제5장 탐색 트리 (추가)

2-3트리, Red-Black트리, B트리

5.3 2-3트리

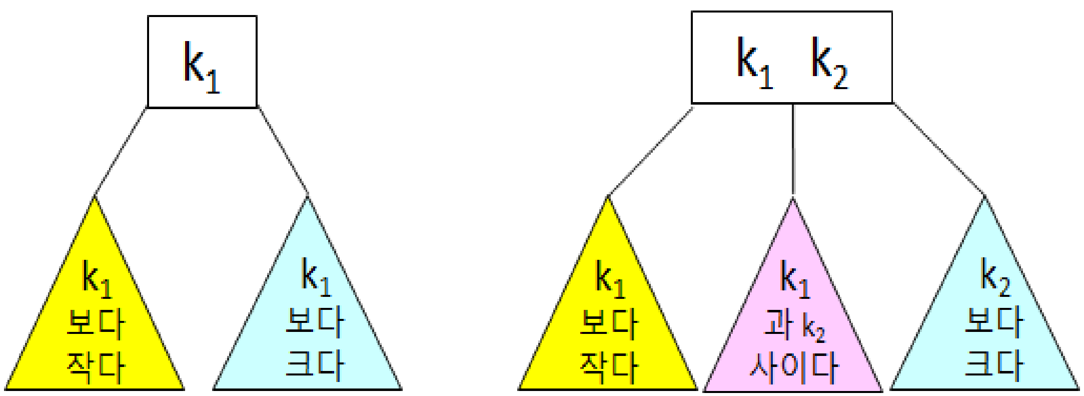
- **2-3트리는 내부노드의 차수가 2 또는 3인 균형 탐색트리**
  - 차수가 2인 노드 = 2-노드, 차수가 3인 노드 = 3-노드
  - 2-노드: 한 개의 키를 가지며, 3-노드는 두 개의 키를 가짐

  - 2-3트리는 루트로부터 각 이파리노드까지 경로의 길이가 같고, 모든 leaf node들이 동일한 층에 있는 완전한 균형트리
  - 2-3트리가 2-노드들만으로 구성되면 Full Binary Search Tree와 동일한 형태를 가짐

2-3트리

[핵심 아이디어] 2-3트리는 이파리노드들이 동일한 층에 있어야 하므로 트리가 위로 자라나거나 낮아진다.

- 2-노드의 키가 k_1이라면, 노드의 왼쪽 서브트리에는 k_1보다 작은 키들이 있고, 오른쪽 서브트리에는k_1보다 큰 키들이 있다.
- k_1과 k_2를 가진 3-노드는 3개의 서브트리를 가지는데, 왼쪽 서브트리에는 k_1보다 작은, 중간 서브트리에는 k_1보다 크고 k_2보다 작은, 오른쪽 서브트리에는 k_2보다 큰 키들이 있다.



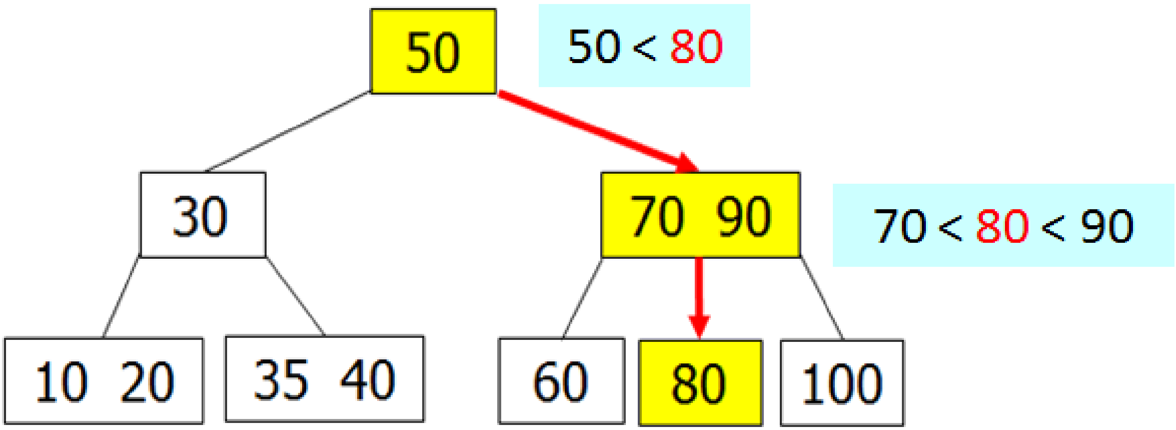
2-3트리

- **2-3트리에서도 이진탐색트리에서의 중위순회와 유사한 방법으로 중위순회 수행**
  - 2-노드는 이진트리의 중위순회 방문과 동일
  - k_1과 k_2를 가진 3-노드에서는 먼저 노드의 왼쪽 서브트리에 있는 모든 노드들을 방문한 후에 k_1을 방문하고, 이후에 중간 서브트리에 있는 모든 노드들을 방문
  - 다음으로 k_2를 방문하고 마지막으로 오른쪽 서브트리에 있는 모든 노드들을 방문한다.
- **따라서 2-3트리에서 중위순회를 수행하면 키들이 정렬된 결과를 얻음**

5.3.1 탐색 연산

- 루트노드에서 시작하여 방문한 노드의 키들과 탐색하고자 하는 키를 비교하며 다음 레벨의 노드를 탐색

  - [예제] 80 탐색



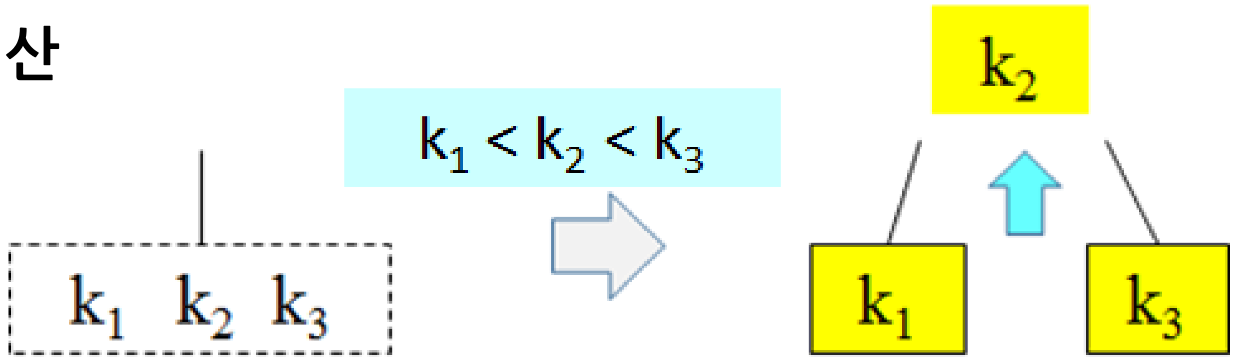
5.3.2 삽입 연산

- **2-3트리에서의의 삽입을 수행하려면 먼저 탐색과 동일한 과정을 거쳐 새로운 키가 삽입되어야 할 이파리노드를 찾아야**

- **Leaf Node에서의 삽입 방식**
  - 이파리노드가 2-노드이면 그 노드에 새 키를 삽입
  - 이파리노드가 3-노드이면 새로운 키를 저장할 수 없으므로(overflow),
    (1) 이 노드에 있는 기존의 2 개의 키와 새로운 키 중에서
        중간값이 되는 키를 부모노드로 올려 보내고,
    (2) 남은 두 개의 키를 각각 별도의 노드에 저장
  - 이 과정을 분리(Split) 연산이라고 함.
  - 부모 노드에도 이미 2개의 키가 있다면 분리 연산을 부모 노드에서도 수행
    - 이 작업이 반복되면 2-3 트리의 높이가 하나 높아지게 됨

5.3.2 삽입 연산

- **분리 연산**



(a) 삽입 전

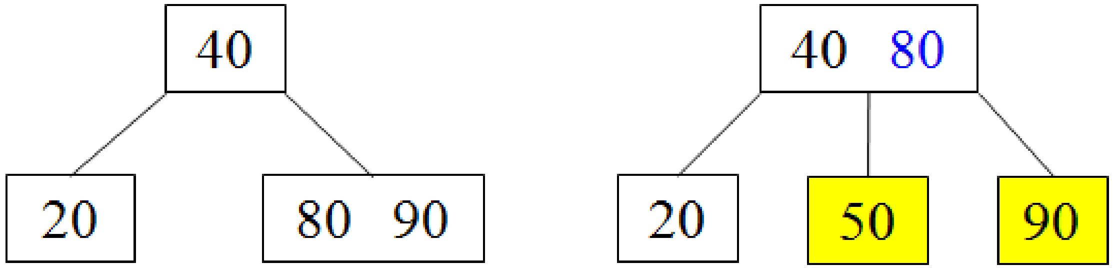
(b) 분리 저장 후 중간값을 위로

- (a) overflow가 발생된 노드에 3개의 키가 있을 때, k1 < k2 < k3이라면,
  (b) k1과 k3을 각각 2-노드에(하나는 기존 노드에 다른 하나는 생성하여)
    저장하고, 중간값인 k2를 부모노드로 올려보내
    k1과 k3의 분기점 역할을 하도록

5.3.2 삽입 연산

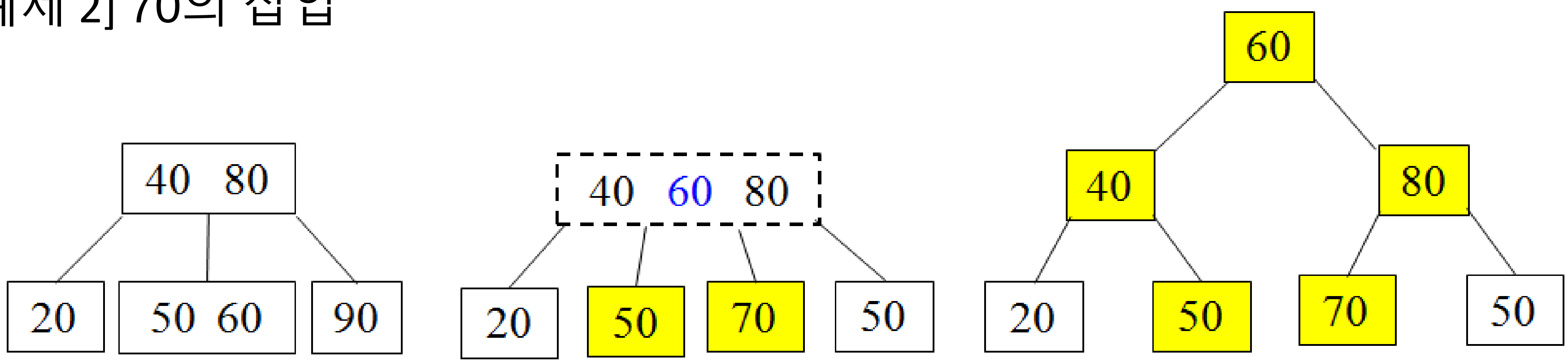
- 부모노드로 올려 보내진 키는
  - 이파리노드에서와 마찬가지로 자리가 있으면, 즉 부모노드가 2-노드이면, 부모노드에 저장하고 삽입 연산을 종료
  - 부모노드가 3-노드이면 분리 연산을 다시 수행
  - 위 과정은 루트까지 올라가면서 반복될 수 있다.
  - 루트에서 노드 분리가 일어나면 2-3트리의 높이가 1 증가

[예제 1] 50의 삽입



(a) 삽입 전　　(b) 삽입 후

[예제 2] 70의 삽입



(a) 삽입 전　　(b) 분리 연산 수행　　(c) 분리 연산 수행

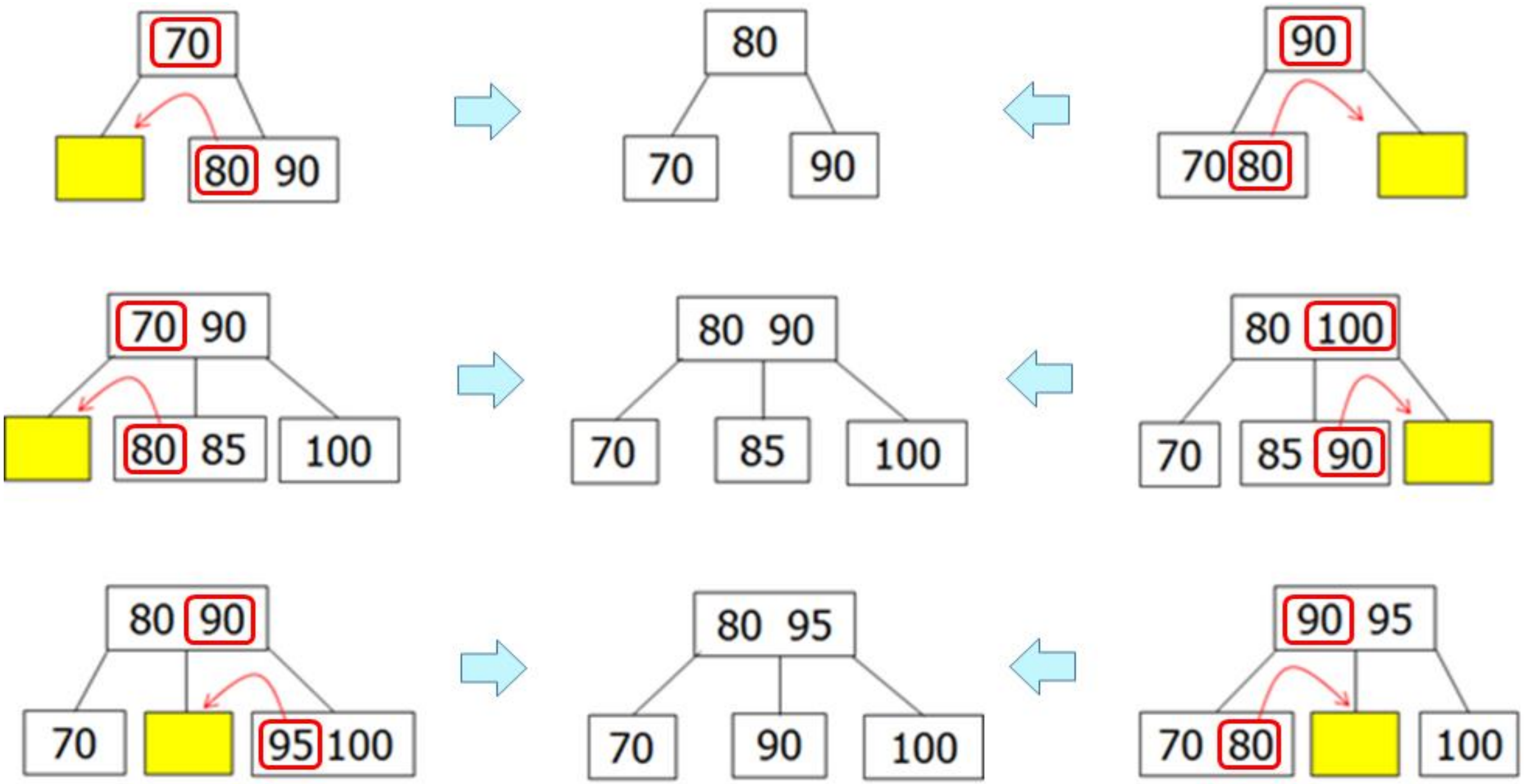
5.3.3 삭제 연산

- 2-3트리에서의 삭제는 항상 이파리노드에서 이루어짐
- 만약 삭제할 키가 있는 노드가 이파리노드가 아닌 경우, 이진탐색트리의 삭제와 유사하게 중위 선행자 또는 중위 후속자와 교환한 후에 이파리노드에서 실질적인 삭제를 수행
- 2-3트리의 삭제: 이동(Transfer) 연산과 통합(Fusion) 연산 사용

이동 연산

- **이동 연산이란 키가 삭제되어 노드가 empty가 되었을 때, 이 노드의 형제노드와 부모노드의 도움을 받아 1 개의 키를 empty 노드로 이동시키는 연산**
  - 형제노드는 반드시 3-노드이어야 함
  - 3-노드가 empty 노드의 왼쪽 형제노드라면,
    2 개의 키 중에서 큰 키를 부모노드로 올려 보내고
    부모노드의 키를 empty 노드로 내려 보냄
  - 형제노드가 empty 노드의 오른쪽 형제노드인 경우,
    2 개의 키 중에서 작은 키를 부모노드로 올려 보내고
    부모노드의 키를 empty 노드로 내려 보냄

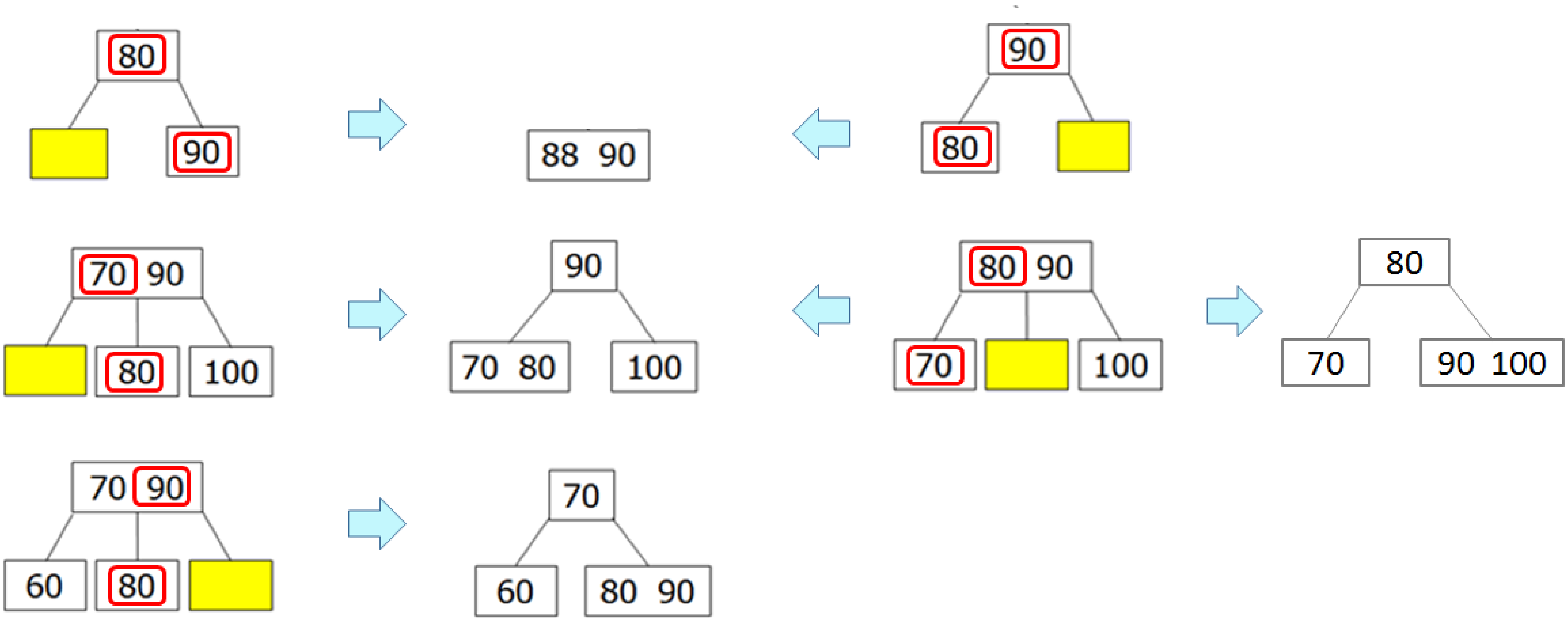
부모노드가 2-노드, 3-노드인 경우 이동 연산



통합 연산

- **노드가 empty일때 이동 연산이 불가능한 경우**
  empty 노드와 그의 형제노드를 1개의 노드로 통합하고,
  empty 노드와 그의 형제노드의 분기점 역할을 하던 부모노드의 키를
  통합된 노드로 끌어내려 저장하는 연산
  - 통합 연산과 분리 연산은 상호 역(Reverse) 연산 관계

부모노드가 2-노드, 3-노드인 경우 통합 연산



2-3트리의 삭제 연산 알고리즘

[1] 삭제할 키 k가 있는 노드x를 탐색

[2] if (x가 이파리노드이면), k를 노드 x에서 삭제.
x를 삭제 후 empty가 아니면 알고리즘 종료.
만약 x가 empty인 경우, x의 형제노드들 중에 3-노드가 있으면 이동 연산을 수행하고, 그렇지 않으면 통합 연산 수행

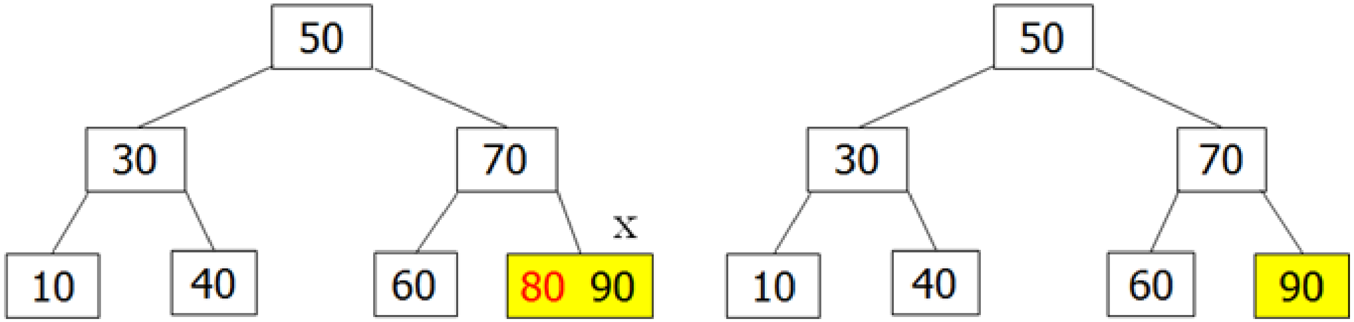
[3] if (x 가 이파리노드가 아니면), k의 중위 선행자가 있는 노드 y와 중위 후속자가 있는 노드 z 탐색.

