생한 노드 가장 왼쪽에 둔다.

(1.2) 동생이 여유가 없고 형(오른쪽 형제)이 여유가 있는 경우

- 형의 가장 작은 key를 부모 노드의 나와 형 사이의 키로 둔다

- 부모 노드에 있던 키는 delete가 발생한 노드 가장 오른쪽에 둔다.

delete

(2) 형제 노드들이 여유가 없을 경우 부모 노드의 지원을 받고 형제 노드와 합친다.

(2.1) 동생 노드가 있으면 동생과 나 사이의 key를 부모로부터 받는다.

- 받은 key와 나의 키들을 동생에 합친 후 노드를 삭제한다.

(2.1) 동생 노드가 없으면 나와 형 노드 사이의 key를 부모로부터 받는다.

- 받은 key와 나의 키들을 형 노드에 합친 후 노드를 삭제한다.

(3) 부모 노드가 지원한 후 문제가 생길 경우 아래와 같이 대응한다.

(3.1) 부모 노드가 루트 노드가 아닐 경우 그 위치에서 (1)부터 재조정을 시작한다.

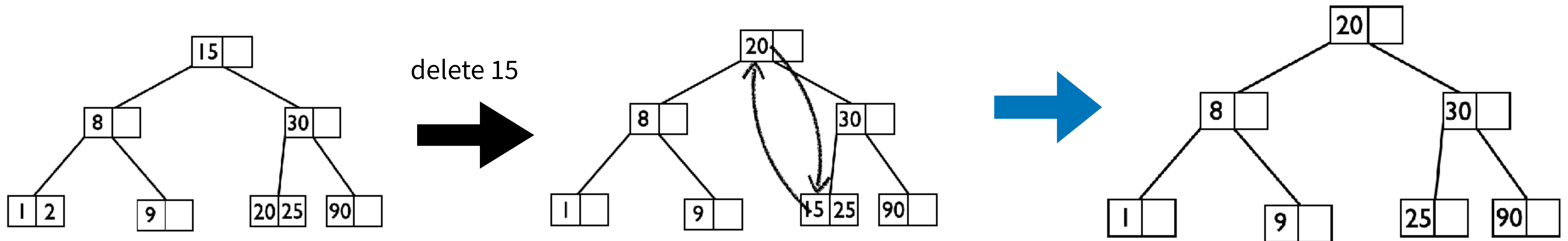
(3.2) 부모 노드가 비어있는 루트 노드일 경우 그 노드를 삭제하고 합쳐진 노드가 루트 노드가 된다.

delete

- B-tree에서의 삭제는 항상 리프 노드에서 발생함
- 리프 노드가 아닌 노드에 있는 key를 삭제해야할 경우 리프 노드에 있는 키와 위치를 바꾼 후 삭제함

