

1

B-트리

1 B-트리

- ◆ 이진 탐색 트리의 문제점
 - 좌우 균형이 맞지 않으면 비효율적임
- ◆ 균형 트리(Balanced Tree)
 - 삽입과 삭제시 필요하면 스스로 균형을 유지함
 - 레드 블랙 트리, B-트리, AVL Tree 등
 - 항상 $O(\log n)$ 의 검색 성능

1 B-트리

- ◆ 검색 트리가 방대하면 검색 트리를 메모리에 올려 놓고 사용할 수 없음
- ◆ 결국 검색 트리가 디스크에 있는 상태로 작업해야 하는데 이 경우 디스크 접근 횟수가 효율을 좌우함
- ◆ 디스크의 접근 단위는 블록(페이지)
- ◆ 디스크에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹음
- ◆ 검색 트리가 디스크에 저장되어 있다면 트리의 높이를 최소화하는 것이 유리함

1 B-트리

- ◆ 외부 검색 트리
 - 검색 트리가 디스크에 있는 상태로 사용됨
 - ◆ 다진 검색 트리
 - 분기의 수가 2개를 넘음
- B-트리는 디스크 환경에서 사용하기에 적합한 외부 다진 검색 트리이며
다진 검색 트리가 균형을 유지하도록 하여
최악의 경우 디스크 접근 횟수를 줄인 것

1 B-트리

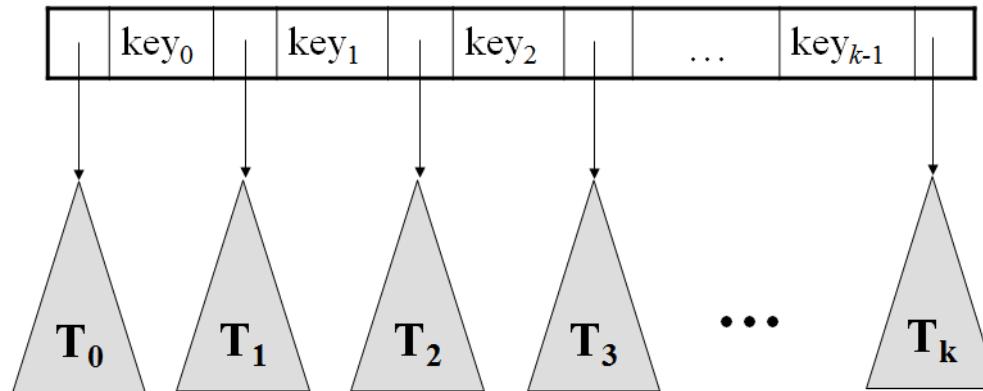
- ◆ 이진 트리를 확장해 하나의 노드가 가질 수 있는
자식 노드의 최대 숫자가 2보다 큰 트리 구조
- ◆ 하나의 노드에 많은 수의 데이터가 배치
- ◆ 대용량의 파일을 효율적으로 검색하고
갱신하기 위해 고안된 자료 구조
- ◆ 한 노드에 M개의 자료가 배치되면 M차 B-트리
- ◆ B-트리는 자료를 정렬된 상태로 보관하고,
삽입 및 삭제를 대수 시간으로 할 수 있음

1 B-트리

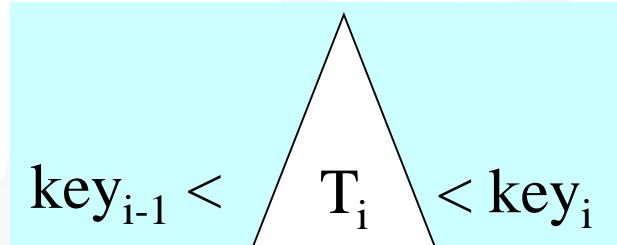
- ◆ B-트리는 스스로 균형을 맞추는 트리
- ◆ 이진 탐색 트리를 일반화한 것으로 볼 수 있음
- ◆ 최악의 경우에도 $O(\log n)$ 의 검색 성능을 보임