[3-1] if (y 또는 z에서 이동 연산이 가능하면), 이동 연산 가능한 키를 k와 서로 교환하고 이동 연산을 수행하며, 동시에 k를 삭제한 후에 알고리즘을 종료

[3-2] if (y와 z 둘 다 이동 연산이 불가능하면), y나 z 중에서 임의로 하나를 선택한다. 그리고 선택한 노드의 키를 k와 서로 교환한 후 k를 삭제하고, 통합 연산 수행

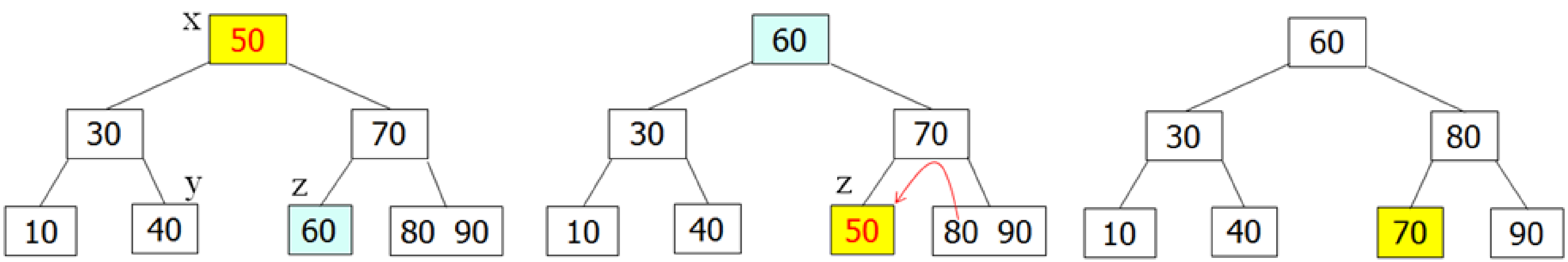
- 통합 연산 수행 후 루트 방향으로 연속적인 통합 연산이 수행될 수도

[예제 1] 80을 삭제

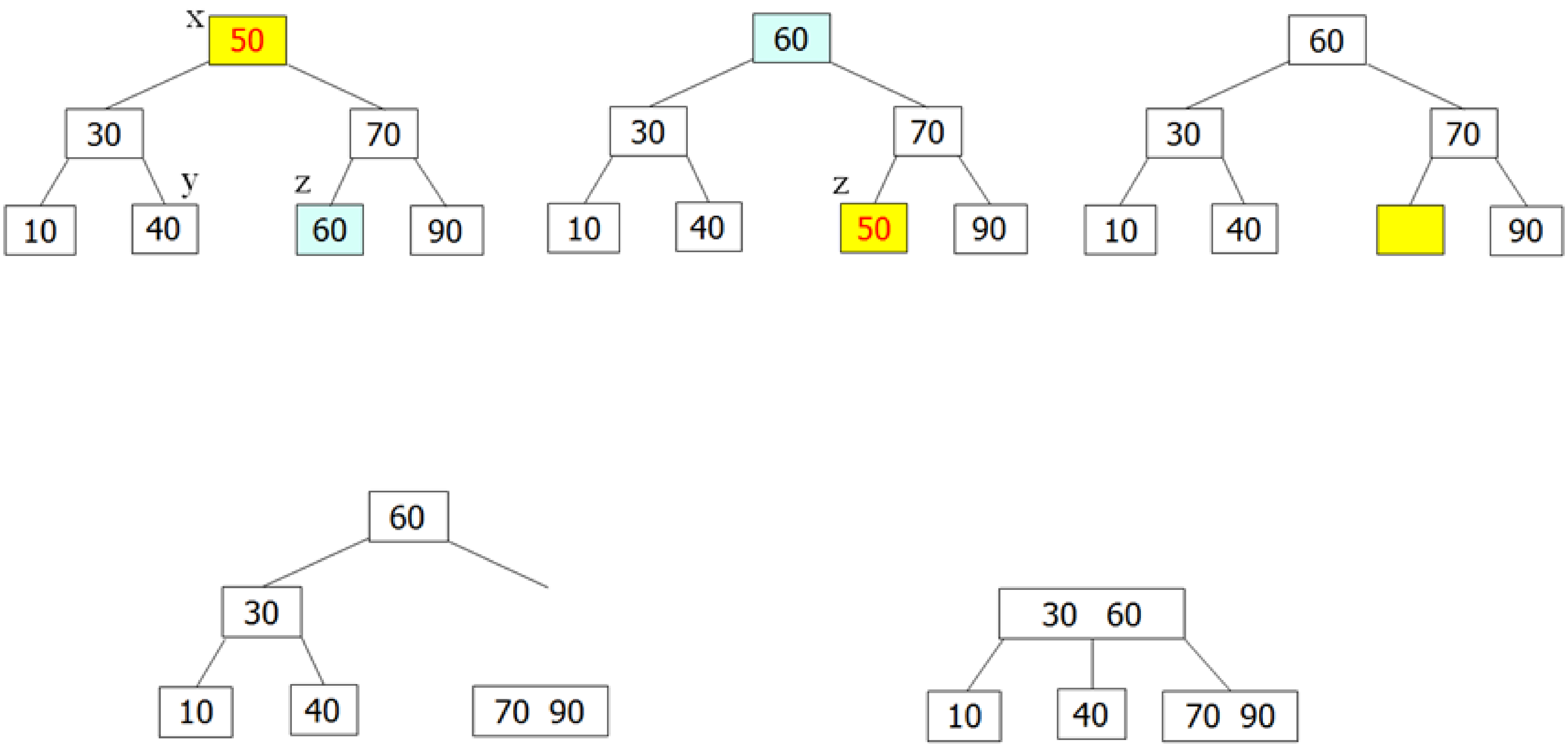


(a) 80이 있는 노드 탐색 (b) 80을 삭제

[예제 2] 50을 삭제



[예제 3] 50을 삭제


x
50
30
70
10
40
y
z
60
90
60
30
70
10
40
z
50
90
60
30
70
10
40
90
60
30
10
40
70 90
30 60
10
40
70 90

수행시간

- **2-3트리의 탐색, 삽입, 삭제 연산 시간은 각각 트리 높이에 비례**
  - 각 연산은 루트노드부터 이파리노드까지 탐색해야 하고, 삽입이나 삭제는 분리나 통합 연산을 수행하며 다시 루트노드까지 올라가는 경우도 있기 때문
  - 단, 개별적인 분리 연산이나 통합 연산은 각각 트리의 지역적인 부분에서만 수행되므로 O(1) 시간만 소요
- **2-3 트리가 가장 높은 경우는 모든 노드가 2-노드인 경우이고, 이 때의 트리 높이 = $\log_2(N+1)$**
- **트리의 모든 노드가 3-노드이면 트리의 높이가 최소이며, 높이는 $\log_3 N \approx 0.63 \log_2 N$이다.**
- **따라서 2-3트리의 탐색, 삽입, 삭제 연산의 수행시간은 각각 O(logN)**

수행시간

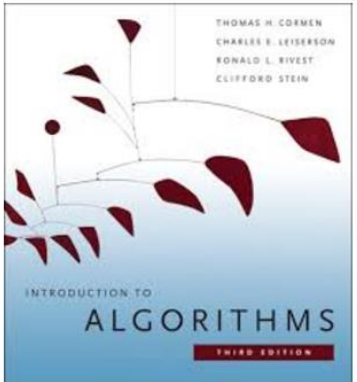
- **2-3트리는 이진탐색트리에 비해 매우 우수한 성능을 보이나, 2-3트리를 실제로 구현하기에 다소 어려움이 따름**
  - 노드를 2 개의 타입으로 정의해야 하고, 분리 및 통합 연산에서의 다양한 경우를 고려해야 하기 때문
  - 3-노드에서는 키를 2회 비교하는 것도 고려해야
  - 2-3 트리는 5.4절에서 설명할 좌편향(Left-Leaning) 레드블랙트리의 기본 형태를 제공

2-3-4 트리

- **2-3트리를 확장한 2-3-4트리는 노드가 자식노드를 4개까지 가질 수 있는 완전균형트리**
  - 2-3-4트리의 장점: 2-3트리보다 높이가 낮아 그 만큼 빠른 탐색, 삽입, 삭제 연산이 수행이 가능
  - 2-3-4트리에서는 삽입 연산을 루트부터 이파리노드로 내려가며 4-노드를 만날 때마다 미리 분리 연산을 수행할 수 있기 때문에 다시 이파리노드부터 위로 올라가며 분리 연산을 수행할 필요가 없고, 따라서 보다 효율적인 삽입 연산이 가능
  - 삭제 연산도 삽입 연산과 유사하게 루트로부터 이파리노드 방향으로 내려가며 2-노드를 만날 때마다 미리 통합 연산을 수행하므로 키를 삭제한 후 다시 루트 방향으로 올라가며 통합 연산을 수행할 필요 없음
  - 그러나 이러한 삽입과 삭제 연산도 이론적으로는 2-3트리의 수행시간과 동일한 $O(\log_2 N)$

5.4 레드블랙트리

- **노드에 색을 부여하여 트리의 균형을 유지**
- **탐색, 삽입, 삭제 연산의 수행시간이 각각 O(logN)을 넘지 않는 매우 효율적인 자료구조**
- **일반적인 레드블랙트리(Intro. to Algorithms, CLRS)**
  - 삽입이나 삭제 수행시 트리의 균형을 유지하기 위해 상당히 많은 경우를 고려해야 한다는 단점이 있으며, 이에 따라 프로그램이 복잡하고, 그 길이도 증가
- **좌편향 레드블랙(Left-Leaning Red-Black, LLRB)트리**
  - 삽입이나 삭제 시 고려해야 하는 경우의 수가 매우 적어 프로그램의 길이도 일반 레드블랙트리 프로그램의 1/5정도에 불과

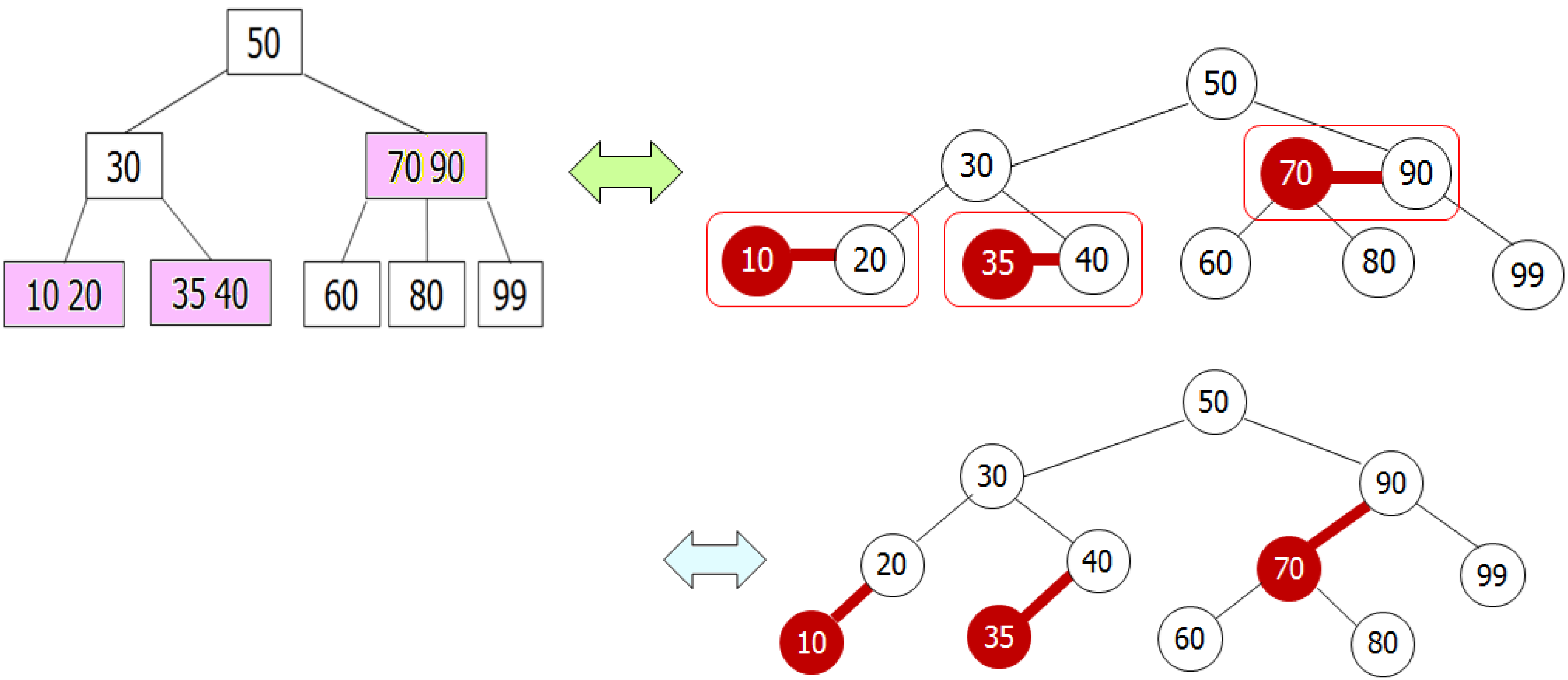


5.4 레드블랙트리

- LLRB 트리는 AVL-트리, 2-3트리, 2-3-4트리, 일반 레드블랙트리보다 매우 우수한 성능을 가짐
- Introduction to Algorithms (CLRS)에 소개된 레드블랙트리가 일반적으로 사용되며, 전문 프로그래머가 프로그램을 작성해도 적어도 400 line이나 든다.

[핵심 아이디어]

LLRB트리는 2-3트리에서 3-노드의 두 개의 키를 두 노드로 분리 저장하고, 작은 키는 레드, 큰 키는 블랙으로 만든 형태와 같다.

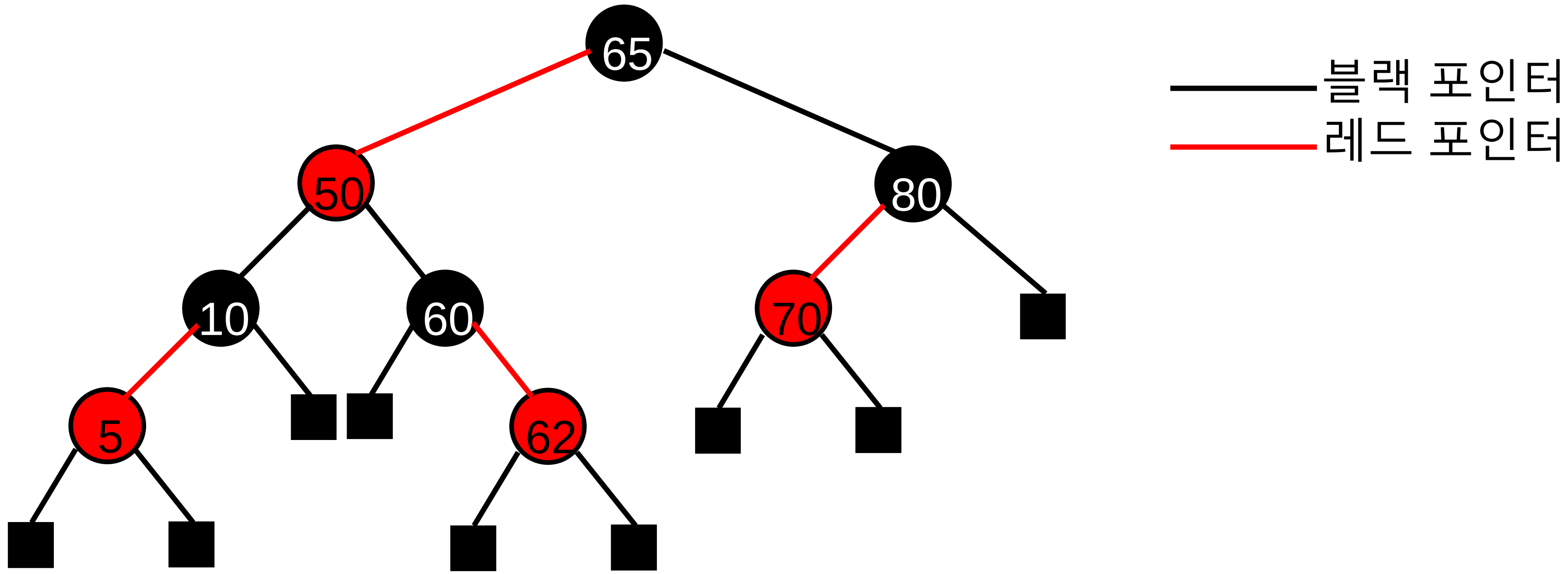


LLRB트리와 2-3트리의 관계

일반 레드-블랙 트리(1/15)

- **정의**
  - 노드의 컬러가 레드 또는 블랙인 이원 탐색트리
  - 확장이진트리인 레드-블랙 트리의 성질
    - RB1. 루트와 모든 외부 노드들은 컬러가 블랙이다.
    - RB2. 루트에서 외부 노드로 경로는 두 개의 연속적인 레드 노드를 가질 수 없다.
    - RB3. 루트에서 외부 노드로의 모든 경로들에 있는 블랙 노드의 수는 동일하다.
  - 포인터에 자식의 컬러를 부여하는 경우에 대한 성질
    - RB1'. 내부 노드로부터 외부 노드로의 포인터는 블랙이다.
    - RB2'. 루트에서 외부 노드로 경로는 두 개의 연속적인 레드 포인터를 가질 수 없다.
    - RB3'. 루트에서 외부 노드로의 모든 경들에 있는 블랙 포인터의 수는 동일하다.
  - 랭크: 한 노드로부터 외부노드로의 경로상에 있는 블랙포인터의 수
    (노드의 수에서 하나를 뺀 것과 동일)

일반 레드-블랙 트리(2/15)



- 확장이진트리인 레드-블랙 트리의 성질
  - RB1. 루트와 모든 외부 노드들은 컬러가 블랙이다.
  - RB2. 루트에서 외부 노드로 경로는 두 개의 연속적인 레드 노드를 가질 수 없다.
  - RB3. 루트에서 외부 노드로의 모든 경로들에 있는 블랙 노드의 수는 동일하다.
- 포인터에 자식의 컬러를 부여하는 경우에 대한 성질
  - RB1'. 내부 노드로부터 외부 노드로의 포인터는 블랙이다.
  - RB2'. 루트에서 외부 노드로 경로는 두 개의 연속적인 레드 포인터를 가질 수 없다.
  - RB3'. 루트에서 외부 노드로의 모든 경들에 있는 블랙 포인터의 수는 동일하다.

일반 레드-블랙 트리(3/15)

- **보조 정리 10.1**
  - 루트로부터 외부 노드로의 2개의 경로 P, Q가 있을때

    length(P) ≤ 2length(Q)

  - 증명
    - 임의 레드-블랙 트리에서 루트의 랭크를 r로 둠.
    - RB1'로부터 루트에서 외부노드로의 경로 상에 있는 마지막 포인터는 블랙
    - RB2'로부터 2개의 연속적인 레드 포인터를 갖는 경로 미존재
    - 각 레드포인터 뒤에는 항상 블랙포인터가 와야함.
    - 결과적으로 루트에서 외부노드로의 각 경로는 r, 2r사이에서 포인터를 갖게 됨.
    - 따라서 length(P)≤2length(Q) 임.