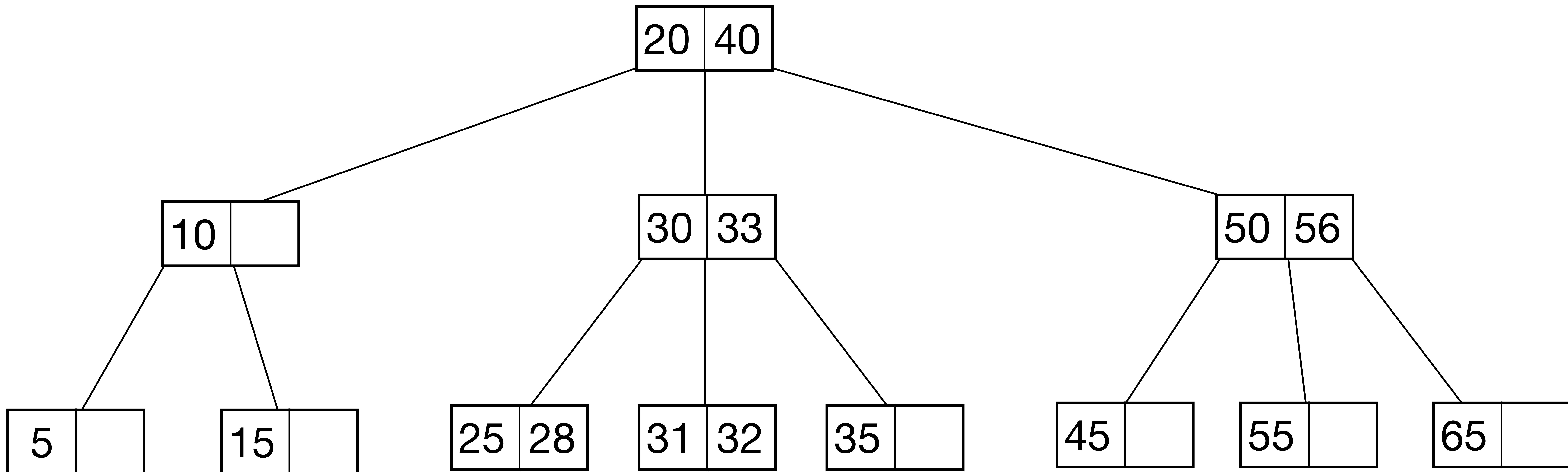
  - (case 1) 리프에 있는 키들중 나보다 큰 키들 중 가장 작은 키값과 교체 또는

  - (case 2) 리프에 있는 키들중 나보다 작은 키들 중 가장 큰 키값과 교체



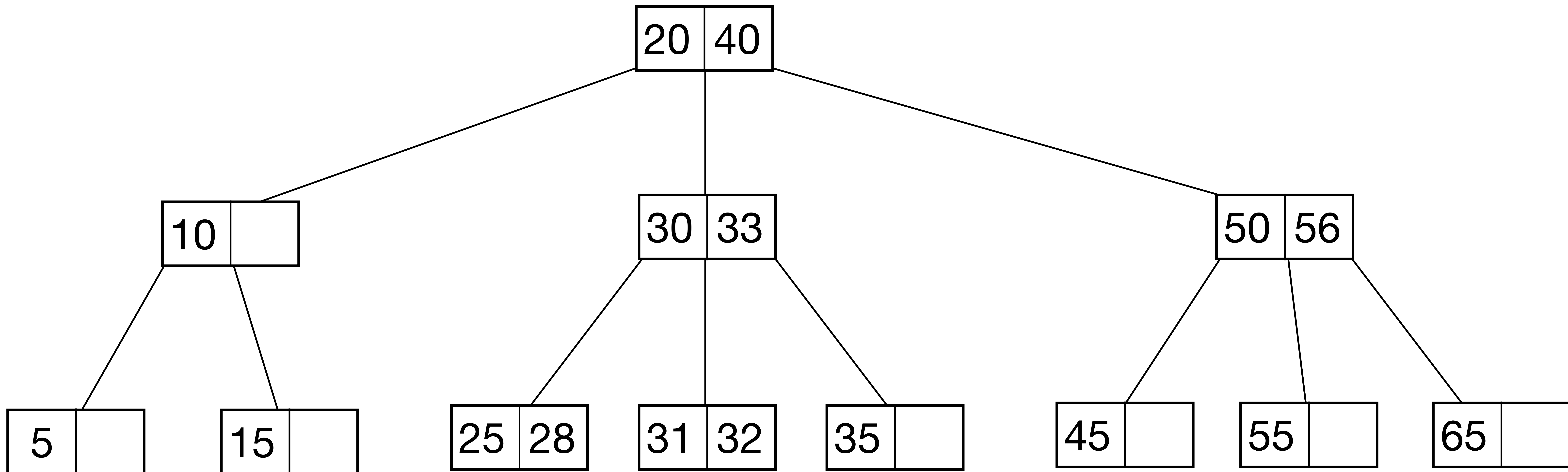
delete

- Example 1: 다음의 차수가 3인 B-tree에서 32, 31,30 순으로 삭제가 일어날 때 각 단계에서 트리의 형태는?