2 / 다진 검색 트리

- ◆ 키가 k 개 있으면 $k+1$ 개의 자식을 가짐
- ◆ 각각 대응되는 서브 트리는 T_0, T_1, \dots, T_k



[다진 검색 트리에 키가 k 개 있는 예]



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

2 다진 검색 트리

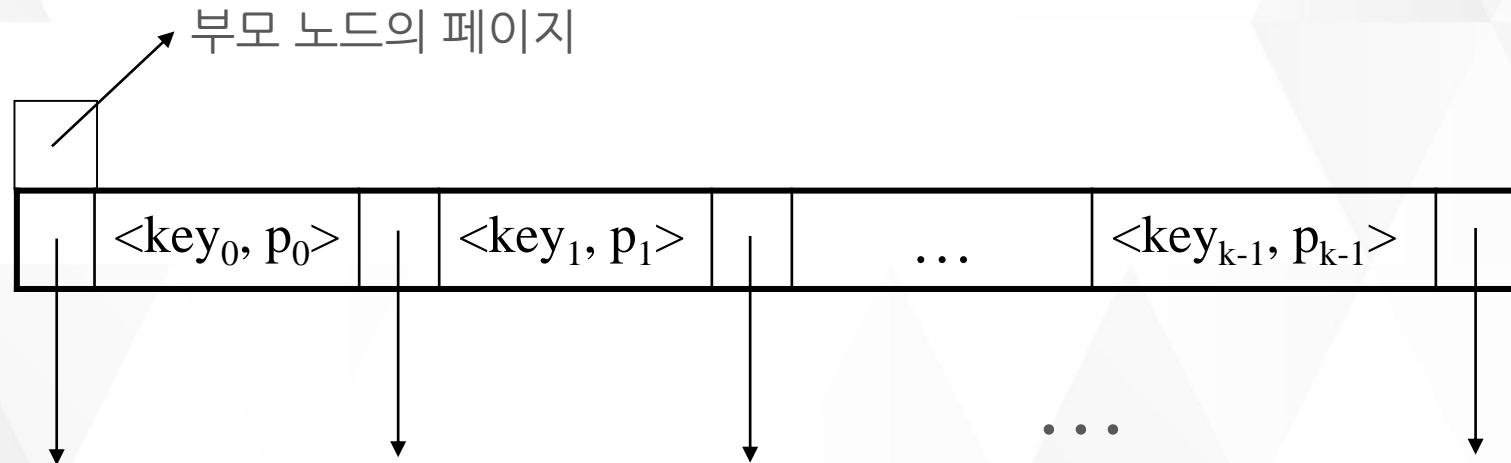
◆ B-트리는 균형 잡힌 다진 검색 트리로
다음의 성질을 만족함

- 루트를 제외한 모든 노드는 $\lfloor k/2 \rfloor \sim k$ 개의 키를 갖음
- 모든 리프 노드는 같은 깊이를 가짐

- B-트리는 분기의 수를 가능하면 늘리되 균형을
맞추기 위해 각 노드가 채울 수 있는 최대 허용량의
반 이상의 키는 채워야 하는 검색 트리

3 B-트리의 노드 구조

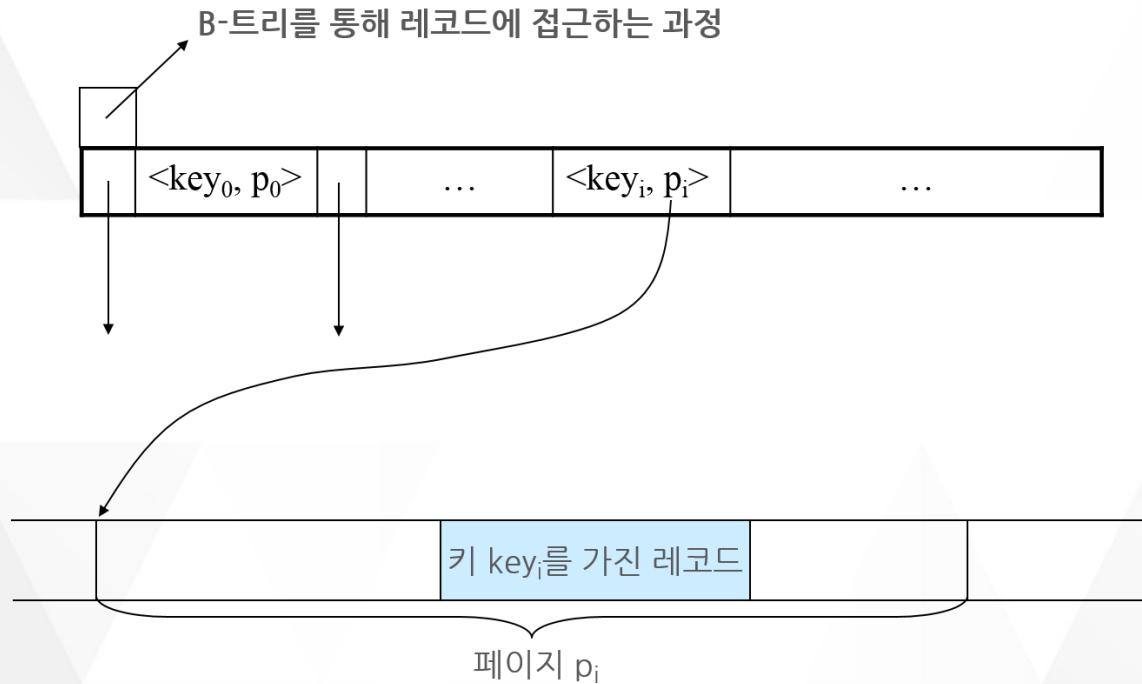
- 자식 노드로의 포인터, 부모 노드의 페이지, p_i 들은 모두 페이지 번호로 나타냄



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

4

레코드에 접근하는 절차를 보여주는 구조



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

5 B-트리에서의 탐색

- ▶ B-트리에서의 탐색은 이진 탐색 트리에서 탐색과 같음