일반 레드-블랙 트리(4/15)

- **보조 정리 10.2**
  - 레드-블랙 트리 높이 h, 트리 내부 노드수 n, 랭크 r이면

    (a) $h \le 2r$

    (b) $n \ge 2^r - 1$

    (c) $h \le 2\log_2(n+1)$

  - 증명
    - 보조정리 10.1에서 length > 2r 은 존재하지 않음.
    - 따라서 $h \le 2r$
    - 레벨 1에서 r까지는 외부 노드가 없고, 이러한 레벨은 2^r-1개 내부노드가 있음.
    - 따라서 2^r-1은 내부노드의 최소한의 수가 됨.
    - (b)로부터 $h \le 2\log_2(n+1)$

일반 레드-블랙 트리(5/15)

- **레드-블랙 트리의 표현**
  - 구현에 있어 외부 노드를 표현하기 위해 물리적 노드보다 널 포인터 이용
  - 노드의 컬러를 저장하기 위해 노드당 1bit 필요
  - 자식 포인터의 컬러를 저장하기 위해 노드당 2bit 필요
- **레드-블랙 트리에서의 탐색**
  - 일반적인 이원 탐색 트리의 탐색에서 사용하는 알고리즘으로 탐색

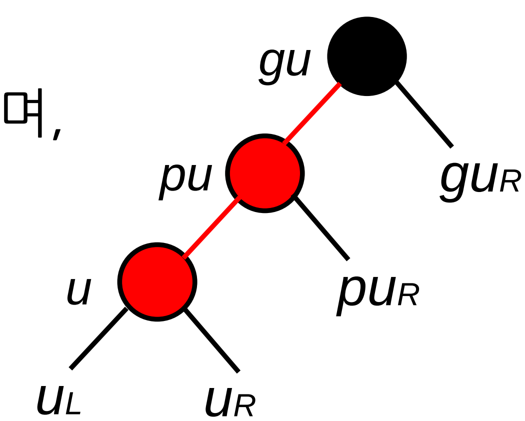
일반 레드-블랙 트리(6/15)

- **레드-블랙 트리로의 삽입에서의 노드의 컬러 지정**
  - 루트는 무조건 블랙.
  - 루트가 아닌 경우
    - 블랙으로 지정하는 경우 경로상 블랙 노드의 개수 문제 발생
    - 레드로 지정하는 경우 두개의 연속적인 레드 노드 발생 가능

    ➔새 노드는 무조건 레드로 지정하고 불균형을 조정

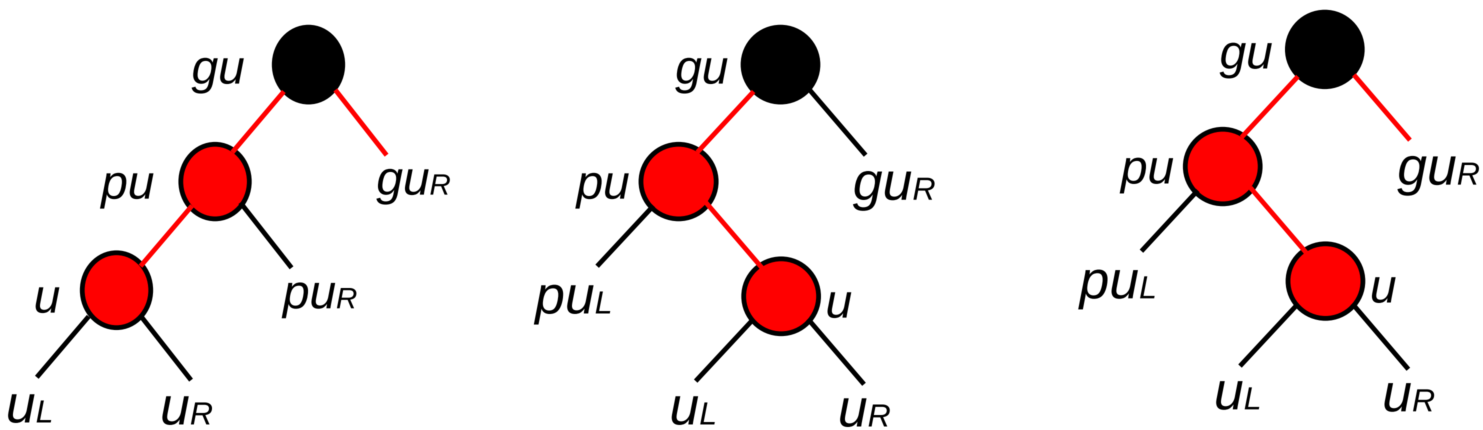
- **불균형 타입**
  - 새 노드 u, u의 부모 pu, u의 조부모 gu
  - LLb : pu가 gu의 왼쪽 자식이고 u는 pu의 왼쪽 자식이며,

    gu의 다른 자식이 블랙인 경우



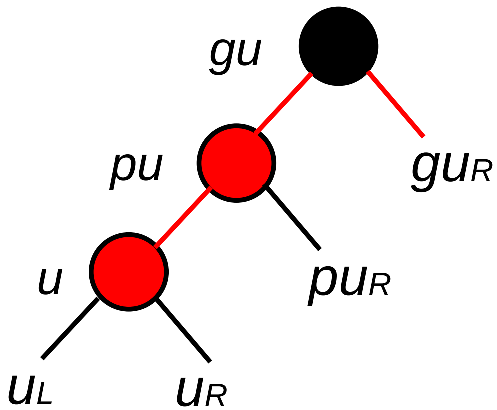
일반 레드-블랙 트리(7/15)

- LLr : pu가 gu의 왼쪽 자식이고 u는 pu의 왼쪽 자식이며,
  gu의 다른 자식이 레드인 경우
- LRb : pu가 gu의 왼쪽 자식이고 u는 pu의 오른쪽 자식이며,
  gu의 다른 자식이 블랙인 경우
- LRr : pu가 gu의 왼쪽 자식이고 u는 pu의 오른쪽 자식이며,
  gu의 다른 자식이 레드인 경우
- 위와 마찬가지로 RRb, RRr, RLb, RLr이 있다.

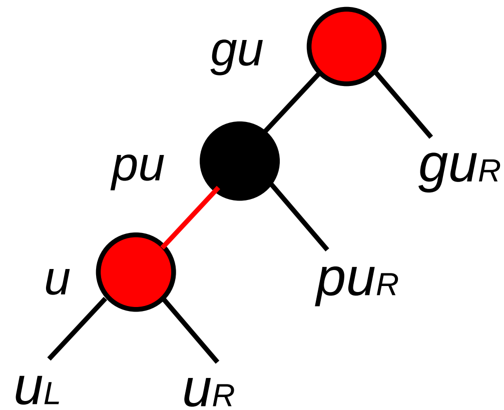


- 불균형 타입 XYr는 컬러에 의해 변경되지만, XYb은 회전이 필요함.

일반 레드-블랙 트리(8/15)

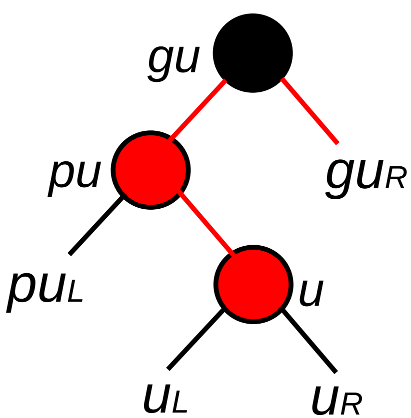


(a) LLr 불균형

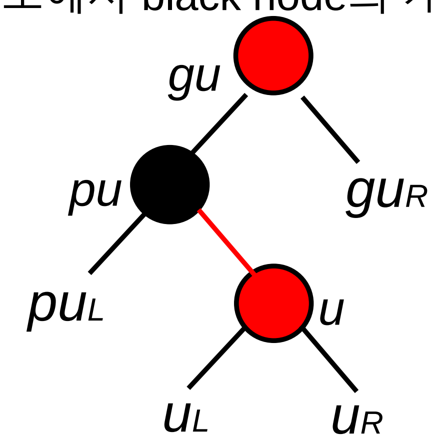


(b) LLr 컬러 변화 뒤

gu가 루트라면?
전체 경로에서 black node의 개수를 하나씩 증가



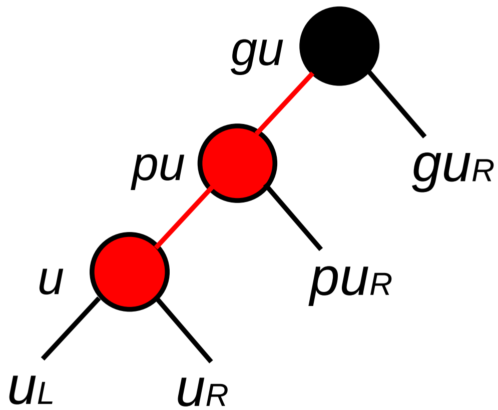
(c) LRr 불균형



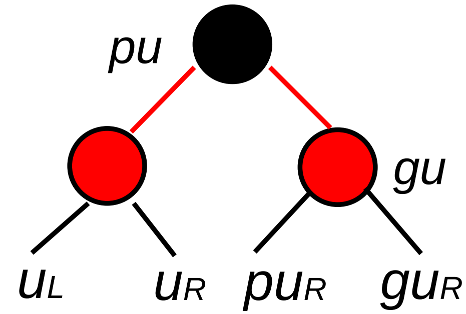
(d) LRr 컬러 변화 뒤

gu의 부모가 red node라면?
gu를 새로운 노드로 보고 root로 올라가면서 반복 적용

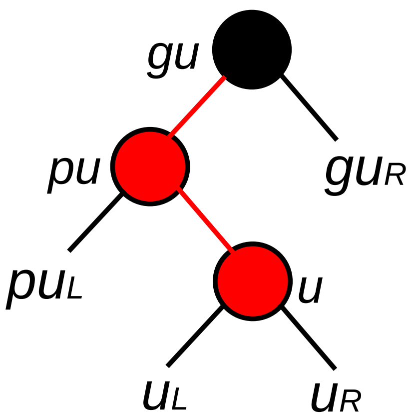
일반 레드-블랙 트리(9/15)



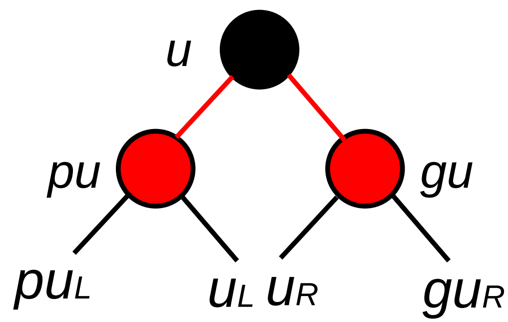
(a) LLb 불균형



(b) LLb 회전 뒤



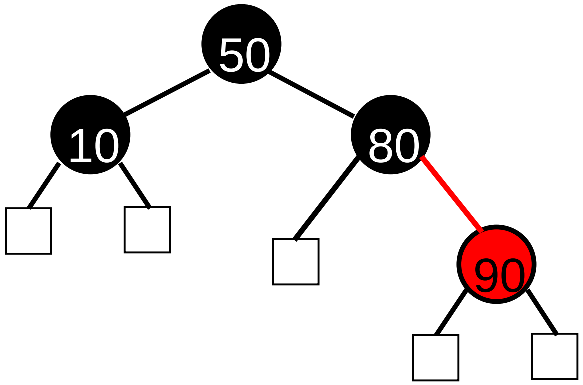
(c) LRb 불균형



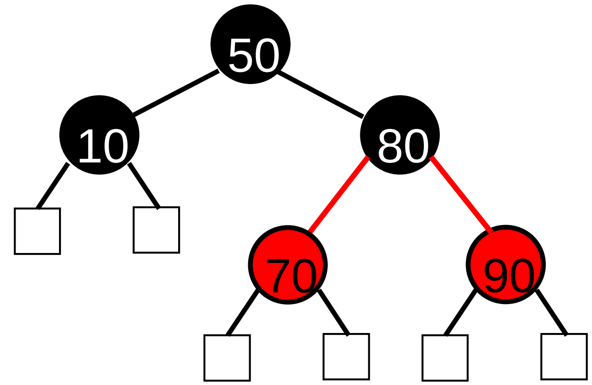
(d) LRb 회전 뒤

일반 레드-블랙 트리(10/15)

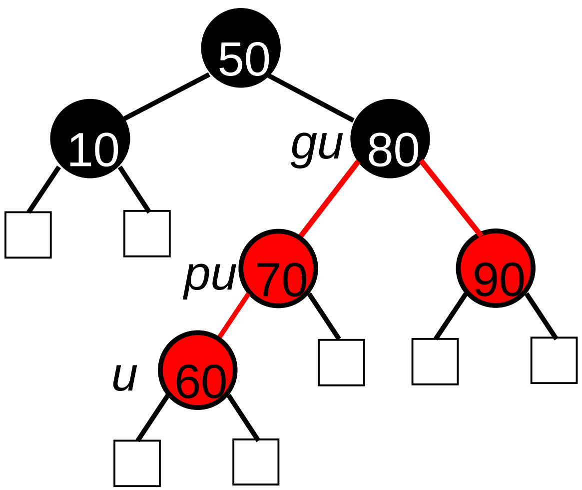
- **예제 10.4**



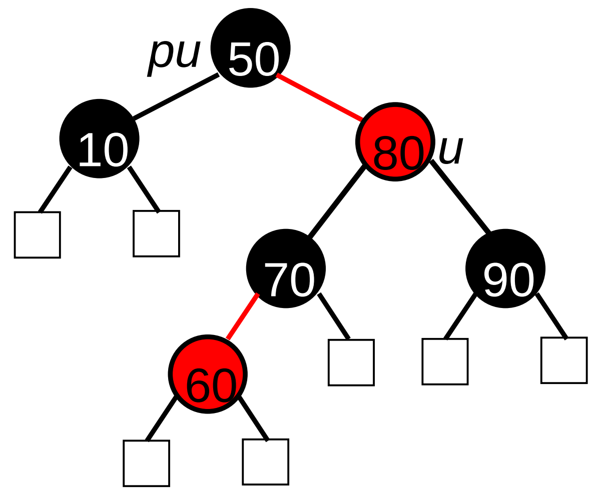
(a) 초기



(b) 70을 삽입

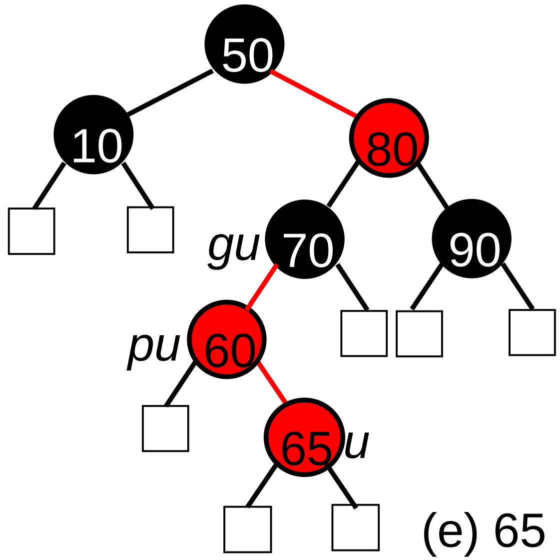


(c) 60을 삽입

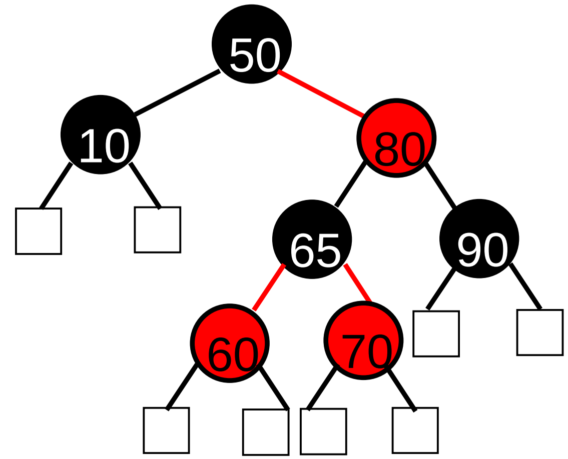


(d) LLr 컬러 변환

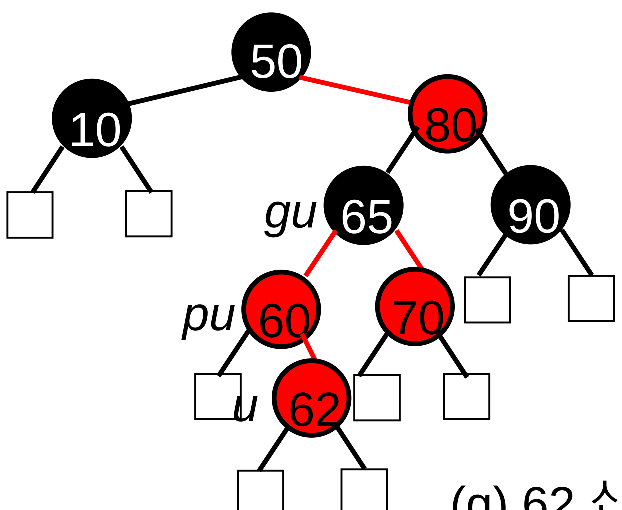
일반 레드-블랙 트리(11/15)



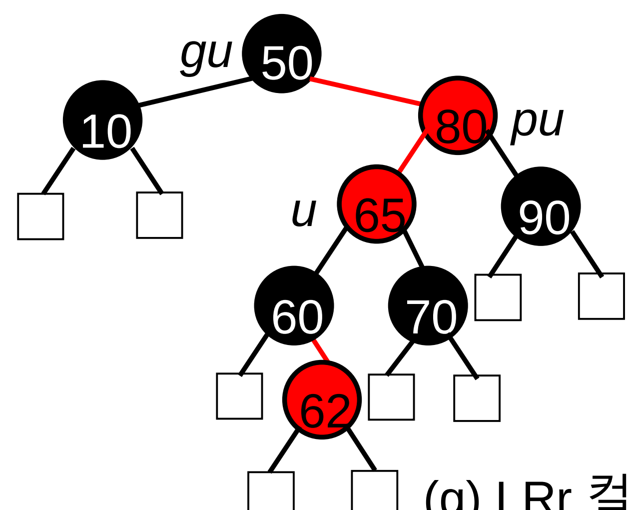
(e) 65 삽입



(f) LRb 회전

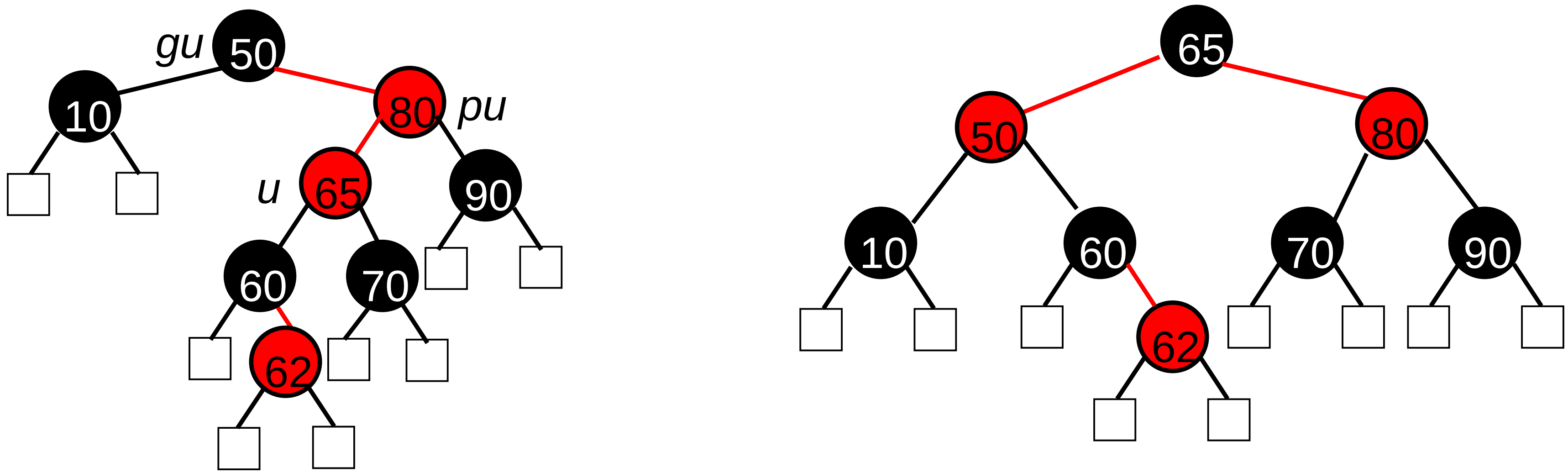


(g) 62 삽입



(g) LRr 컬러 변경

일반 레드-블랙 트리(12/15)



(i) RLb 회전

일반 레드-블랙 트리(13/15)

- **레드-블랙 트리에 조인 C.ThreeWayJoin(A, x, B)**
  - 경우1
    - A와 B가 같은 랭크를 갖는다면
    - A(=leftChild), x, B(=rightChild)의 쌍을 갖는 새로운 root를 생성함으로써 C가 만들어진다고 하자
    - C의 랭크는 A와 B의 랭크보다 하나 높다
  - 경우2
    - rank(A) > rank(B)를 갖는다면 A에서부터 B와 같은 랭크를 갖는 첫번째 노드 Y까지 rightChild 포인터를 따라간다.
    - p(Y)가 Y의 부모이면 rank(p(Y)) = rank(Y) + 1
    - P(Y)에서 Y로의 포인터는 블랙포인터
    - Y(=leftChild), x, B(rightChild)의 쌍을 갖는 새로운 노드 Z 생성
      - Z의 포인터는 레드포인터
  - 경우3
    - rank(A) < rank(B), 경우 2와 비슷

일반 레드-블랙 트리(14/15)

- **3원 조인 연산의 분석**
  - 기술된 함수가 정확한지에 대해서 쉽게 알수 있다.
    - 경우 1은 O(1)의 시간이 걸리고
    - 경우 2, 3은 O(|rank(A) - rank(B)|)이 걸린다
    - 따라서 3원 조인은 O(log n)의 시간에 수행될수 있으며 이때 n은 조인되고 있는 두 트리의 노드 수를 의미
- **레드-블랙 트리의 분할 A.Split(i, B, x, C)**
  - 단계 1
    - 키 값이 i인 원소를 포함하고 있는 노드 P를 찾기 위해 A를 탐색
    - 그 원소를 참조인자(parameter) x에 복사
    - B와 C를 P의 왼쪽과 오른쪽 서브트리가 되도록 초기화
  - 단계 2

```
for (Q = parameter(P); Q; P = Q, Q = parent(Q))  {
      if(P == Q→leftChild)  C.ThreeWayJoin(C,Q→data, Q→rightChild)
      else   B.ThreeWayjoin(Q→leftChild, Q→data, B);
}
```

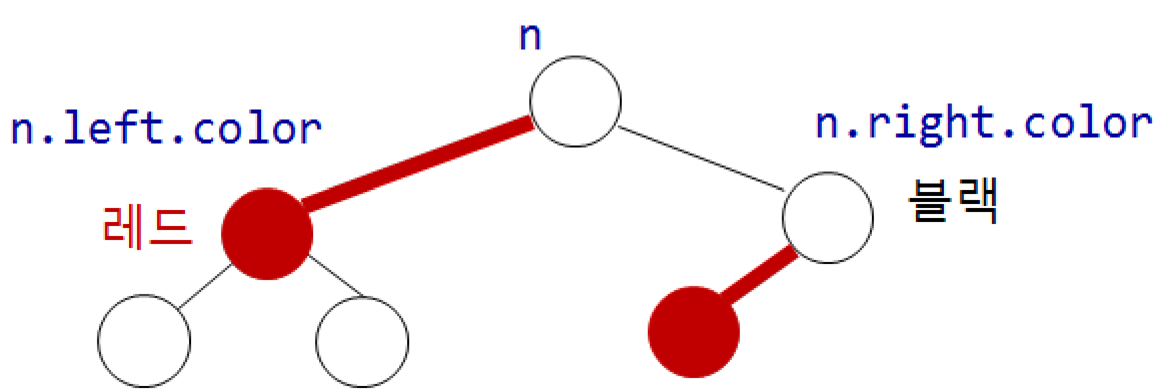
일반 레드-블랙 트리(15/15)

- **분할연산의 분석**
  - 분할되지 않은 트리의 노드 X의 랭크를 $r(X)$라 하면 $R(Q) \ge \max\{r(B), r(C)\}$
  - 정의로부터 Q의 부모 q'에 대해서 $r(q')=r(Q)+1$이고, $R(B') \le r(B)+1$이고 $r(C') \le r(C)+1$이므로, Q가 블랙 자식 P를 가진 노드를 가리킬 때, $R(q')=r(Q)+1 \ge \max\{r(B), r(C)\}+1 \ge \max\{r(B'), r(C')\}$ 성립

  - Q가 블랙 자식을 가진 한 노드를 가리킬 때 시작부터, Q가 이 노드에 도달할 때까지 분할 알고리즘 2단계에서 수행되는 모든 작업이 $O(r(B)+r(C)+r(Q))$를 알수 있음
  - $R(Q) \ge \max\{r(B), r(C)\}$이기 때문에 단계 2에서 이루어진 모든 작업은 $O(r(Q))$이다.
  - 요구되는 시간은 $O(\log n)$임을 알수 있음

5.4 좌편향 레드블랙트리

- **LLRB트리는 개념적으로 2-3트리와 같기 때문에 2-3트리의 장점인 완전 균형트리의 형태를 내포**
- **LLRB 트리의 노드는 블랙 또는 레드의 두 가지 색 정보를 가지며, 노드와 부모노드를 연결하는 link의 색은 노드의 색과 동일**
  - 따라서 LLRB 트리에서는 link의 색을 별도로 저장 안함
  - 아래 예제에서 노드 n의 왼쪽 자식노드는 레드이고 그 연결 link도 레드이며, n의 오른쪽 자식노드는 블랙이고 그 연결 link도 블랙



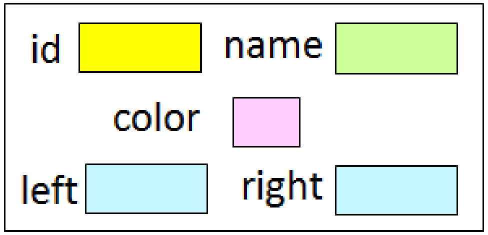
[정의] LLRB트리는 이진탐색트리로서 다음 네 가지 조건을 만족.