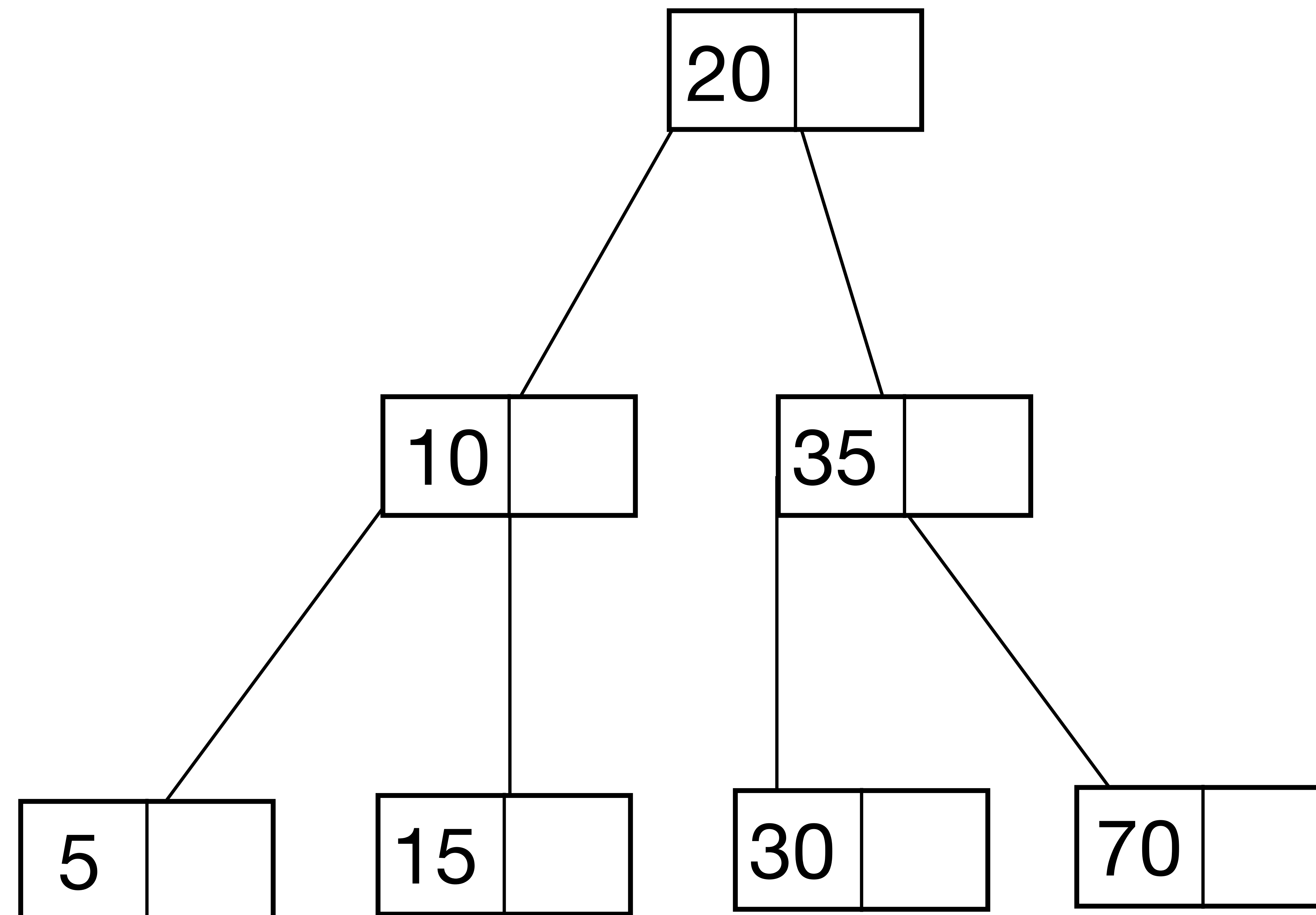
delete

- Example 1: 다음의 차수가 3인 B-tree에서 33, 30 순으로 삭제가 일어날 때 각 단계에서 트리의 형태는?