- ▶ 이진 탐색 트리에서는 각 노드에 키가 하나밖에 없지만 B-트리는 최대 k개까지 가질 수 있음
- ▶ B-트리에서는 노드의 여러 키 중 탐색 키와 일치하는 것이 있는지 확인함

2

B-트리에서 삽입

1

B-트리에서 삽입

- ◆ B-트리에서 키 x 를 삽입하는 과정
 - ① x 를 삽입할 리프 노드 r 을 찾음
 - ② 노드 r 에 공간의 여유가 있으면 키를 삽입하고 끝냄
 - ③ 노드 r 에 여유가 없으면 형제 노드를 살펴 공간의 여유가 있으면 형제 노드에 키를 하나 넘기고 끝냄
 - ④ 형제 노드에 여유가 없으면 가운데 키를 부모 노드로 넘기고 노드를 두개로 분리하며 분리 작업은 부모 노드에서의 삽입 작업을 포함함

2 B-트리에서 삽입 알고리즘

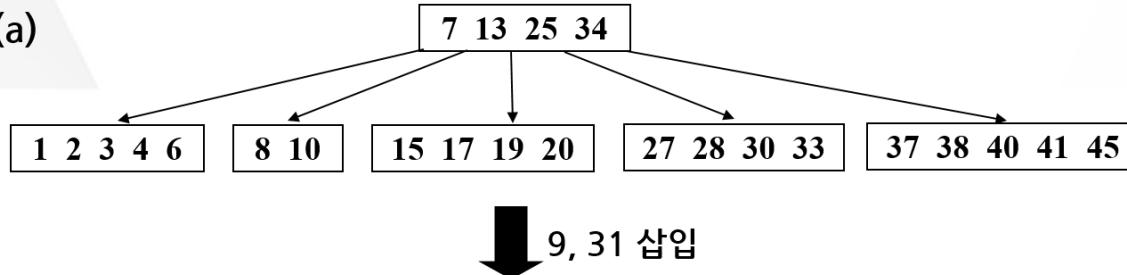
```
BTreeInsert(t, x)
{
    x를 삽입할 리프 노드 r을 찾는다;
    x를 r에 삽입한다;
    if (r에 오버플로우 발생) then clearOverflow(r);
}
clearOverflow(r)
{
    if (r의 형제 노드 중 여유가 있는 노드가 있음) then
        {r의 남는 키를 넘긴다;};
    else {
        r을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
        if (부모 노드 p에 오버플로우 발생) then clearOverflow(p);
    }
}
```