1. 루트노드와 null 은 블랙이다.
2. 루트노드로부터 각 null까지 2개의 연속된 레드 link는 없다. (연속 레드 link 규칙)
3. 루트노드로부터 각 null까지의 경로에 있는 블랙 link 수는 모두 같다. (동일 블랙 link 수 규칙)
4. 레드 link는 왼쪽으로 기울어져 있다. (레드 link 좌편향 규칙)

RedBlack Tree Class

```java
public class RedBlackTree<Key extends Comparable<Key>, Value> {
    private static final boolean RED   = true;
    private static final boolean BLACK = false;
    private Node root;
    private class Node {  // Node 클래스
        Key     id;
        Value   name;
        Node    left, right;
        boolean color;  // 부모노드 link의 색
        public Node(Key k, Value v, boolean col) {  // 노드 생성자
            id    = k;
            name  = v;
            color = col;
            left = right = null;
        }
    }
    private boolean isEmpty(){ return root == null;}
    private boolean isRed(Node n) {
        if (n == null) return false; // null의 색은 블랙
        return (n.color == RED);
    }

    // get(), put(), deleteMin(), delete()
    // 메소드들 선언
}
```



좌편향 레드블랙트리

- **RedBlackTree 클래스는 Node 클래스를 내부(Inner) 클래스로 선언**
- **Node 객체는 id(키), name(키에 관련된 정보), 왼쪽 자식과 오른쪽 자식을 각각 참조하기 위한 left와 right를 가지며, 노드의 색을 저장하기 위해 color를 가진다.**
  - 여기서 노드의 색은 노드의 부모와 연결된 link의 색과 동일하며, 색은 레드와 블랙 두 가지만을 사용하므로, boolean 타입을 사용
  - Line 01: Key와 Value는 generic 타입이고, Key는 비교 연산을 위해 자바의 Comparable 인터페이스를 상속 받으며, Comparable에 선언되어 있는 compareTo() 메소드를 통해 키를 비교
  - Line 02~03: 구현 및 사용 편의를 위해 RED를 true로, BLACK을 false로 정의

좌편향 레드블랙트리

- Line 04: root는 트리의 루트노드를 참조
- Line 05~16: Node 클래스
- Line 17: 트리가 empty일 때 true를 리턴하는 메소드
- Line 18: isRed() 메소드는 노드 n이 레드이면 true를, 아니면 false를 리턴. 단, line 19에서 노드가 null인 경우에도 false를 리턴
- Line 21이후: 탐색, 삽입, 최솟값 삭제를 위한 메소드 선언

```
01 public Value get(Key k) {return get(root, k);} // 탐색 연산
02 public Value get(Node n, Key k) {
03     if (n == null) return null;    // 탐색 실패
04     int t = n.id.compareTo(k);
05     if (t > 0)      return get(n.left,  k); // if (k < 노드 n의 id) 왼쪽 서브트리 탐색
06     else if (t < 0) return get(n.right, k); // if (k > 노드 n의 id) 오른쪽서브트리 탐색
07     else            return (Value) n.name;  // 탐색 성공
08 }
```

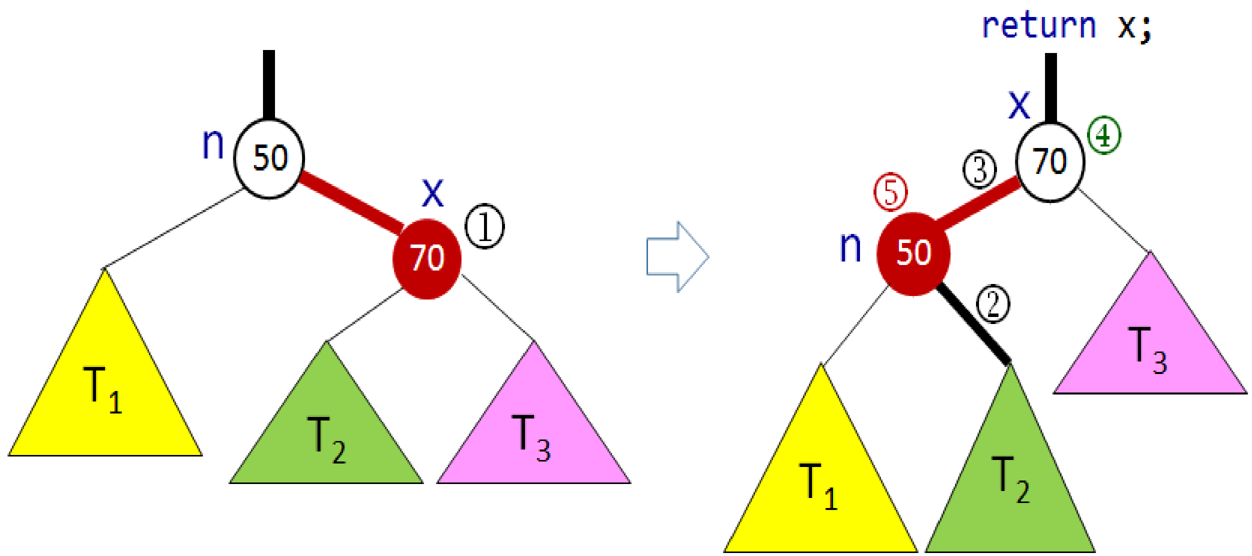
- 탐색하고자 하는 Key가 k 일 때, 루트의 id와 k를 비교하는 것으로 탐색 시작
- k가 id 보다 작은 경우에는 루트의 왼쪽 서브트리에서 k를 찾고, k가 id보다 큰 경우에는 루트의 오른쪽 서브트리에서 k를 찾으며, id가 k와 같으면 노드를 찾은 것이므로 찾아낸 노드의 Value, 즉, name을 리턴
- 왼쪽이나 오른쪽 서브트리에서 k를 탐색하는 것은 루트에서의 탐색과 동일
- 노드의 id를 k와 비교하는 line 04의 compareTo() 메소드는 id가 k 보다 작으면 음수, id가 k 보다 크면 양수, 같으면 0을 리턴

5.4.3 레드블랙트리의 기본 연산

- **LLRB 트리의 삽입, 삭제 연산을 위한 기본 연산**
  - rotateLeft: 노드의 오른쪽 레드 link를 왼쪽으로 옮기는 연산
  - rotateRight: 노드의 왼쪽 레드 link를 오른쪽으로 옮기는 연산
  - flipColors: 노드의 두 link의 색이 같을 때, 둘 다 다른 색으로 바꾸는 연산
- **회전이나 색 변환 연산은 삽입과 삭제 연산을 수행하는 도중에 트리의 규칙에 어긋나는 부분을 수정하는데 이용**

좌편향 레드블랙트리- rotateLeft 연산

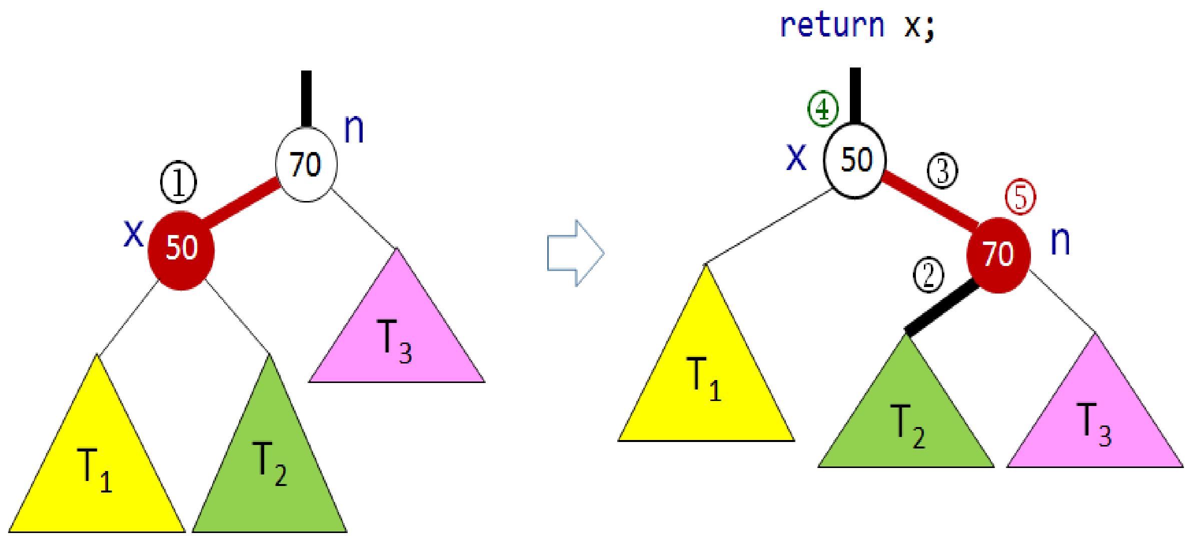
```
private Node rotateLeft(Node n){
  ① Node x   = n.right;
  ② n.right = x.left;
  ③ x.left  = n;
  ④ x.color = n.color;
  ⑤ n.color = RED;
     return x;
}
```



- 50을 n이라 할 때, 오른쪽 자식인 노드 x가 왼쪽으로 회전하여 n의 자리로 이동하고, 노드 색이 블랙으로 되며, n은 x의 왼쪽 자식으로서 레드 link로 연결
  - 동일한 개수의 블랙 link를 유지 가능

좌편향 레드블랙트리 - rotateRight 연산

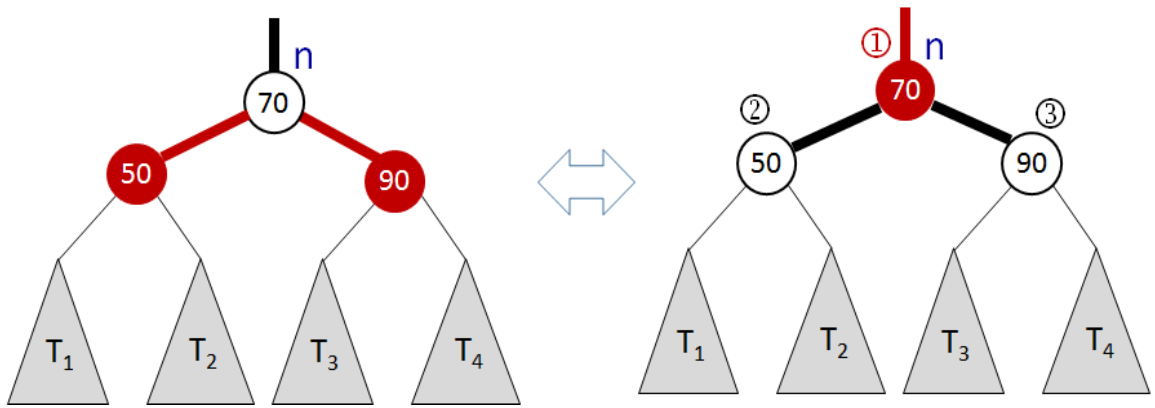
```
01 private Node rotateRight(Node n){
02   ① Node x   = n.left;
03   ② n.left   = x.right;
04   ③ x.right = n;
05   ④ x.color = n.color;
06   ⑤ n.color = RED;
07      return x;
08 }
```



- rotateRight는 노드 n의 왼쪽 레드 link를 오른쪽으로 옮기는 메소드이고, rotateRight()의 line 02～06에 각각 붙여진 번호순으로link와 색이 변경
- 삽입이나 삭제 연산 중에 노드 n의 왼쪽 방향에 발생한 연속 레드 link문제를 해결하기 위해 rotateRight()를 사용

좌편향 레드블랙트리 - flipColors 연산

```
private void flipColors(Node n){
  ① n.color = !n.color;
  ② n.left.color  = !n.left.color;
  ③ n.right.color = !n.right.color;
}
```



(a) (b)

- 색 변환연산은 (a)에서 (b)로 수행하는 경우와
  (b)에서 (a)로 각각 수행하는 경우를 모두 포함

좌편향 레드블랙트리 - 삽입 연산

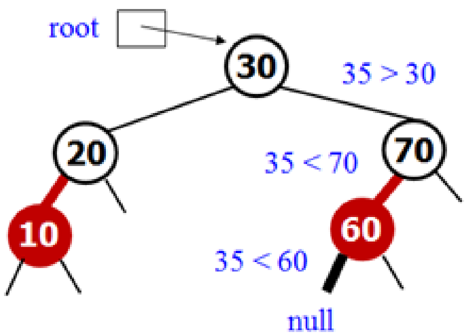
- step1) 삽입하고자 하는 키를 탐색하여 노드의 자식이 null인 곳에
  새로운 노드를 레드 노드로 생성 (블랙 link수 유지)
- step2) 오른쪽 자신이 레드이고 왼쪽 자식이 블랙이면 rotateLeft 수행
- step3) 왼쪽 자식과 왼쪽왼쪽 손자가 모두 레드이면 rotateRight수행
- step4) 왼쪽 오른쪽이 모두 레드이면 flipColors 수행

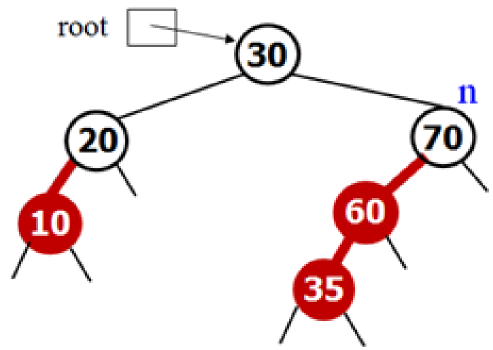
좌편향 레드블랙트리 - 삽입연산

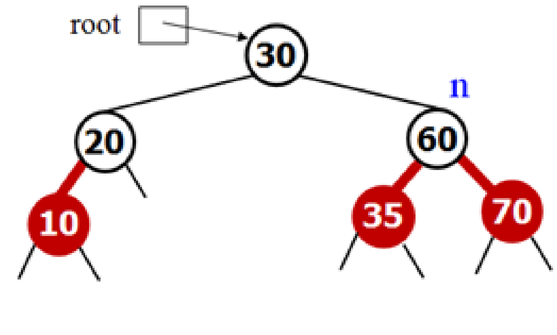
```
01  public void put(Key k, Value v) {
02      root = put(root, k, v);
03      root.color = BLACK;
04  }
05  private Node put(Node n, Key k, Value v) {
06      if (n == null)  return new Node(k, v, RED);  // 새로운 레드 노드 생성
07      int t = k.compareTo(n.id);
08      if (t < 0)      n.left  = put(n.left,  k, v);
09      else if (t > 0) n.right = put(n.right, k, v);
10      else            n.name  = v;    // k가 트리에 있는 경우 v로 name을 갱신
11      // 오른쪽 link가 레드인 경우 바로잡는다.
12      if (!isRed(n.left)&& isRed(n.right))     n = rotateLeft(n);
13      if (isRed(n.left) && isRed(n.left.left)) n = rotateRight(n);
14      if (isRed(n.left) && isRed(n.right))     flipColors(n);
15      return n;
16  }
```

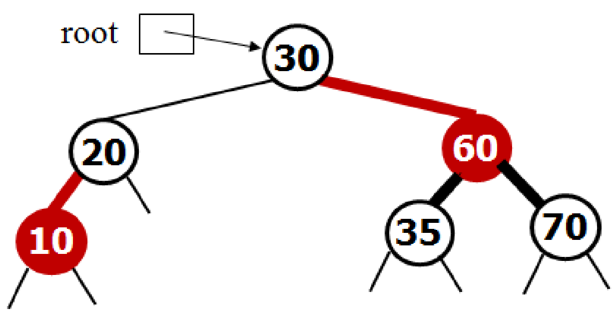
- Line 01: put() 메소드는 line 05의 put() 메소드를 호출
- Line 02: root가 line 05의 put() 메소드로 리턴되는 Node를 가리키도록 한다.
- Line 08과 09: n.left와n.right를put() 메소드가 리턴하는 Node와 각각 연결시키는데, 이는 새로 삽입된 노드로부터 루트노드까지 올라가기 위함
- Line 12 ~ 14: 새 노드를 삽입한 후에 발생할 수 있는 연속 레드 link문제를 해결하기 위해 rotateRight, rotateLeft, flipColors를 차례로 수행
- 마지막으로 호출이 리턴되는 line 02에서는 root가 루트노드를 가리키며, line 03에서 루트노드를 (레드인 경우도 있으므로) 블랙으로 만든 후 삽입 연산을 종료

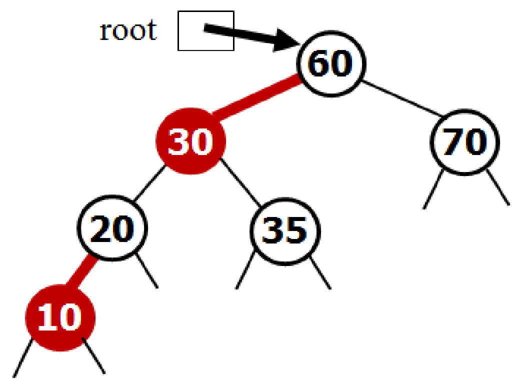
[예제] 35 삽입


root
30
35 > 30
20
70
35 < 70
10
60
35 < 60
null


root
30
n
20
70
10
60
35


root
30
n
20
60
10
35
70


root
30
20
60
10
35
70


root
60
30
70
20
35
10

좌편향 레드블랙트리 - 최솟값 삭제 연산

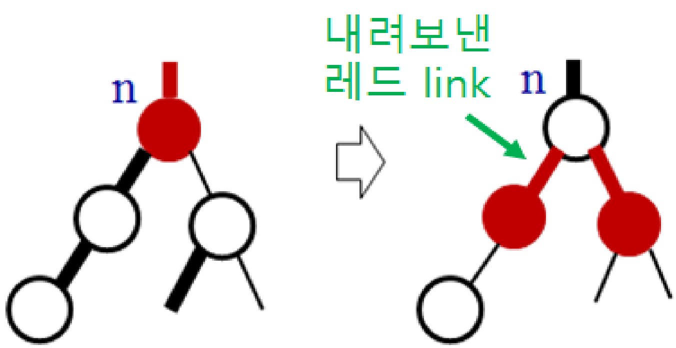
[핵심 아이디어]

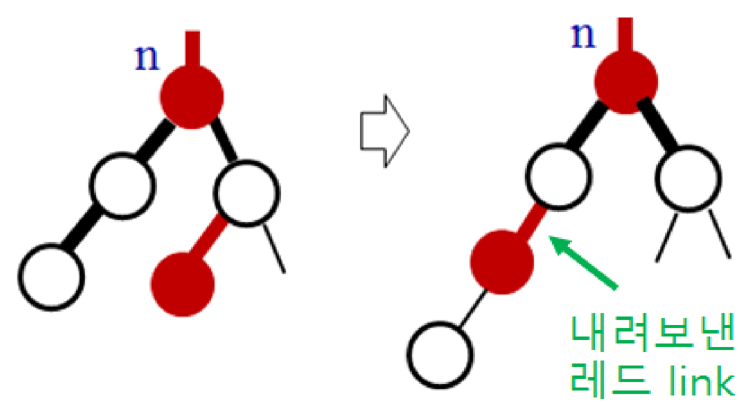
루트노드로부터 삭제하는 노드 방향으로 레드 link를 옮기어
궁극적으로 삭제되는 노드를 레드로 만든 후에 삭제한다