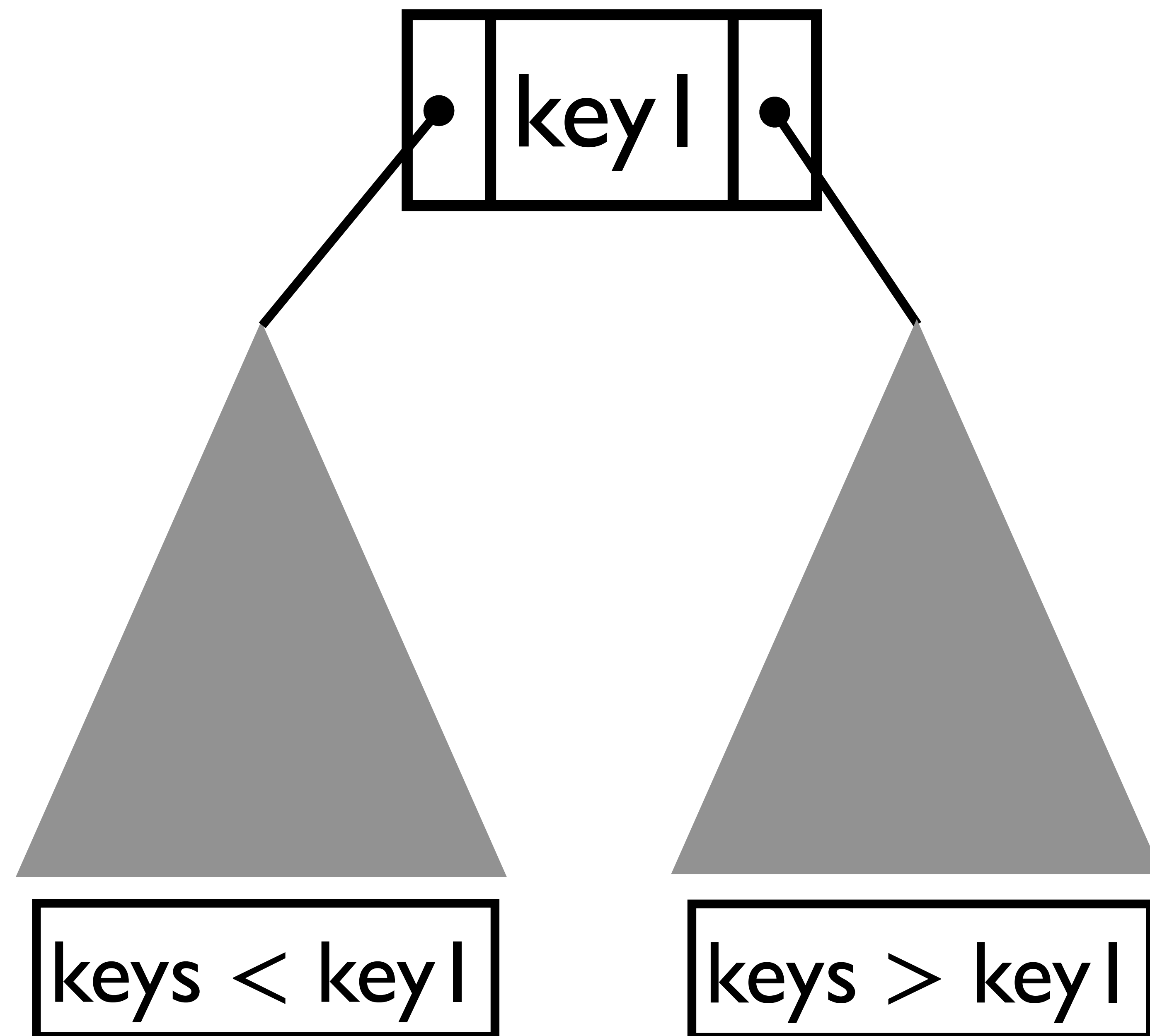
delete

- Example 1: 다음의 차수가 3인 B-tree에서 10이 삭제될 때 트리의 변화는?