- ✓ t : 트리의 루트 노드
- ✓ x : 삽입하고자 하는 키

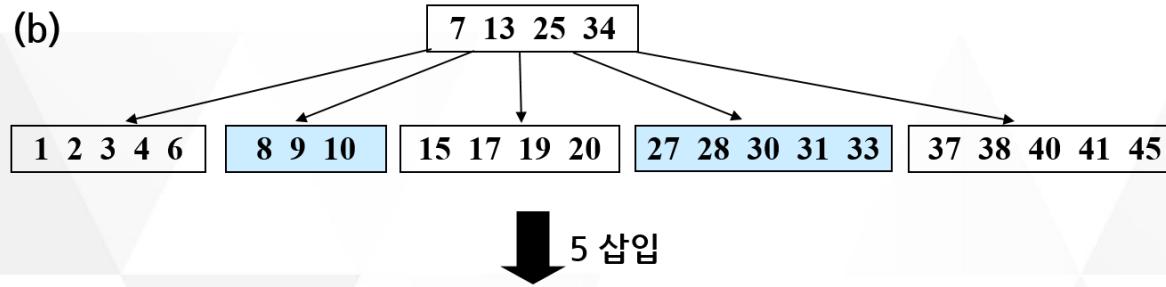
3 B-트리에서 삽입 예

- 각 노드가 최대 5개의 키를 가질 수 있다고 가정함

(a)



(b)



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

3 B-트리에서 삽입 예

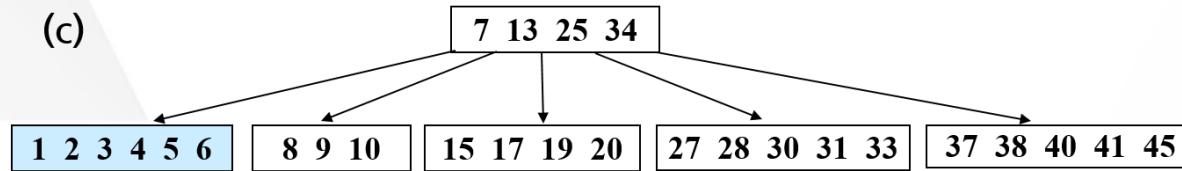
◆ 5 삽입

- 5를 해당 리프 노드에 삽입하면
오버플로우가 발생함
 - 오른쪽 형제 노드에 공간의 여유가 있으므로
키를 하나 넘김
 - 맨 오른쪽 6을 형제 노드에 바로 넘기면
검색 트리의 성질이 깨지므로 부모 노드에 있는
7을 넘기고 그 자리에 6을 놓음
- 이것을 재분배라고 함

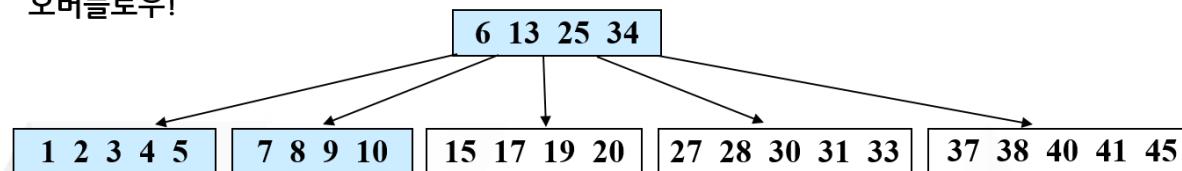
B-트리에서 삽입

3 B-트리에서 삽입 예

(c)



오버플로우!