- 루트로부터 삭제하는 노드 방향으로 레드 link를 옮기는 과정은 트리의 조건을 위반하지 않는 상태를 유지하며 진행
- 이를 위해 2 가지 방법으로 레드 link를 왼쪽 아래로 내려 보낸다.
- 다만 레드 link 좌편향 규칙에 위배되는 경우가 발생할 수 있으나 이는 삭제 후에 다시 루트 방향으로 올라가면서 수정

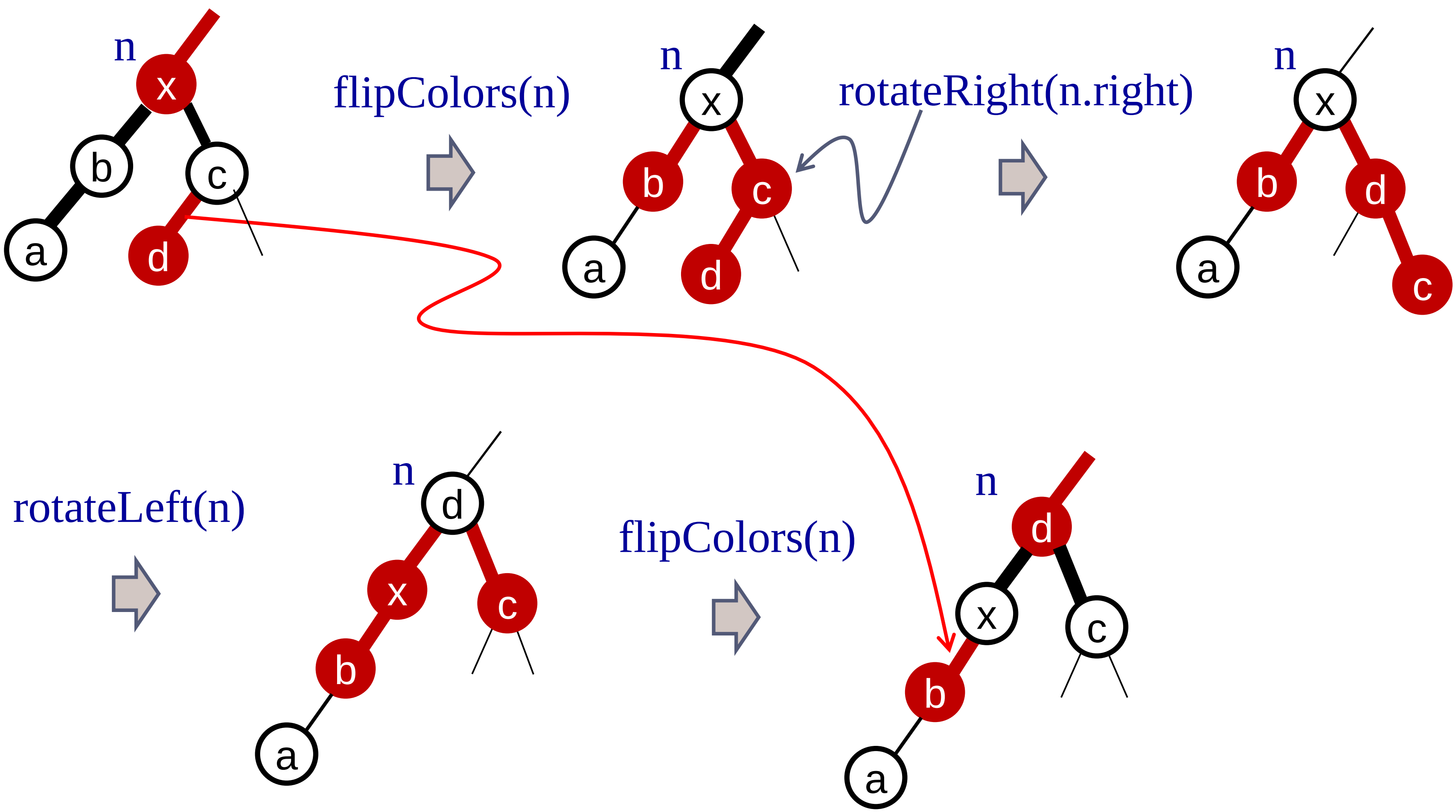
[case 1] n.left와 n.left.left가 모두 블랙이고,
동시에 n.right.left도 블랙이면, flipColors(n)을 수행

[case 2] n.left 와 n.left.left가 모두 블랙이고,
동시에 n.right.left가 레드이면,
n.right.left의 레드 link를 왼쪽 방향으로 보낸다.





Case 2는 다음과 같은 일련의 기본 연산을 통해 레드 link를 왼쪽 아래로 내려 보낸다.



- 다음은 두 가지 경우를 모두 고려한 moveRedLeft() 메소드.
- Case 1과2의 공통된 첫 연산은 flipColors
- Case 2는 세 개의 연산(line 04~06)이 추가로 필요

```
01  private Node moveRedLeft(Node n){
02      flipColors(n);          // case 1과 case 2
03      if (isRed(n.right.left)) {   // case 2
04          n.right = rotateRight(n.right);
05          n         = rotateLeft(n);
06          flipColors(n);
07      }
08      return n;
09  }
```

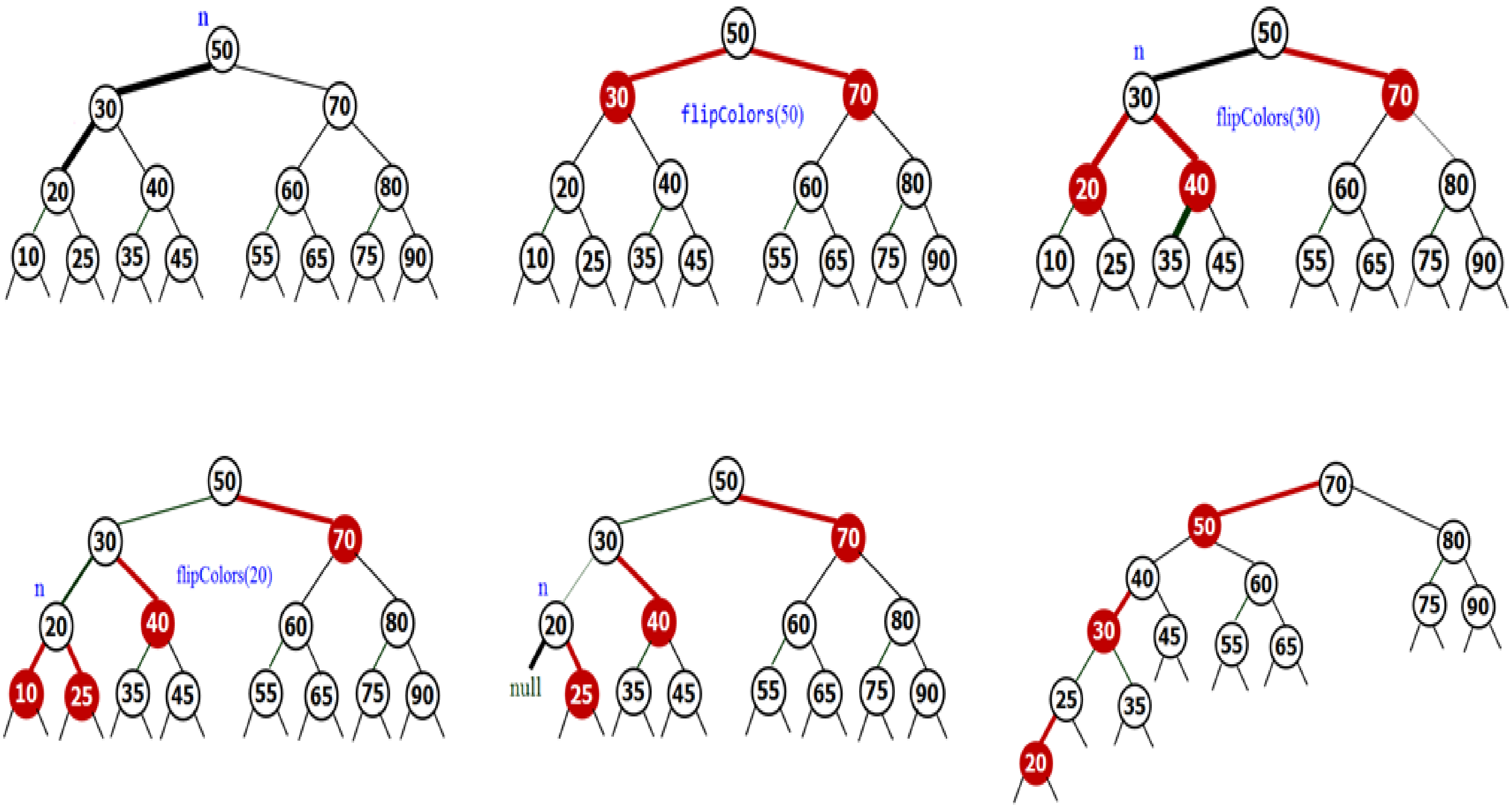
```java
01 public void deleteMin() { // 최솟값 삭제
02     root = deleteMin(root);
03     root.color = BLACK;
04 }
05 private Node deleteMin(Node n) {
06     if (n.left == null)  return null;
07     if (!isRed(n.left) && !isRed(n.left.left))
08         n = moveRedLeft(n);
09     n.left = deleteMin(n.left);
10     return fixUp(n);
11 }
```

- Line 06에서 (n.left == null)이면 노드 n이 최솟값을 가진 노드인 것으로, 이때 단순히 null을 리턴. 그 이유는 노드 n이 레드 노드로 만들어졌기 때문에, 왼쪽자식이 null인 상태에서 오른쪽 자식노드가 존재할 수 없기 때문
  - 만일 오른쪽 자식노드가 있다면, 이 노드는 블랙 또는 레드
  - 오른쪽 자식노드가 **블랙**이면 동일 블랙 link 수 규칙에 위배되고, 레드이면 좌편향 레드 link 규칙에 위배되므로 어떤 경우에도 LLRB 규칙을 만족하지 못한다.

fixUp() 메소드는 레드블랙트리 규칙에 어긋난 부분을 수정

```java
private Node fixUp(Node n) {
    if (isRed(n.right))   n = rotateLeft(n);
    if (isRed(n.left) && isRed(n.left.left))  n = rotateRight(n);
    if (isRed(n.left) && isRed(n.right))  flipColors(n);
    return n;
}
```

[예제] deleteMin() 의 수행 과정



좌편향 레드블랙트리 - 수행시간

- LLRB트리에서 삽입과 삭제 연산은 공통적으로 루트노드부터 탐색을 시작하여 이파리까지 내려가고, 다시 루트노드까지 거슬러 올라온다.

- 트리를 한 층 내려갈 때나 올라갈 때에 수행되는 연산은 각각 O(1) 시간 밖에 소요되지 않으므로 삽입과 삭제 연산의 수행시간은 각각 트리의 높이에 비례

- N개의 노드를 갖는 레드블랙트리의 높이 h는 2log N 보다 크지 않다.
  - 루트부터 이파리까지 블랙 link 수가 동일하므로 레드 노드가 없는 경우에는 $h = \log N$이며, 레드 노드가 최대로 많이 트리에 있는 경우에도 레드 link가 연속해서 존재할 수 없으므로 $h \le 2\log N$이다.

좌편향 레드블랙트리 - 응용

- **레드블랙트리는 반드시 제한된 시간 내에 연산이 수행되어야 하는 경우에 매우 적합한 자료구조이다.**
- **실제 응용사례로는 logN 시간보다 조금이라도 지체될 경우 매우 치명적인 상황을 야기할 수 있는**
  - 항공 교통 관제(Air Traffic Control)
  - 핵발전소의 원자로(Nuclear Reactor) 제어
  - 심장박동 조정장치(Pacemakers) 등

  - 레드블랙트리는 자바의 java.util.TreeMap과 java.util.TreeSet의 기본 자료구조로 사용되며, C++ 표준 라이브러리인 map, multimap, set, multiset에도 사용되고, 리눅스(Linux) 운영체제의 스케줄러에서도 레드블랙트리가 활용

5.5 B-트리

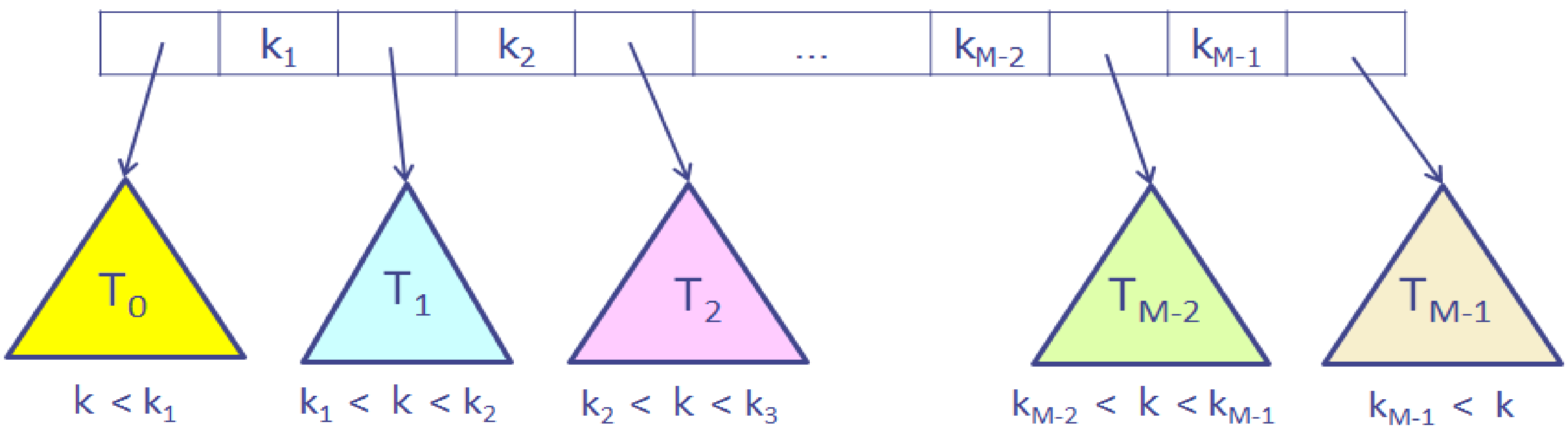
- 다수의 키를 가진 노드로 구성되어
  다방향 탐색(Multiway Search)이 가능한 균형 트리
- 2-3트리는 B-트리의 일종으로 노드에 키가 2 개까지 있을 수 있는 트리
- B-트리는 대용량의 데이터를 위해 고안되어 주로 데이터베이스에 사용

[핵심아이디어]

노드에 수백에서 수천 개의 키를 저장하여 트리의 높이를 낮추자.

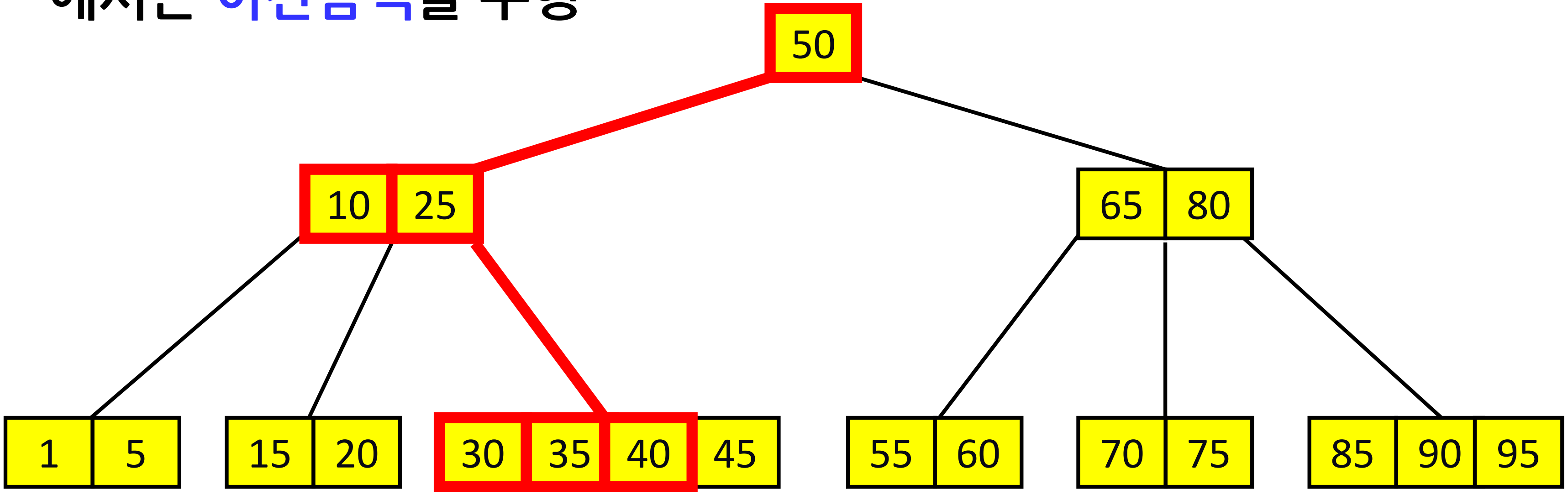
[정의] 차수가 M인 B-트리는

- 모든 이파리노드들은 동일한 깊이를 갖는다.
- 각 내부노드의 자식 수는 $\lceil M/2 \rceil$ 이상 M 이하이다.
- 루트노드의 자식 수는 2 이상이다.



5.5.1 탐색 연산

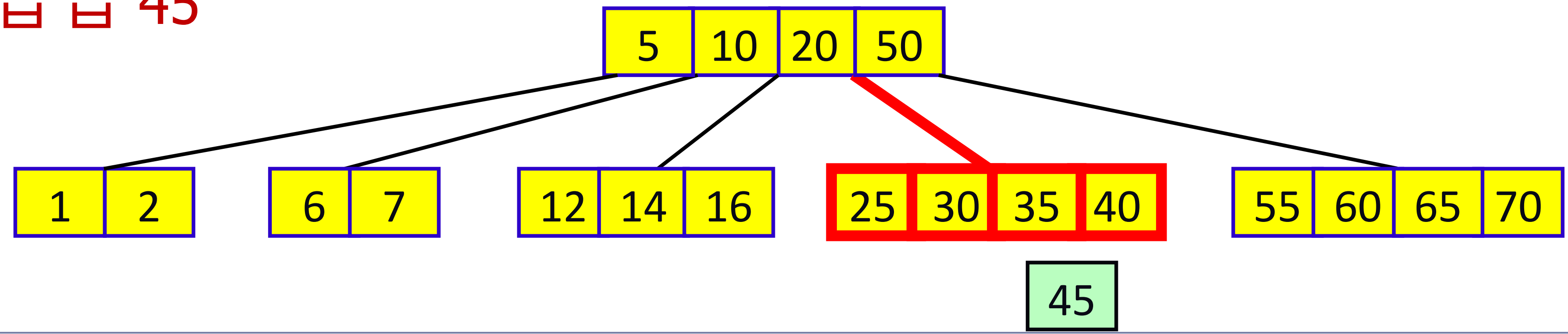
- B-트리에서의 탐색은 루트로부터 시작된다.
- 방문한 각 노드에서는 탐색하고자 하는 키와 노드의 키들을 비교하여, 적절한 서브트리를 탐색
- 단, B-트리의 노드는 일반적으로 수백 개가 넘는 키를 가지므로 각 노드에서는 이진탐색을 수행

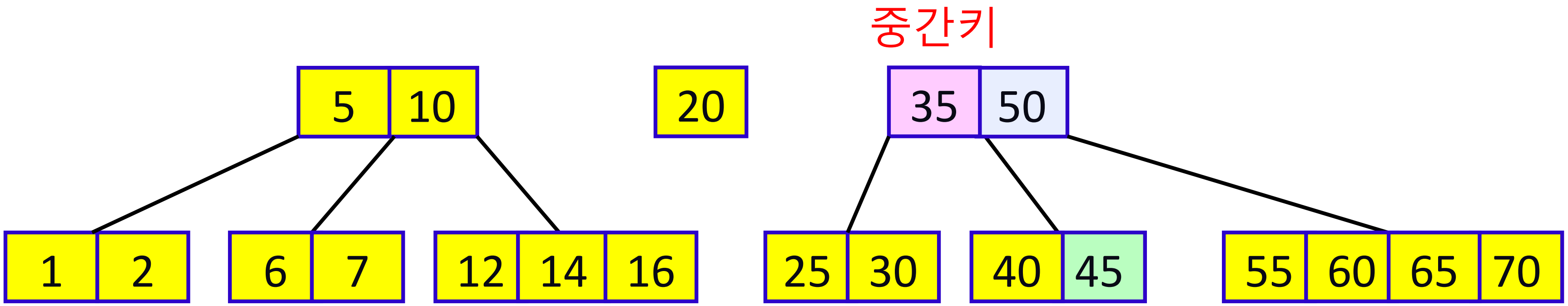


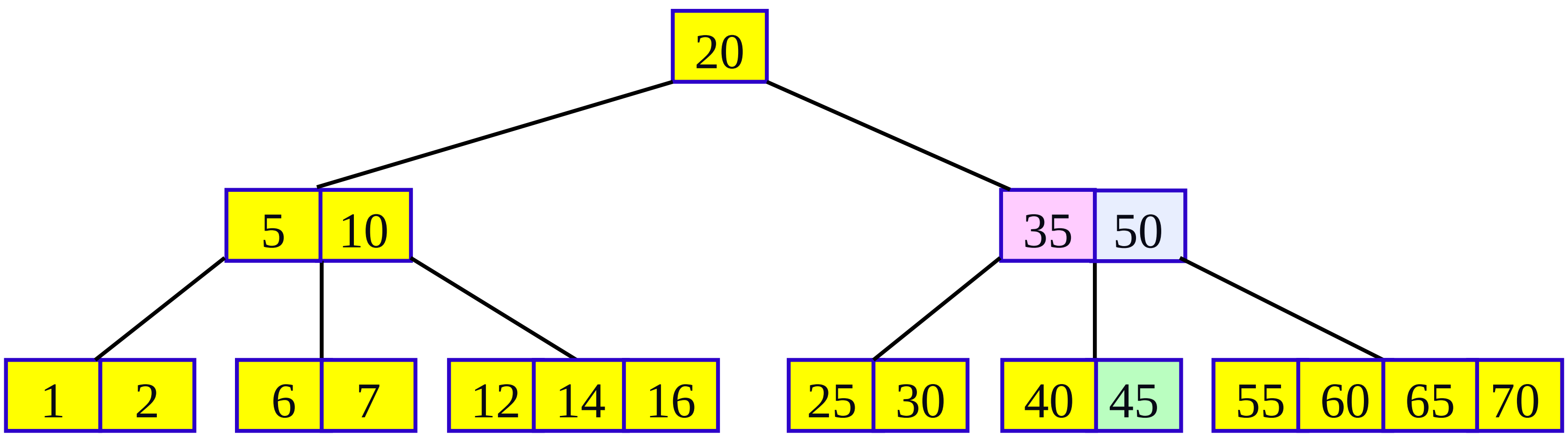
5.5.2 삽입 연산

- B-트리에서의 삽입은 탐색과 동일한 과정을 거쳐 새로운 키가 저장되어야 할 이파리노드를 찾는다.
- 이파리노드에 새 키를 수용할 공간이 있다면, 노드의 키들이 정렬 상태를 유지하도록 새 키를 삽입
- 이파리노드가 이미 M-1개의 키를 가지고 있으면, 이 M-1개의 키들과 새로운 키 중에서 중간값이 되는 키(중간키)를 부모노드로 올려 보내고, 남은 M-1개의 키들을 1/2씩 나누어 각각 별도의 노드에 저장한다. [분리(Split) 연산]