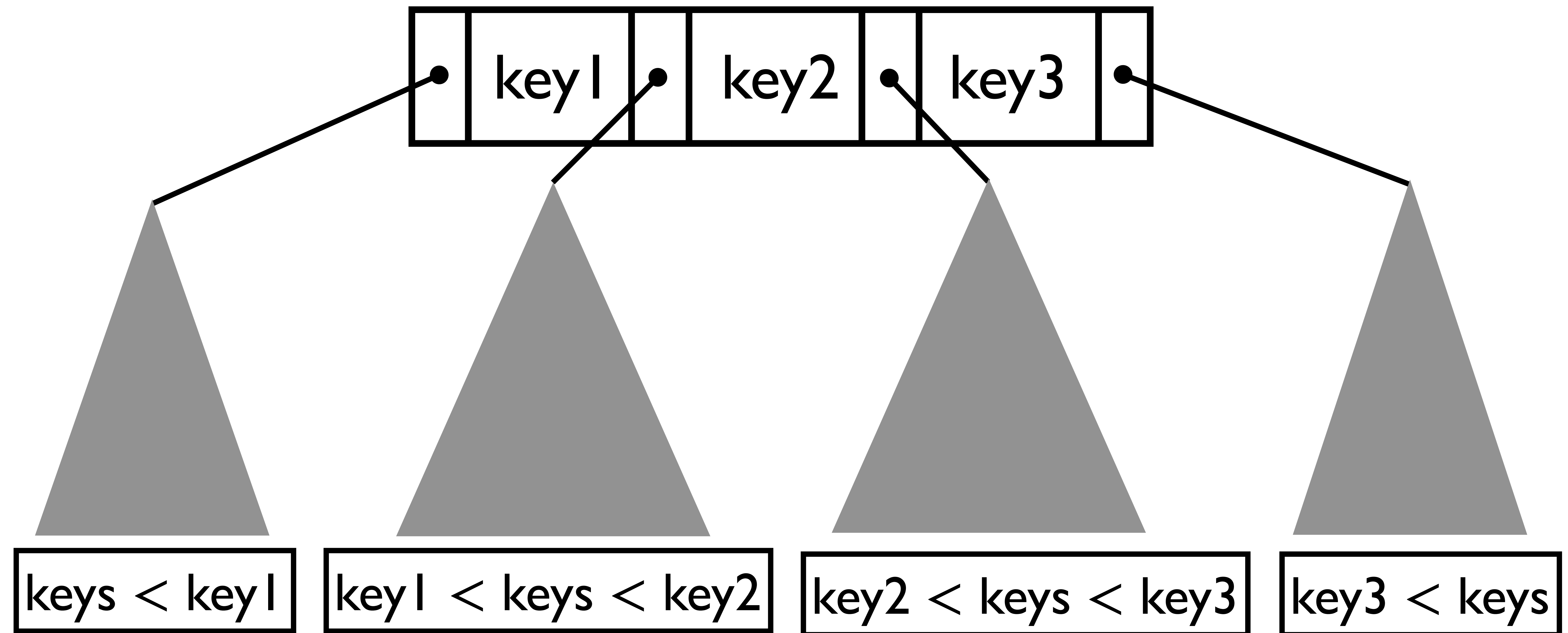
마무리

- 다원 탐색 트리(multi way search tree)는 이진 탐색 트리(binary search tree)의 일반화임
- 다원 탐색 트리의 장점:
  - 한 노드에 여러가지 키를 저장할 수 있어 트리의 높이를 낮출 수 있음



이진 탐색 트리

vs



4-방향 탐색 트리 (4-way search tree)