39 삽입

※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

B-트리에서 삽입

3 B-트리에서 삽입 예

◆ 39 삽입

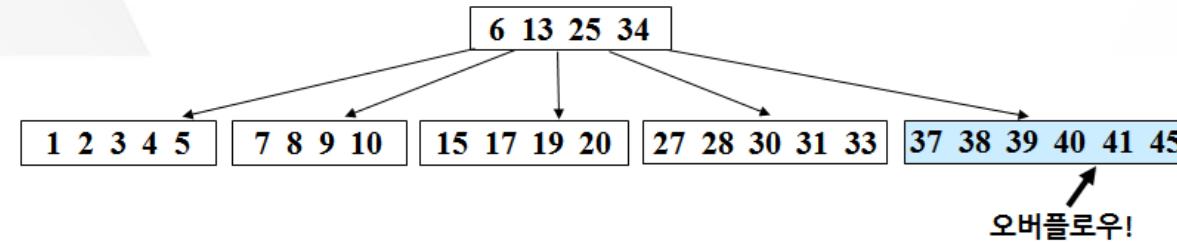
- 39를 해당 리프 노드에 삽입하면 오버플로우가 발생함
- 바로 옆 형제 노드에 공간의 여유가 없으므로 분할해야 함
- 맨 가운데 키 40을 부모 노드로 넘기고 나머지는 두개로 분할함

B-트리에서 삽입

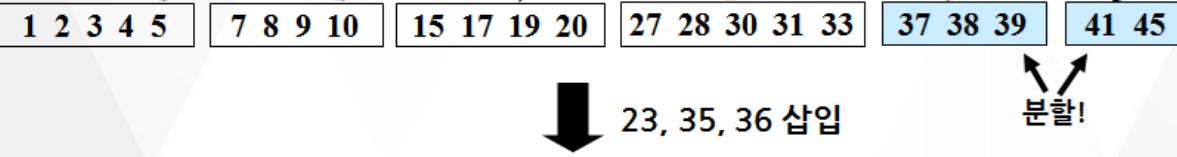
3 B-트리에서 삽입 예

(d)