삽입 45


5 10 20 50
1 2
6 7
12 14 16
25 30 35 40
55 60 65 70
45

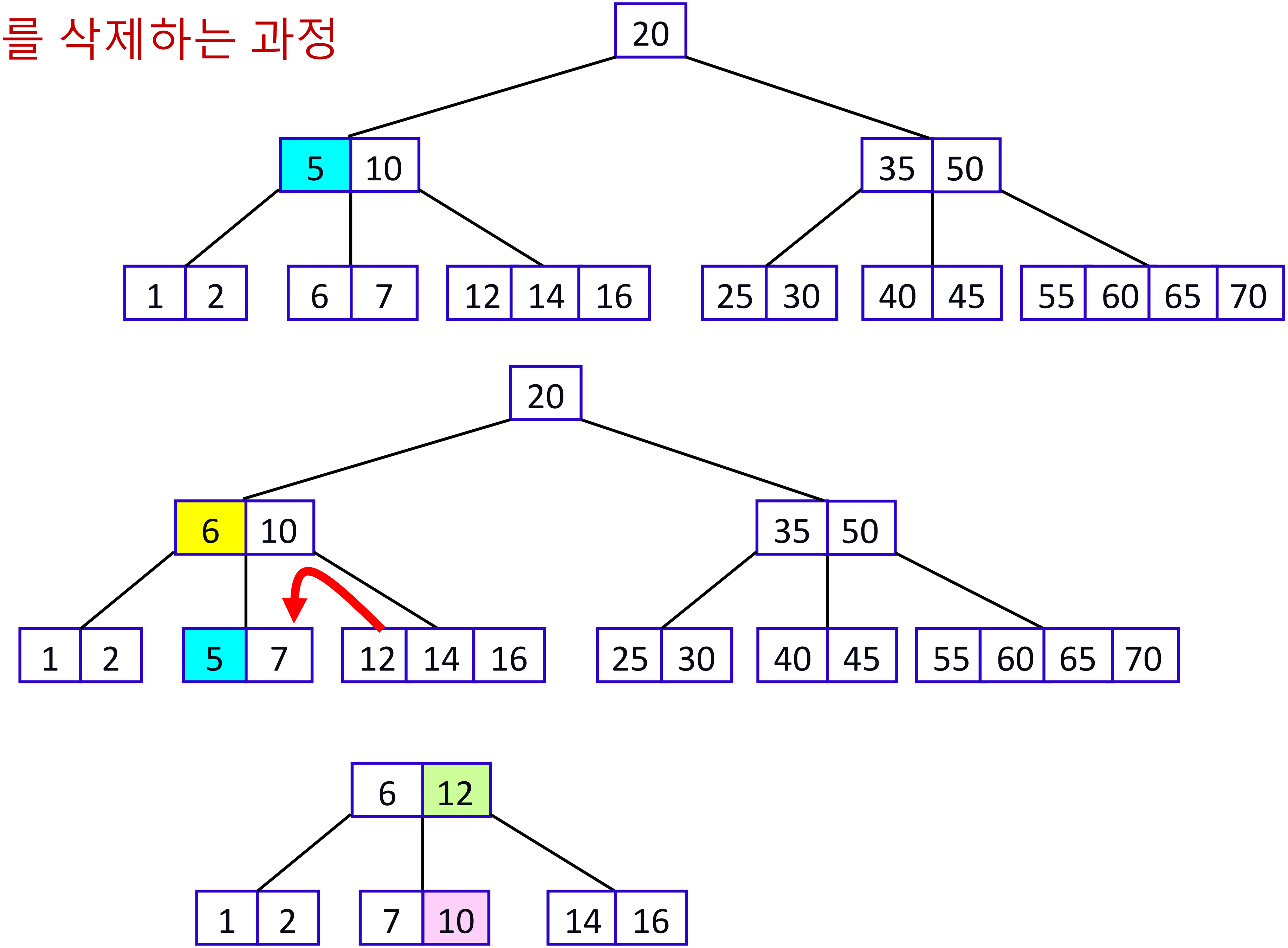

중간키
5 10
20
35 50
1 2
6 7
12 14 16
25 30
40 45
55 60 65 70


20
5 10
35 50
1 2
6 7
12 14 16
25 30
40 45
55 60 65 70

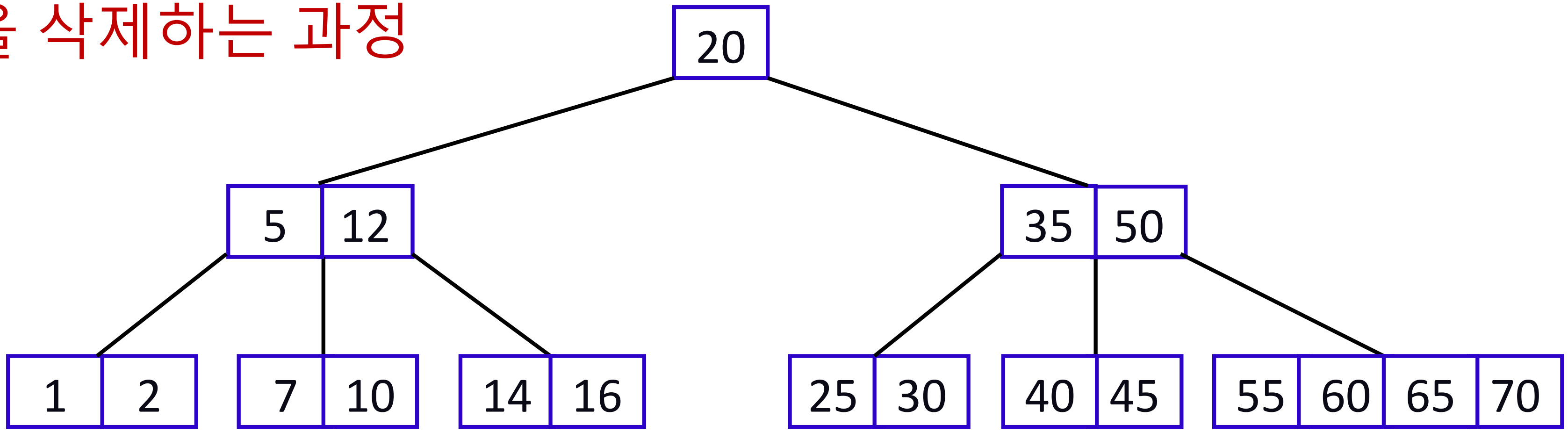
5.5.3 삭제 연산

- **B-트리에서의 삭제는 항상 이파리노드에서 이루어짐.**
  - 이파리노드가 아니면, 이진탐색트리의 삭제와 유사하게 중위 선행자나 중위 후속자를 삭제할 키와 교환한 후에 이파리노드에서 삭제를 수행
- **삭제는 이동(Transfer) 연산과 통합(Fusion) 연산을 사용**
- **이동 연산**
  - 이파리노드에서 키가 삭제된 후에 키의 수가 $\lceil M/2 \rceil$-1보다 작으면, 자식 수가 $\lceil M/2 \rceil$보다 작게 되어 B-트리 조건을 위반
  - 이 때 노드의 좌우의 형제노드들 중에서 도움을 줄 수 있는 노드로부터 1 개의 키를 부모노드를 통해 이동
- **통합 연산**
  - 키가 삭제된 후 underflow가 발생한 노드 x에 대해 이동 연산이 불가능한 경우 노드 x와 그의 형제노드를 1 개의 노드로 통합하고, 노드 x와 그의 형제노드의 분기점 역할을 하던 부모노드의 키를 통합된 노드로 끌어내리는 연산

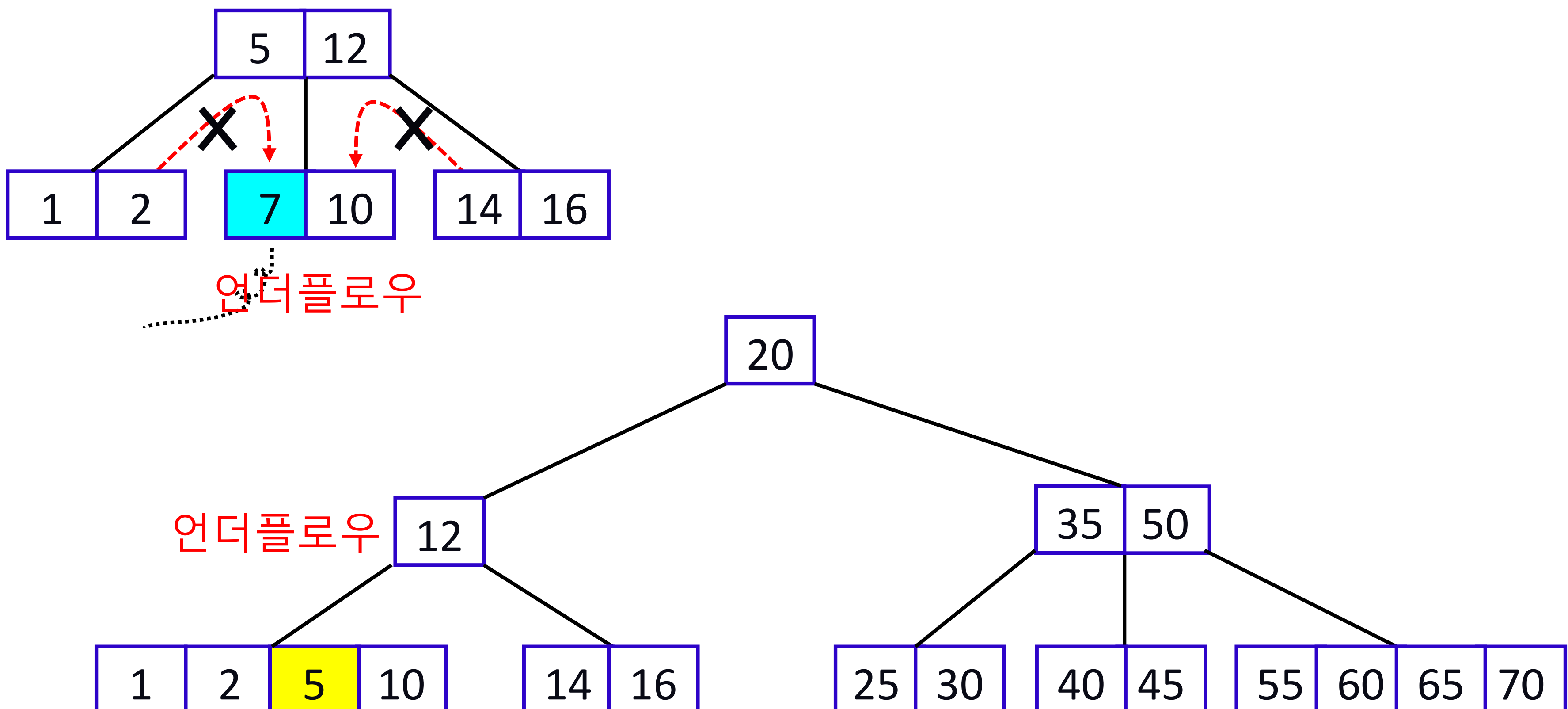
5를 삭제하는 과정


20
5 10
35 50
1 2
6 7
12 14 16
25 30
40 45
55 60 65 70
20
6 10
35 50
1 2
5 7
12 14 16
25 30
40 45
55 60 65 70
6 12
1 2
7 10
14 16

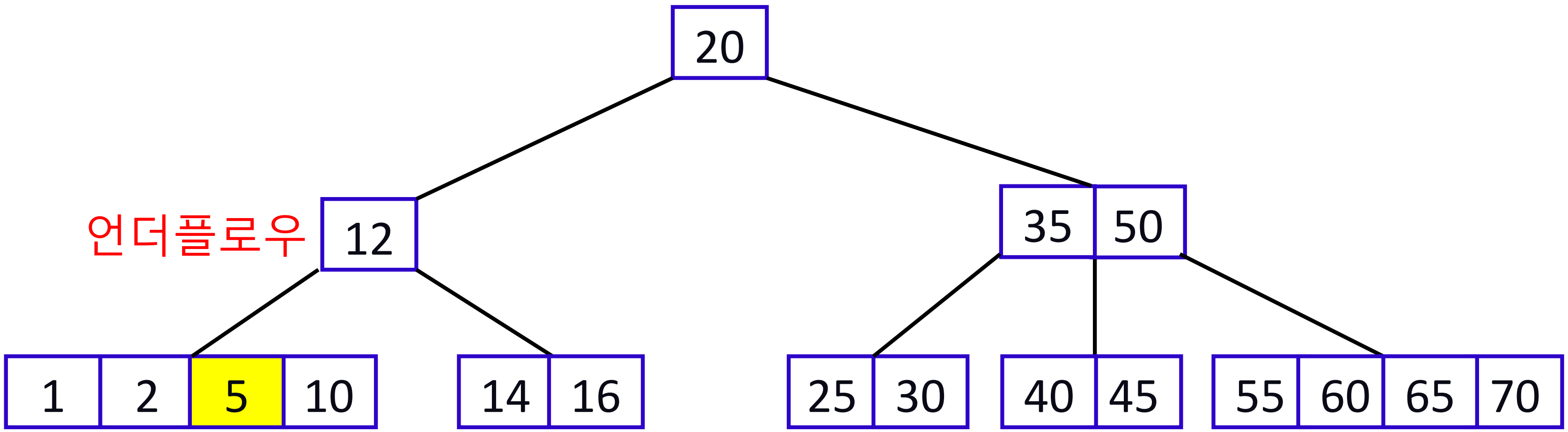
7을 삭제하는 과정

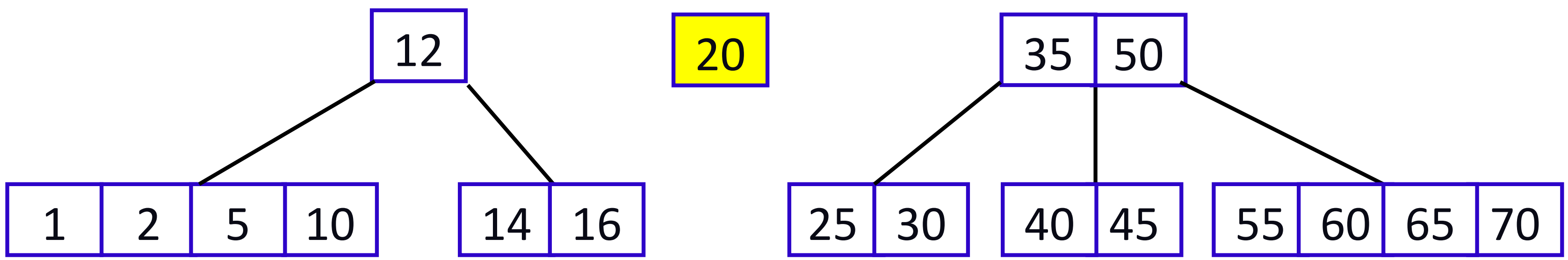


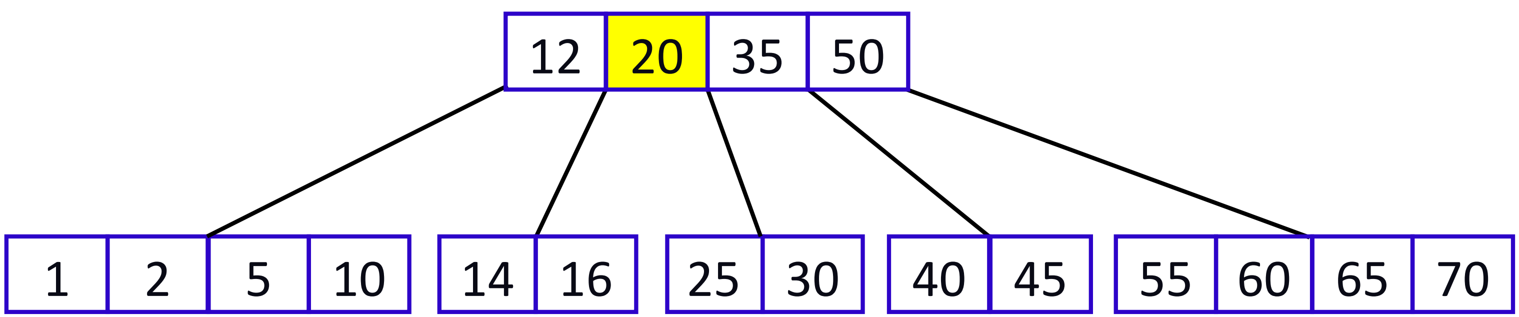
20
5 12
35 50
1 2
7 10
14 16
25 30
40 45
55 60 65 70



5 12
1 2
7 10
14 16
언더플로우
20
언더플로우
12
35 50
1 2 5 10
14 16
25 30
40 45
55 60 65 70


20
언더플로우
12
35 50
1 2 5 10
14 16
25 30
40 45
55 60 65 70


12
20
35 50
1 2 5 10
14 16
25 30
40 45
55 60 65 70

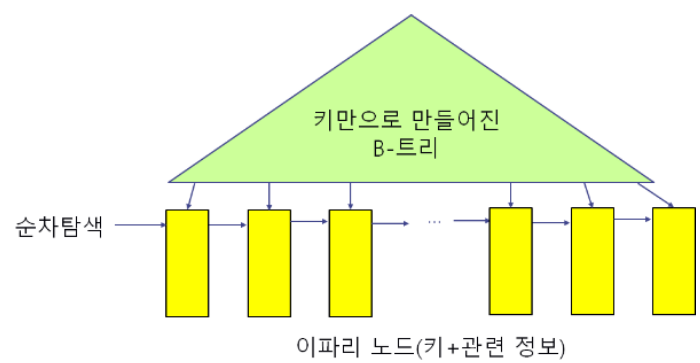

12 20 35 50
1 2 5 10
14 16
25 30
40 45
55 60 65 70

성능 분석

- B-트리에서 삽입이나 삭제 연산의 수행시간은 각각 B- 트리의 높이에 비례. 차수가 M이고 키의 개수가 N인 B-트리의 최대 높이는 O(log M/2 N)이다.
- B-트리는 키들의 비교 횟수보다 디스크와 메인 메모리 사이의 블록 이동(Transfer) 수를 최소화해야
- B-트리의 최고 성능을 위해선 1 개의 노드가 1 개의 디스크 페이지에 맞도록 차수 M을 정함
- 실제로 B-트리들은 M의 크기를 수백에서 수천으로 사용
  - 예를 들어, M = 200이고 N = 1억이라면 B-트리의 연산은4개의 디스크 블록만 메인 메모리로 읽어 들이면 처리 가능하다.
- 성능향상을 위해 루트는 메인 메모리에 상주시킨다.

5.5.4 B-트리의 확장

- **B*-트리는 B-트리로서 루트를 제외한 다른 노드의 자식 수가 2/3M ~ M이어야 한다.**
  - 즉, 각 노드에 적어도 2/3 이상이 키들로 채워져 있어야
  - B-트리에 비해 B*-트리는 공간을 효율적으로 활용
- **B+-트리는 실세계에서 가장 널리 활용되는 B-트리**
  - B+-트리는 키들만으로 가지고 B-트리를 구성, 이파리노드에 키와 관련(실제) 정보를 저장
  - 키들로 구성된 B-트리는 탐색, 삽입, 삭제 연산을 위해 관련된 이파리노드를 빠르게 찾을 수 있도록 안내해주는 역할만을 수행
  - B+-트리는 전체 레코드를 순차적으로 접근할 수 있도록 이파리들은 연결리스트로 구현



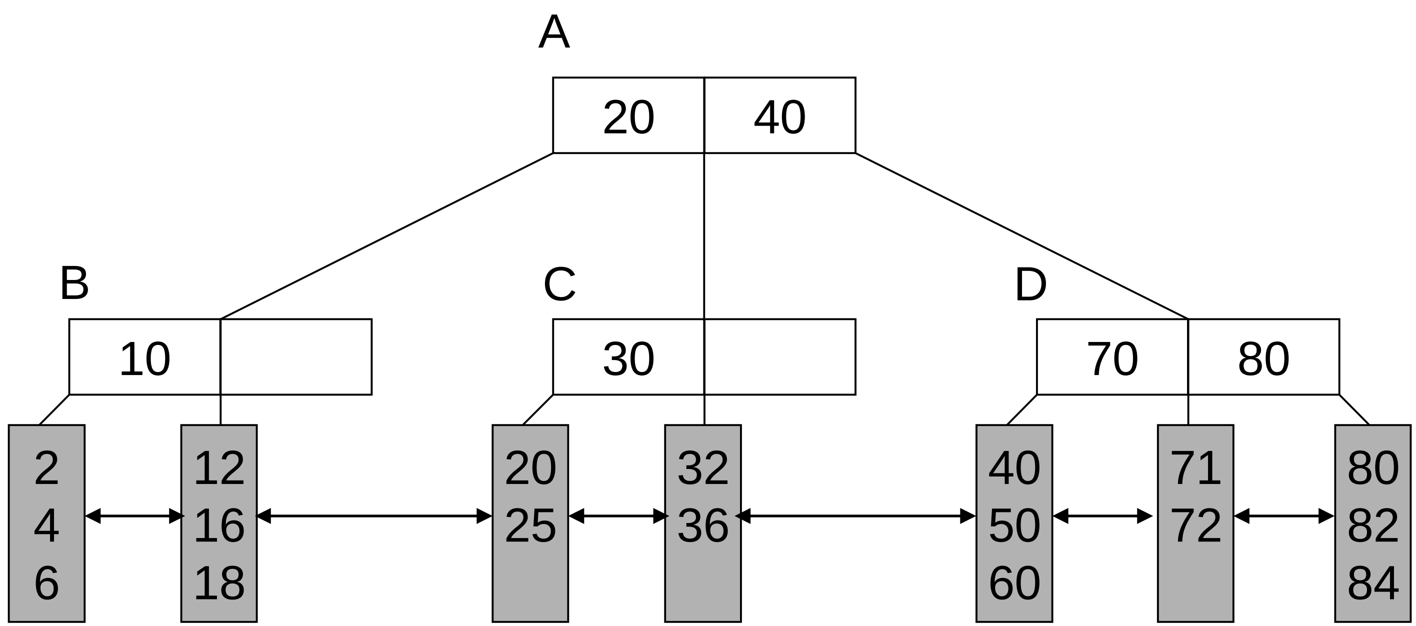
응용

- B-트리, B+-트리는 대용량의 데이터를 저장하고 유지하는 다양한 데이터베이스 시스템의 기본 자료구조로 활용
- Windows 운영체제의 파일 시스템인 HPFS(High Performance File System)
- 매킨토시 운영체제의 파일 시스템인 HFS(Hierarchical File System)과 HFS+
- 리눅스 운영체제의 파일 시스템인 ReiserFS, XFS, Ext3FS, JFS
- 상용 데이터베이스인 ORACLE, DB2, INGRES와 오픈소스 DBMS인 PostgreSQL에서 사용

B+-트리 (1)

- **B-트리와 비슷한 계통**
- **B-트리와의 차이점**
  - (1) 인덱스(index) 노드와 데이타(data)노드
    - 인덱스 노드 : B-트리에서의 내부 노드와 일치, 키와 포인터를 저장
    - 데이타 노드 : B-트리에서의 외부 노드와 일치, 키와 함께 원소를 저장
  - (2) 데이타 노드는 왼쪽에서 오른쪽 순서대로 서로 링크 되어 있고 이중 연결 리스트를 형성

B⁺-트리 (2) - 차수 3인 B⁺-트리의 예



- 데이터 노드 인덱스 노드들은 높이 2인 2-3 트리를 형성하고 있음
- 데이터 노드(회색) 크기와 인덱스 노드 크기는 똑같지 않아도 된다.
  - 데이터 노드 크기가 c일 때, 루트가 아닌 데이타 노드의 최소 원소 수는 $\lceil c/2 \rceil$

B+-트리 (3) - 정의

- **차수가 m인 B^+-트리(B+-tree of order m)**
  - 공백이거나 다음 성질들을 만족
    - (1)모든 데이타 노드는 같은 레벨에 위치해있고, 리프 노드이다.
      데이타 노드는 원소만 포함함.
    - (2)인덱스 노드는 차수가 m인 B-트리를 정의함.
      각 인덱스 노드는 키를 갖고 있지만 원소를 갖고 있지는 않다.
    - (3)인덱스 노드의 형식 : $n, A_i, (K_1, A_1), (K_2, A_2), \ldots, (K_n, A_n)$
      - $A_i (0 \leq i \leq n < m)$가 서브트리에 대한 포인터, $K_i (1 \leq i < n < m)$는 키임
      - $K_0 = -\infty$, $K_{n+1} = \infty$
      - 서브트리 A_i의 모든 원소는 $0 \leq i < n$일 때, K_{i+1}보다 작고 K_i보다 크거나 같은 키를 가진다.

B+-트리 (4) - 탐색

- **두 가지 종류의 탐색 지원**
  - 정확히 일치하는 값에 대한 검색
  - 범위 검색

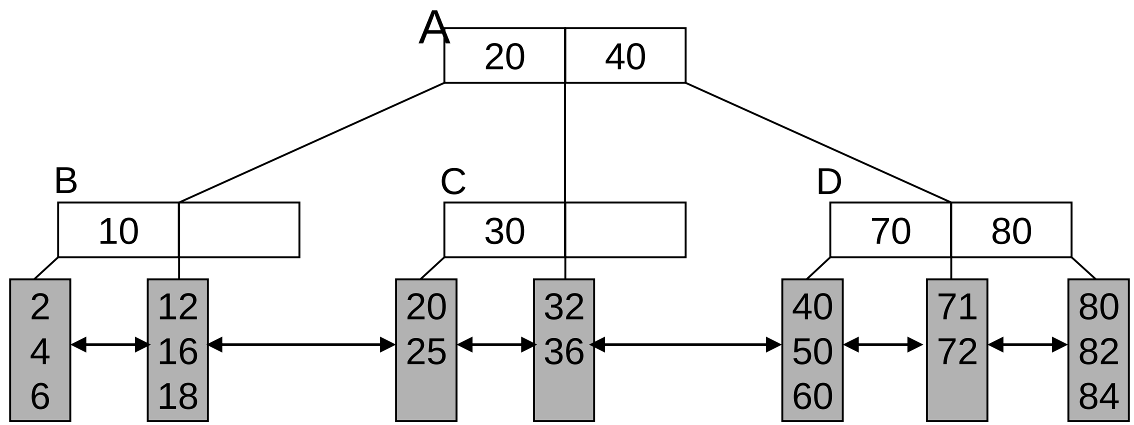
```
// B+-트리에서 x 키를 갖고 있는 원소를 탐색한다.
// 찾으면 원소를 반환한다. 그렇지 않으면 NULL을 반환한다.
if the tree is empty return NULL;
K0 = -MAXKEY;
for(*p = root; p is an index node; p = Ai)
{
        p가 다음과 같은 형식을 갖고 있다. : n, A0, (K1, A1), ...,(Kn,An);
        Kn+1 = MAXKEY;
        Ki <= x < Ki+1;
}
// p 데이타 노드를 탐색한다.
x인 키를 가지고 있는 원소 E에 대한 p를 탐색한다;
if 이러한 원소를 찾으면 E를 return
else return NULL;
```

B+-트리에서의 탐색 알고리즘

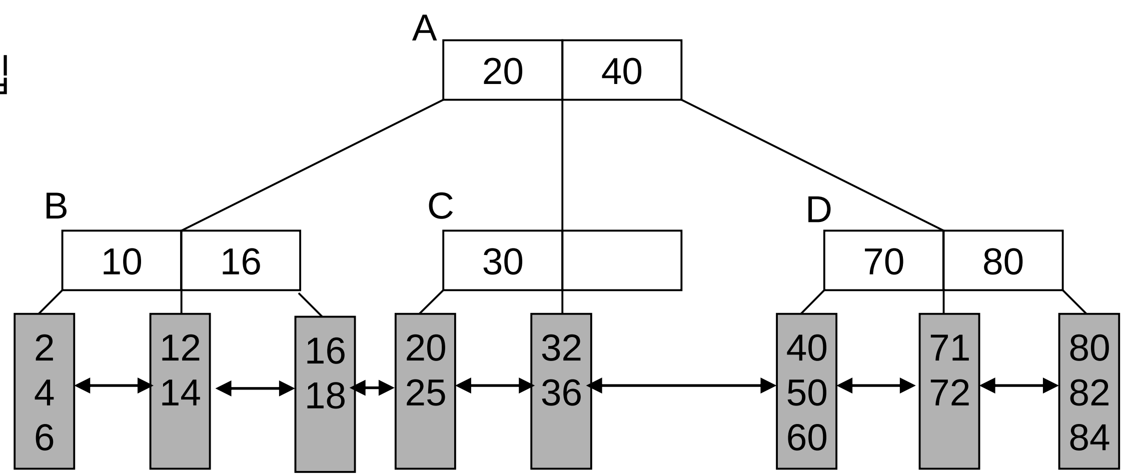
B+-트리 (5) - 삽입 (1)

- **B-트리에서의 삽입과는 분할된 데이타 노드를 처리하는 방법에 차이가 있음**
  - 데이타 노드가 완전히 차면 가장 큰 키들을 가지고 있는 원소의 절반을 새로운 노드로 옮김
  - 이 중 가장 작은 원소의 키를 새로 생성된 데이타 노드에 대한 포인터와 같이 B-트리 삽입 과정을 따라서 부모 인덱스 노드에 삽입
- **인덱스 노드 분할은 B-트리에서의 내부 노드 분할과 같음**

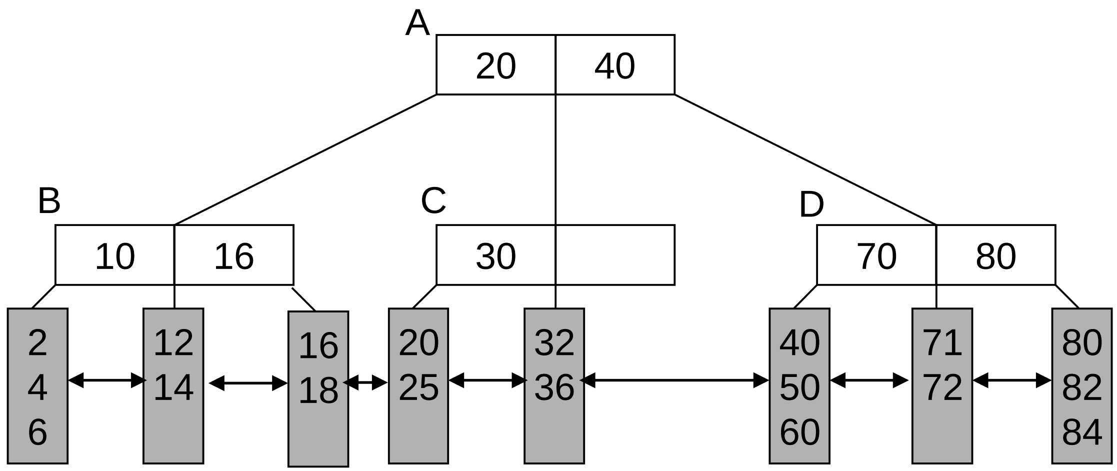
B⁺-트리 (5) - 삽입 (2)



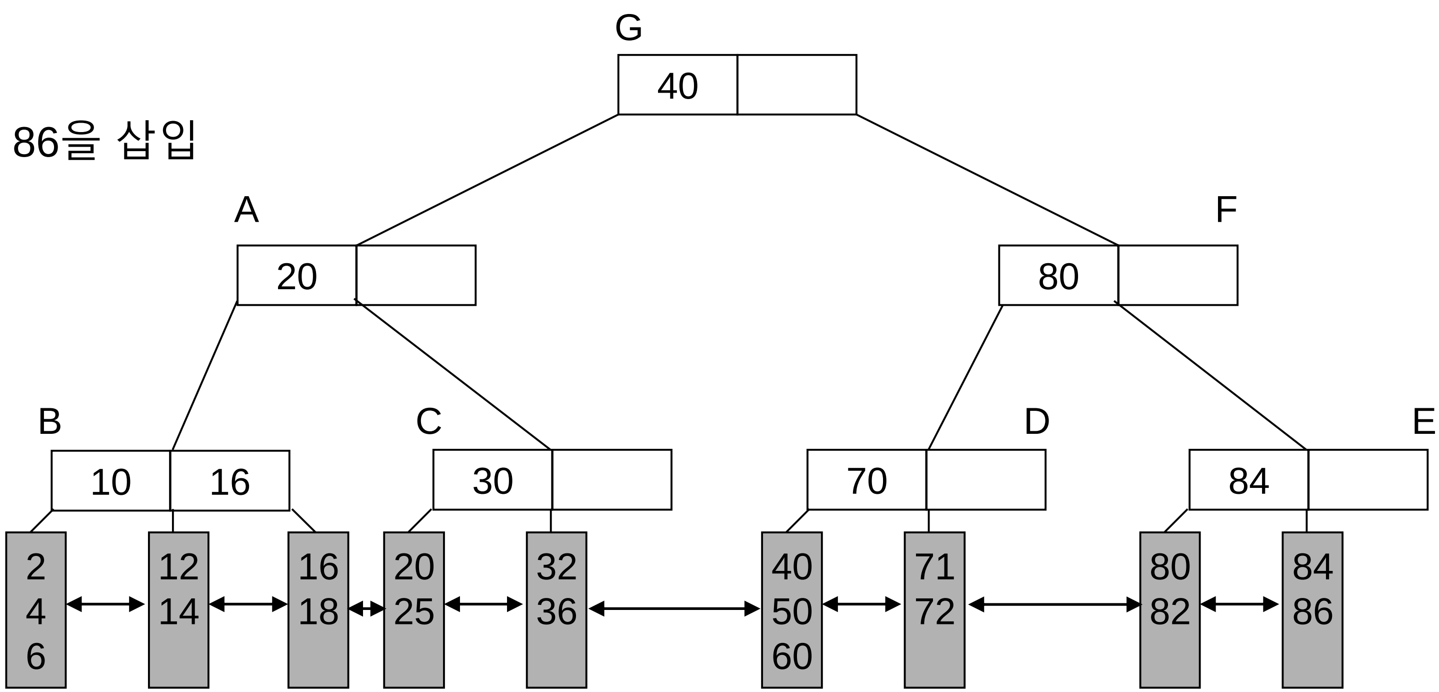
(a) 14를 삽입



B+-트리(6) - 삽입 (3)



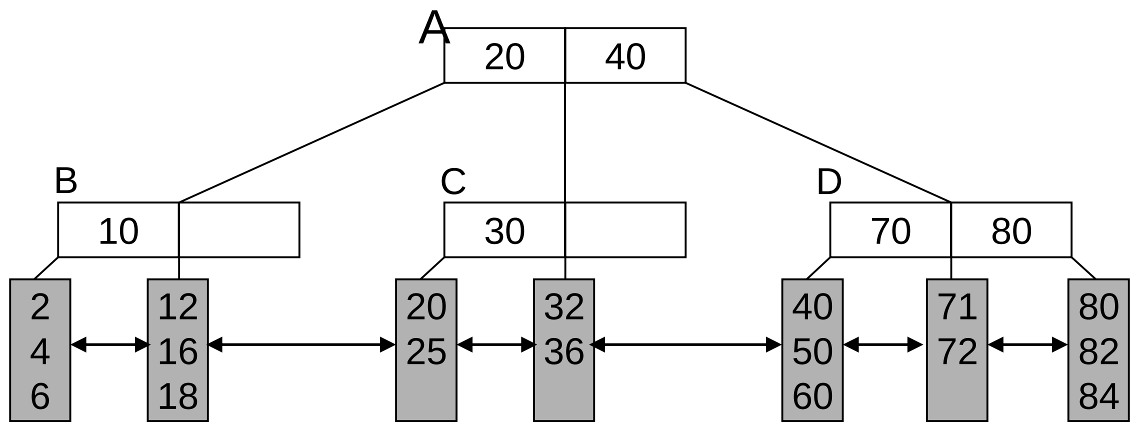
(b) 86을 삽입



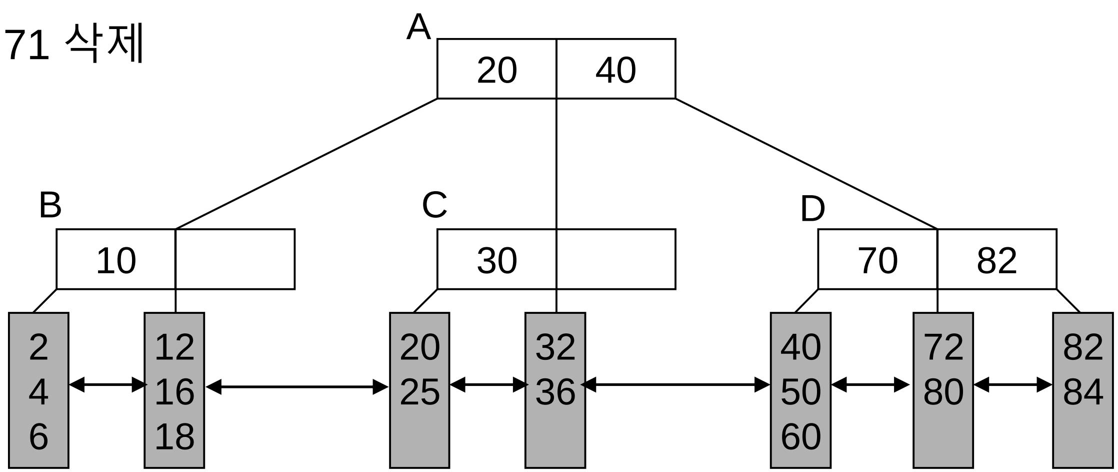
B+-트리 (7) - 삭제 (1)

- **원소들은 리프에만 저장 → ∴리프에서의 삭제만 주의**
- **최소 원소 수가 부족하게 되는 경우**
  - 인덱스 노드 : B-트리를 형성하므로 루트가 아닌 인덱스 노드는 ⌈m/2⌉ -1개보다 키가 적을 때, 루트 인덱스 노드는 키를 갖고 있지 않을 때
  - 데이타 노드 : 루트가 아닌 데이타 노드는 원소 수가 ⌈c/2⌉개보다 적을 때, 루트 노드는 공백일 때
    - (c : 데이타 노드가 가질 수 있는 원소 수)
- **원소를 삭제한 후**
  - 데이타 노드의 최소 원소 수가 부족하지 않은 경우
    - 변경된 데이타 노드는 디스크에 기록되고, 인덱스 노드는 변경되지 않음
  - 데이타 노드의 최소 원소 수가 부족한 경우
    - 최소 원소 수 보다 많은 원소를 보유한 가장 가까운 형제 노드에게 원소를 빌려오며 그에 따른 부모 노드(인덱스 노드)의 해당 키 값을 변경.