39 삽입



6 13 25 34 40

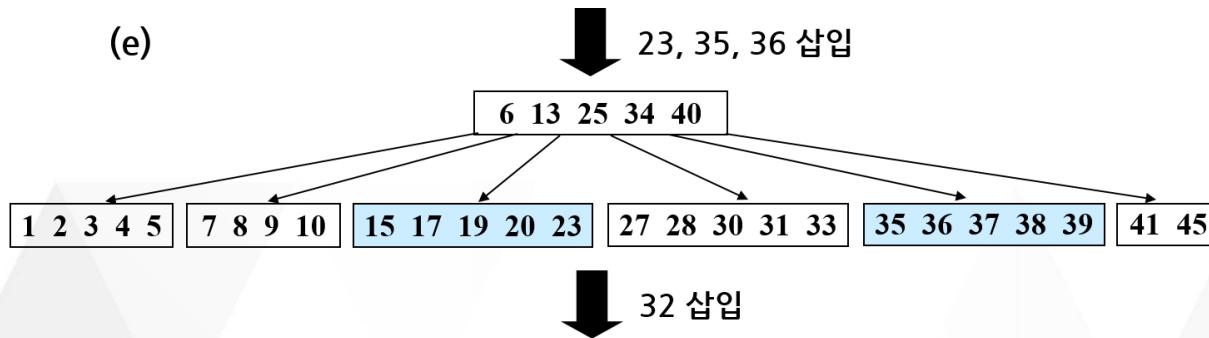


※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

3 B-트리에서 삽입 예

◆ 23, 35, 36 삽입

- 23, 35, 36 은 해당 리프 노드가 여유를 갖고 있으므로 해당 리프 노드에 삽입



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

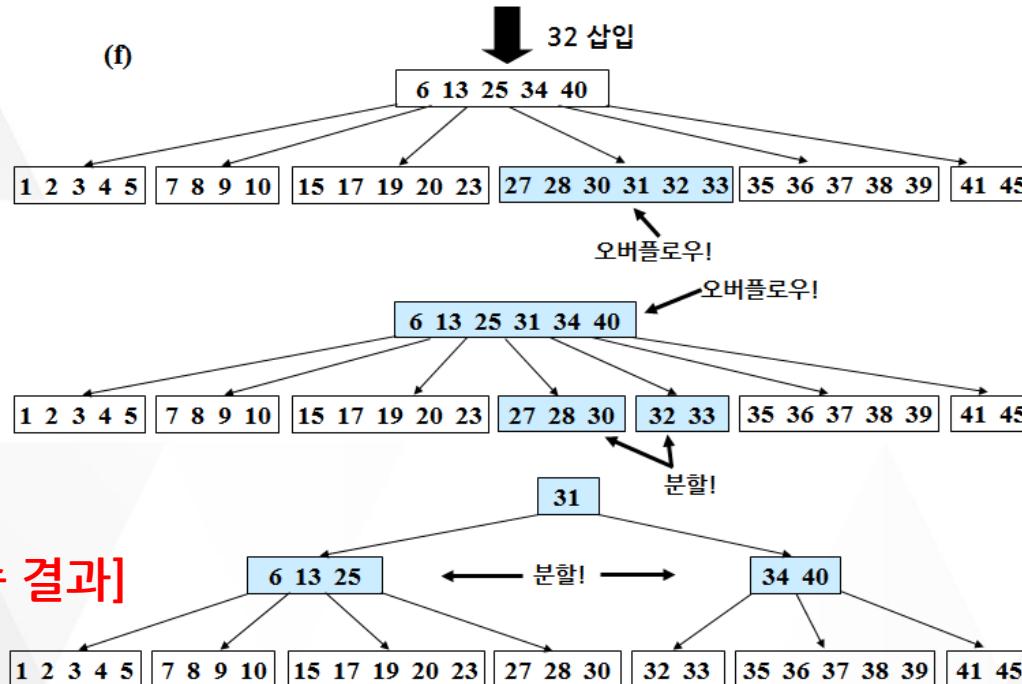
3 B-트리에서 삽입 예

◆ 32 삽입

- 32를 해당 리프 노드에 삽입하면 오버플로우가 발생
- 양쪽 형제 노드 모두 공간의 여유가 없으므로 분할함
- 가운데 키인 31을 부모 노드로 넘기고 노드를 둘로 분할함
- 키를 넘겨받은 부모 노드에 다시 오버플로우가 발생
- 바로 아래에 생겼던 오버플로우와 같은 상황이므로 재귀적으로 처리
- 이 노드가 루트 노드이므로 키를 추가로 받을 형제 노드가 없음
- 노드를 둘로 나누고 가운데 키 31로 새로운 노드를 만들어 이것이 새 루트 노드가 됨

B-트리에서 삽입

3 B-트리에서 삽입 예



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

3

B-트리에서 삭제

B-트리에서 삭제

1

B-트리에서 삭제

- ◆ B-트리에서 키 x 를 삭제하는 과정
 - ① x 를 키로 갖고 있는 노드를 찾음
 - ② 이 노드가 리프 노드가 아니면 x 의 직후 원소 y 를 가진 리프 노드 r 을 찾아 x 와 y 를 바꿈
 - ③ 리프 노드에서 x 를 제거
 - ④ x 제거 후 노드에 언더플로우가 발생하면 적절히 해소함