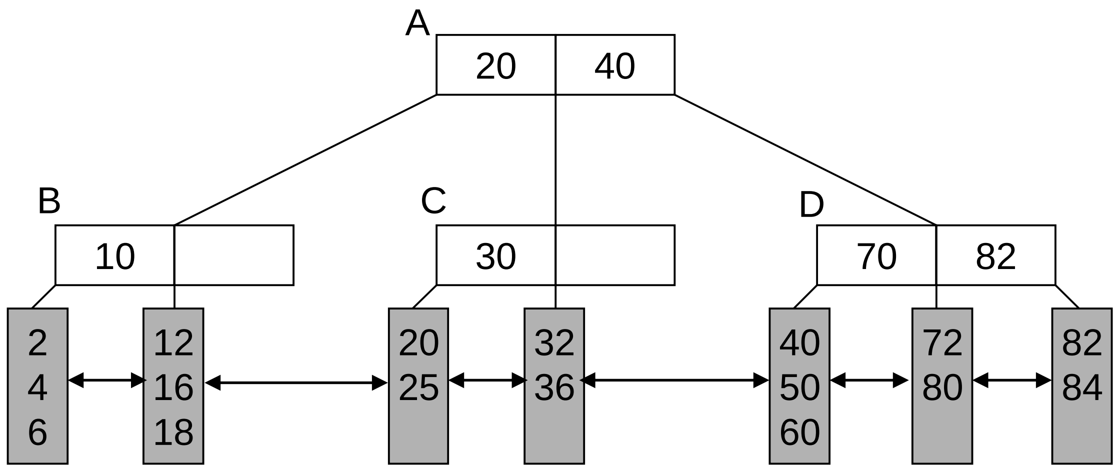
B+-트리 (8) - 삭제 (2)



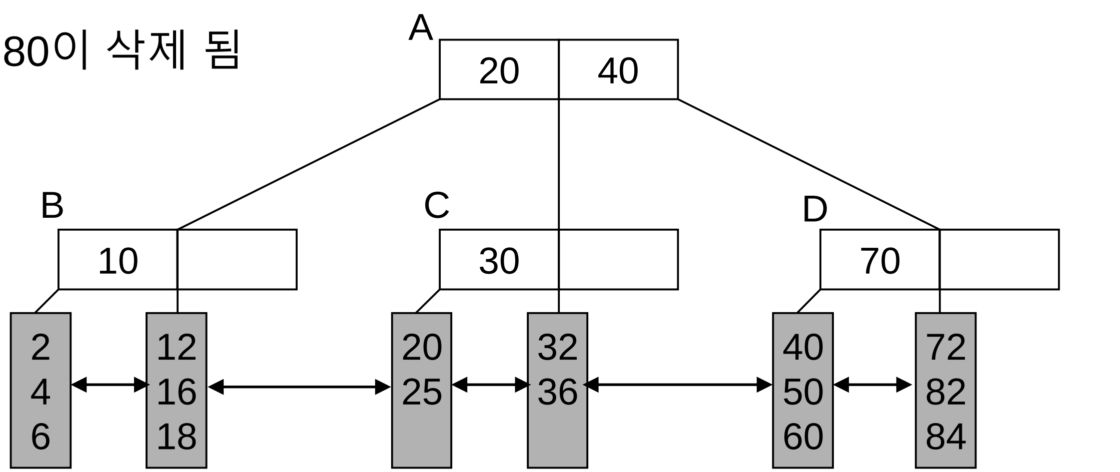
(a) 71 삭제



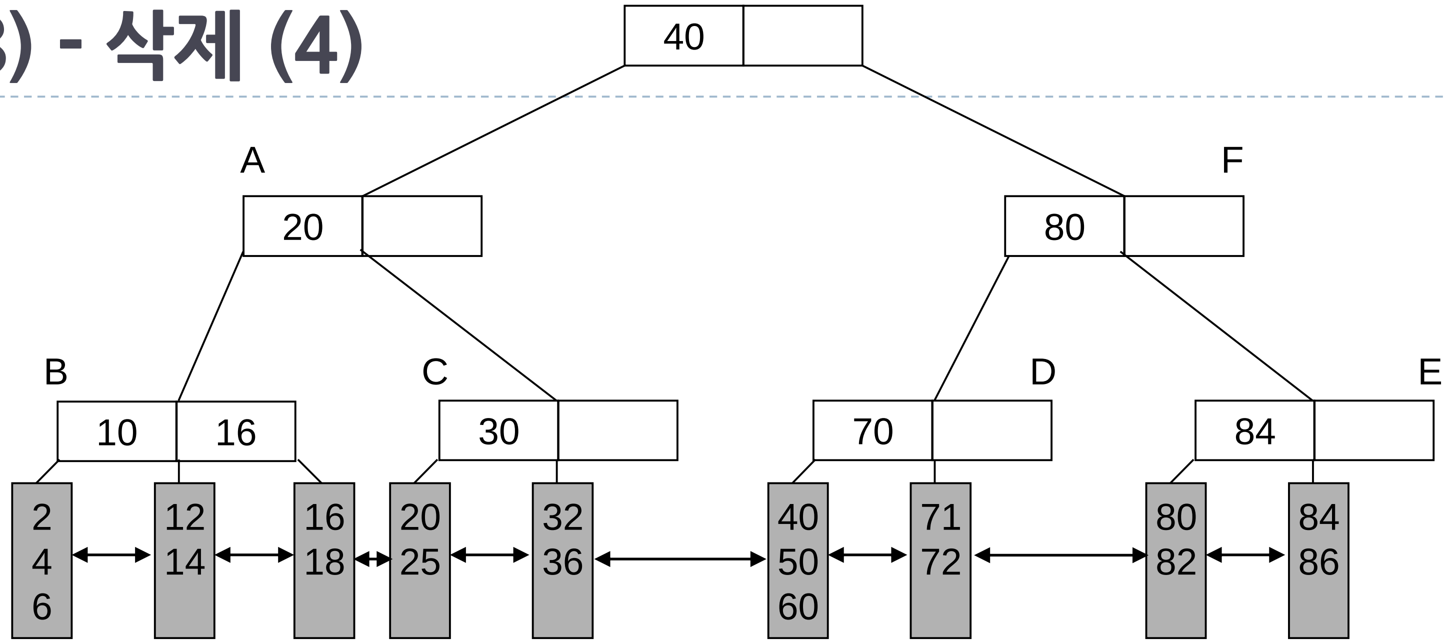
B+-트리 (8) - 삭제 (3)



(b) (a)에서 80이 삭제 됨

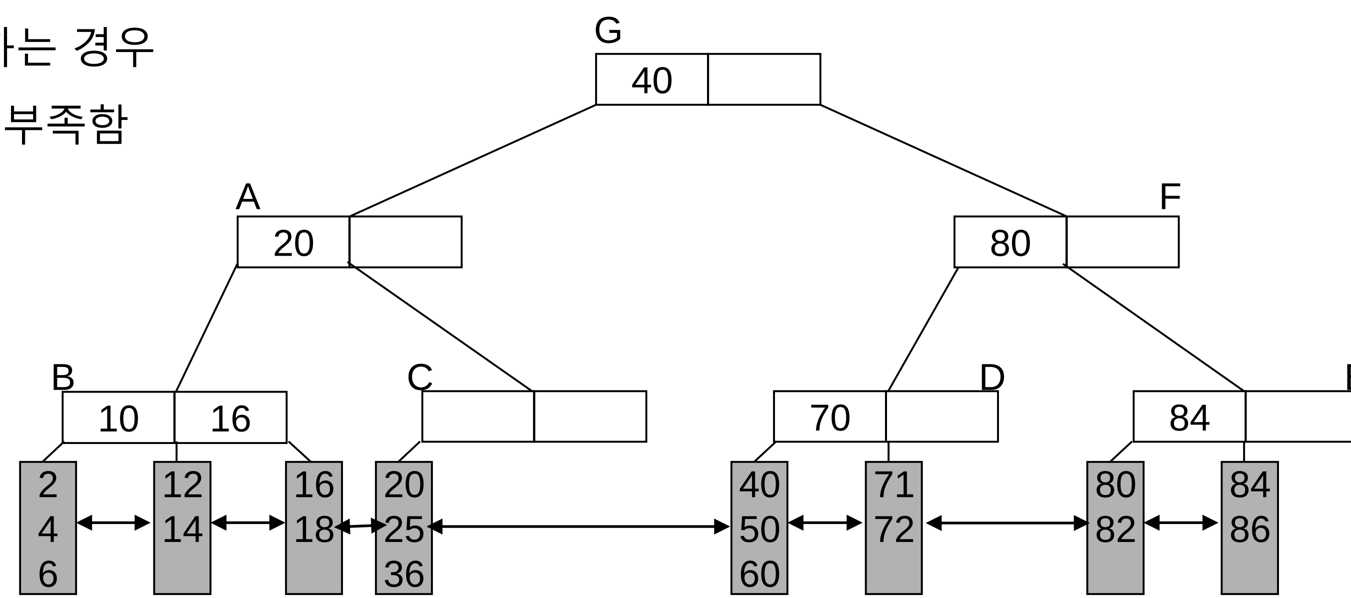


B+-트리 (8) - 삭제 (4)

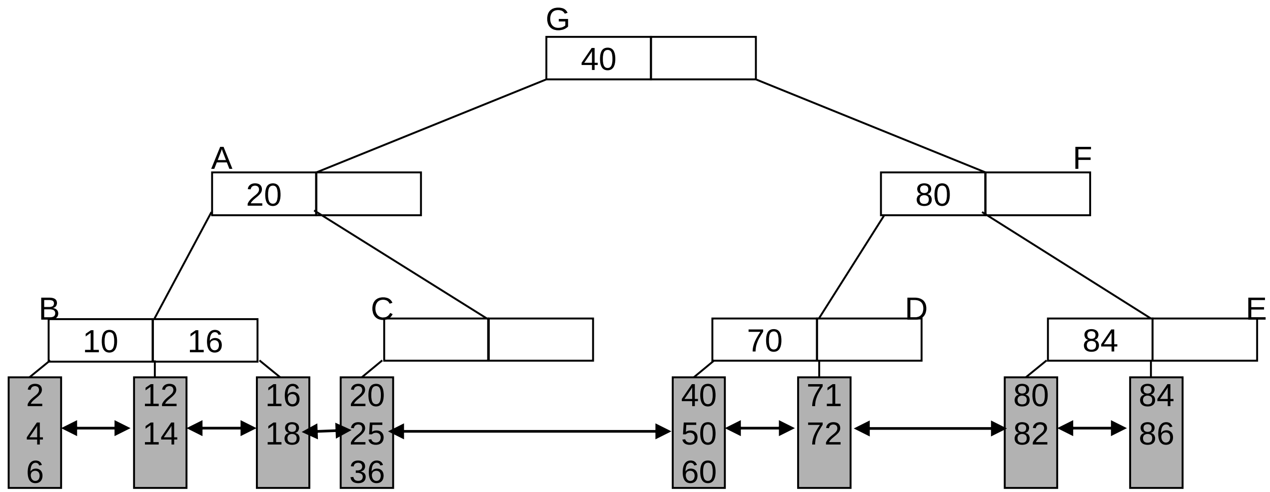


(a) 32를 삭제하는 경우

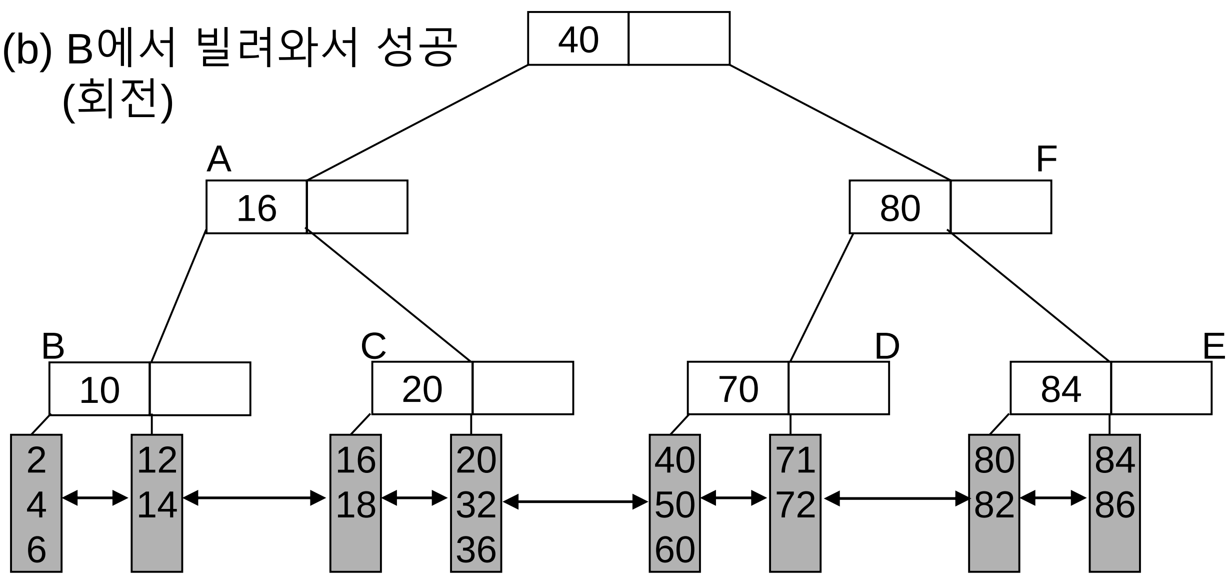
C의 원소가 부족함



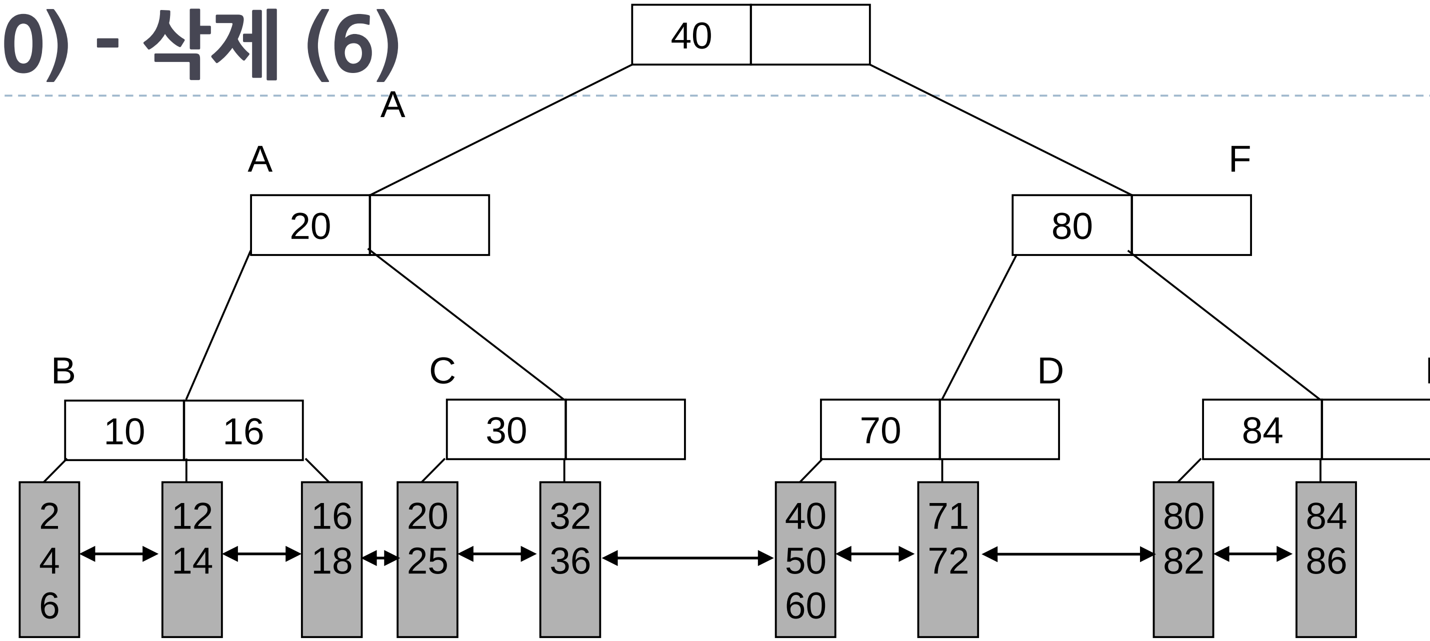
B⁺-트리 (9) - 삭제 (5)



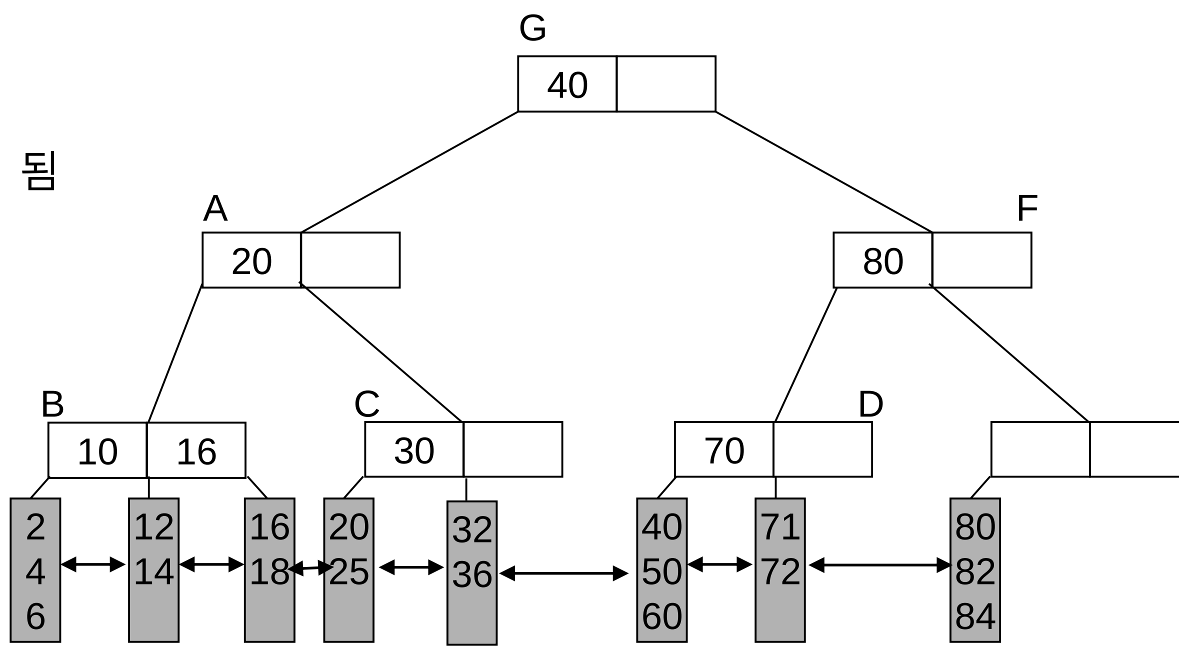
(b) B에서 빌려와서 성공 (회전)



B+-트리 (10) - 삭제 (6)

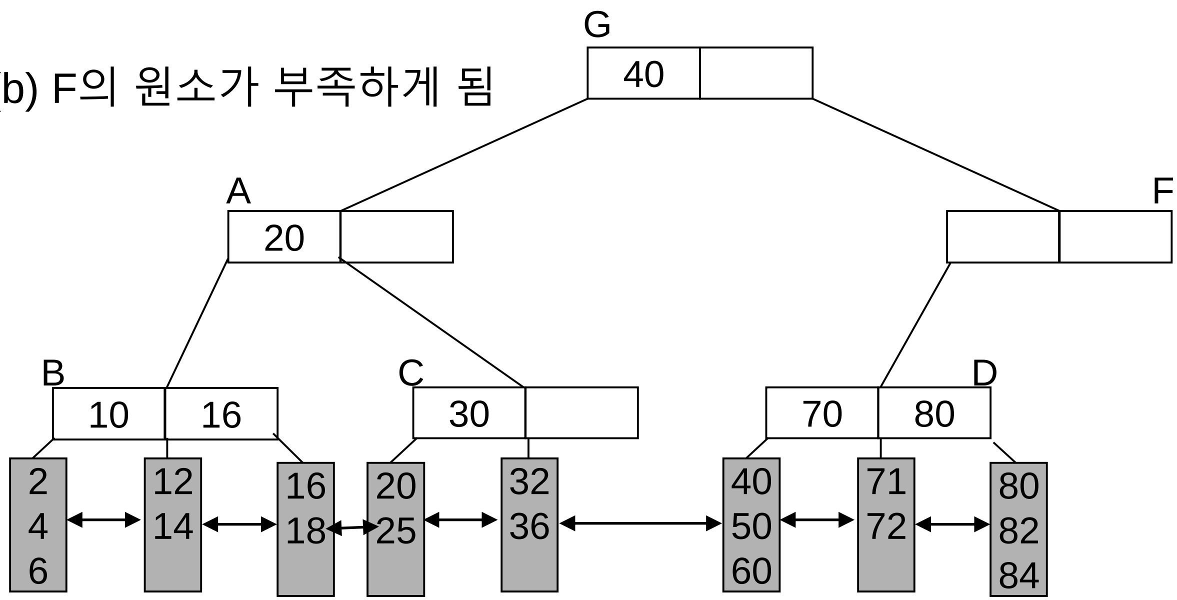


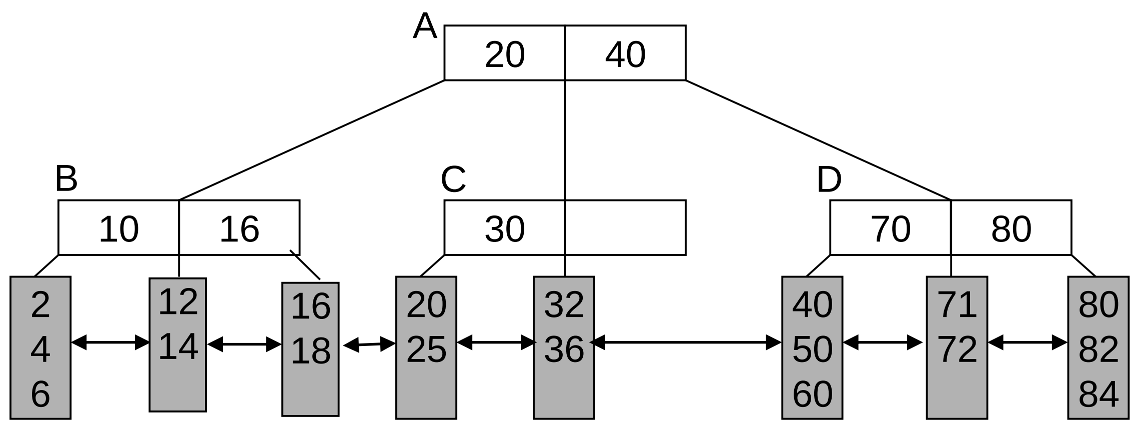
(a) 86을 삭제하는 경우
E의 원소가 부족하게 됨



B+-트리 (10) - 삭제 (7)

(b) F의 원소가 부족하게 됨





요약

- 2-3트리는 내부노드의 차수가 2∽ 3인 완전 균형탐색트리
- 2-3트리의 탐색, 삽입, 삭제 연산의 수행시간은 각각 트리의 높이에 비례하므로 O(logN)
- 2-3-4트리는 2-3트리를 확장한 트리로 4-노드까지 허용
- 2-3-4트리에서는 루트로부터 이파리노드로 한번만 내려가며 미리 분리 또는 통합 연산을 수행하는 효율적인 삽입 및 삭제가 가능
- 레드블랙트리는 노드의 색을 이용하여 트리의 균형을 유지하며, 탐색, 삽입, 삭제 연산의 수행시간이 각각 O(logN)을 넘지 않는 매우 효율적인 자료구조

요약

- N개의 노드를 가진 레드블랙트리의 높이 h는 2logN보다 크지 않다. 탐색, 삽입, 삭제의 수행시간은 O(logN)
- B-트리는 다수의 키를 가진 노드로 구성되어 다방향 탐색이 가능한 완전 균형트리
- B*-트리는 B-트리로서 루트를 제외한 다른 노드의 자식 수가 2/3M ~ M이어야 한다. B*-트리는 노드의 공간을 B-트리보다 효율적으로 활용하는 자료구조
- B+-트리는 키들만을 가지고 B-트리를 만들고, 이파리노드에 키와 관련 정보를 저장
- B-트리는 몇 개의 디스크 페이지(블록)를 메인 메모리로 읽어 들이는지가 더 중요하므로 한 개의 노드가 한 개의 디스크 페이지에 맞도록 차수 M을 정함