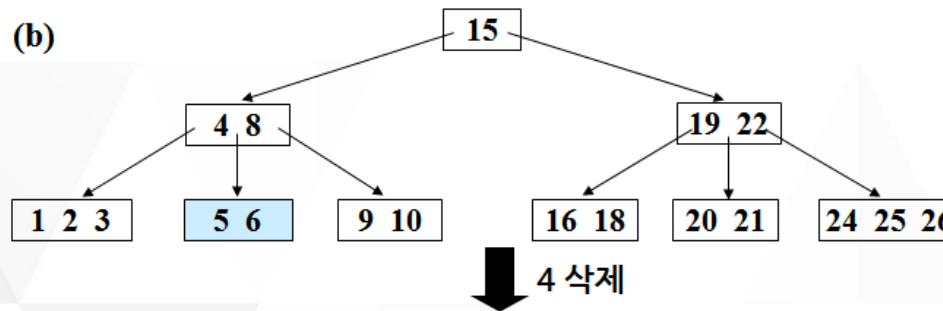
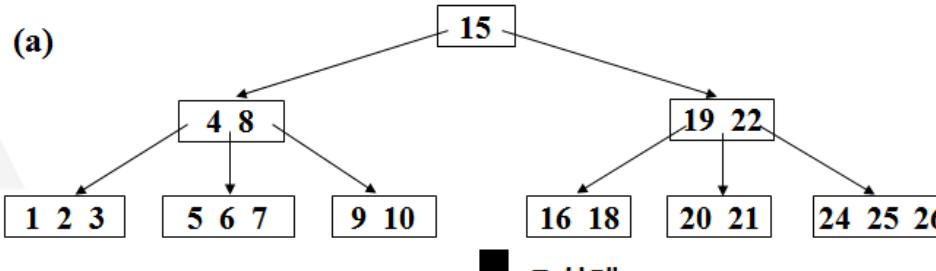
B-트리에서 삭제

1 B-트리에서 삭제

- ◆ B-트리에서 키 x 를 삭제하는 과정
 - 언더플로우가 발생할 때는 우선 키를 가져올 수 있는 형제 노드가 있는지 확인
 - 그런 노드가 있으면 가져다 채우고 끝냄
 - 그렇지 않으면 형제 노드와 병합해야 함
 - 병합은 두 노드를 하나로 합치는 것으로 부모 노드의 키 중 하나가 필요 없음
 - 이 필요 없는 키와 두 노드를 합쳐 하나의 노드로 만듬
 - 이 병합의 결과키가 하나 줄어든 부모 노드에서 언더플로우가 발생하는 경우 재귀적으로 처리함

B-트리에서 삭제

2 B-트리에서 삭제 예



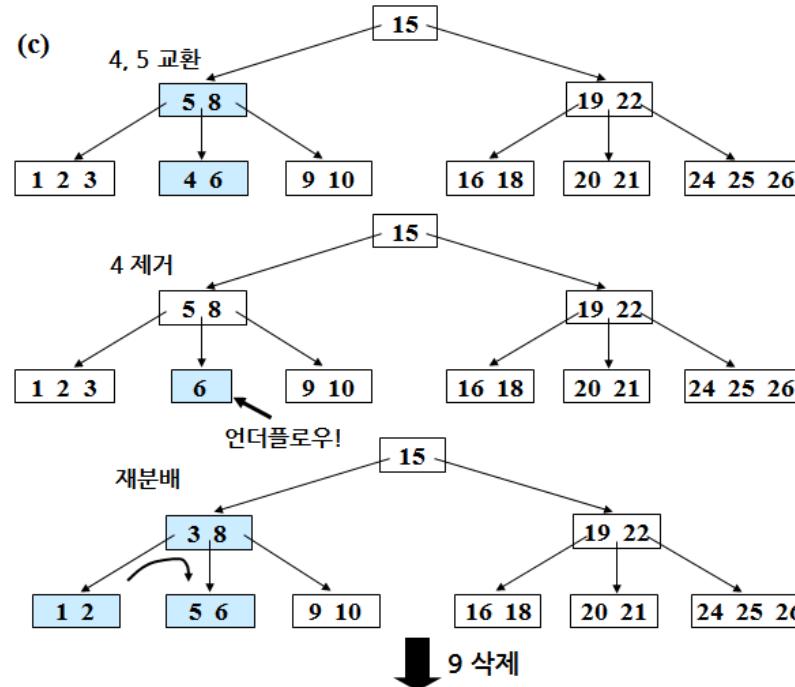
※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

2 B-트리에서 삭제 예

◆ 4 삭제

- 4가 리프 노드가 아니므로 4의 바로 다음 원소를 갖고 있는 리프인 5를 찾음
- 4와 5를 교환함
- 4를 제거하면 해당 리프에서 6 하나만 남아 언더플로우가 발생했음
- 형제 노드 중 키를 내놓을 수 있는 여분이 있는 노드를 확인
- 왼쪽 노드가 여분을 갖고 있으므로 3을 가져옴
- 키 3을 바로 6 옆에 놓을 수 없으므로 부모 노드에 있는 5를 끌어내리고 그 자리에 3을 놓음

2 B-트리에서 삭제 예



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

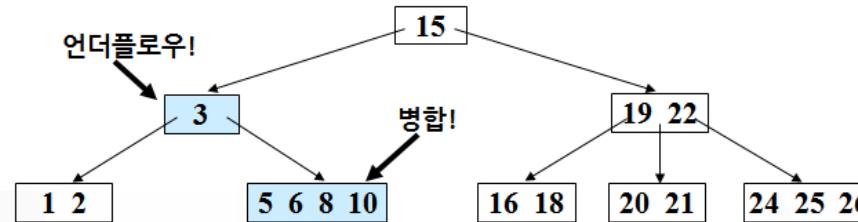
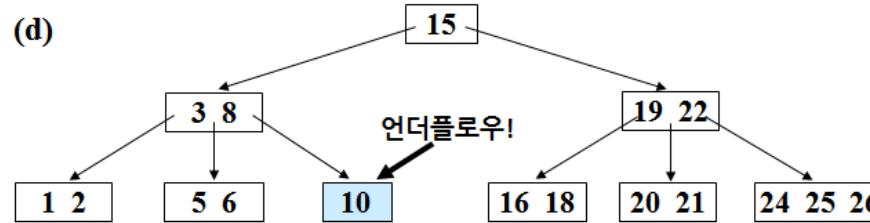
2 B-트리에서 삭제 예

◆ 9 삭제

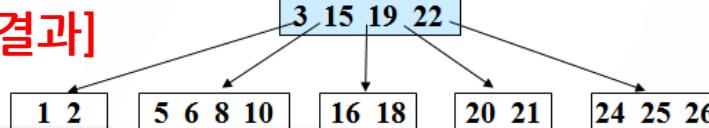
- 9를 리프 노드에서 삭제했더니 언더플로우가 생김
- 바로 옆의 형제 노드가 키를 내놓을 여유가 없으므로 병합함
- 병합은 형제 노드의 내용과 부모 노드의 키 하나(8)을 내려 받아 수행
- 병합에 키 8을 내준 부모 노드에 다시 언더플로우가 발생했으므로 재귀적으로 처리

B-트리에서 삭제

2 B-트리에서 삭제 예



[최종 결과]



※ 출처 : 쉽게 배우는 알고리즘, 문병로, 한빛아카데미

3 B-트리에서 삭제 알고리즘

```

BTreeDelete(t, x, v)
{
    if (v가 리프 노드 아님) then {
        x의 직후원소 y를 가진 리프 노드를 찾는다;
        x와 y를 맞바꾼다;
    }
    리프 노드에서 x를 제거하고 이 리프 노드를 r이라 한다;
    if (r에서 언더플로우 발생) then clearUnderflow(r);
}
clearUnderflow(r)
{
    if (r의 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)
        then { r이 키를 넘겨받는다;}
    else {
        r의 형제 노드와 r을 합병한다;
        if (부모 노드 p에 언더플로우 발생) then clearUnderflow(p);
    }
}

```

✓ t : 트리의 루트 노드
 ✓ x : 삭제하고자 하는 키
 ✓ v : x를 갖고 있는 노드