# B-트리

BST의 균형을 AVL-Tree를 통해서 맞추었다.

MST는 B-Tree를 통해 균형을 맞추면서 삽입/삭제를 할 수 있다.

B-Tree = Balanced-Tree ( 균형트리 )

#### < B-Tree 간단한 정의 >

* m = 분기의 개수
* n = 현재 존재하는 key 수
* 최솟값 = (m/2)-1 ... 분기 개수의 반 보다 1 작음
* 최댓값 = m-1 ... 분기 개수보다 1작음

#### < B-Tree의 정의 >

* 공백이거나 높이가 1 이상인 MST이다.
* root와 leaf를 제외한 내부 노드 : m/2 ~ m개의 서브트리
* 키 : m/2-1 ~ m-1
* leaf가 아니면 2개 이상의 서브트리를 갖는다.
* 모든 leaf는 같은 level에 있다.

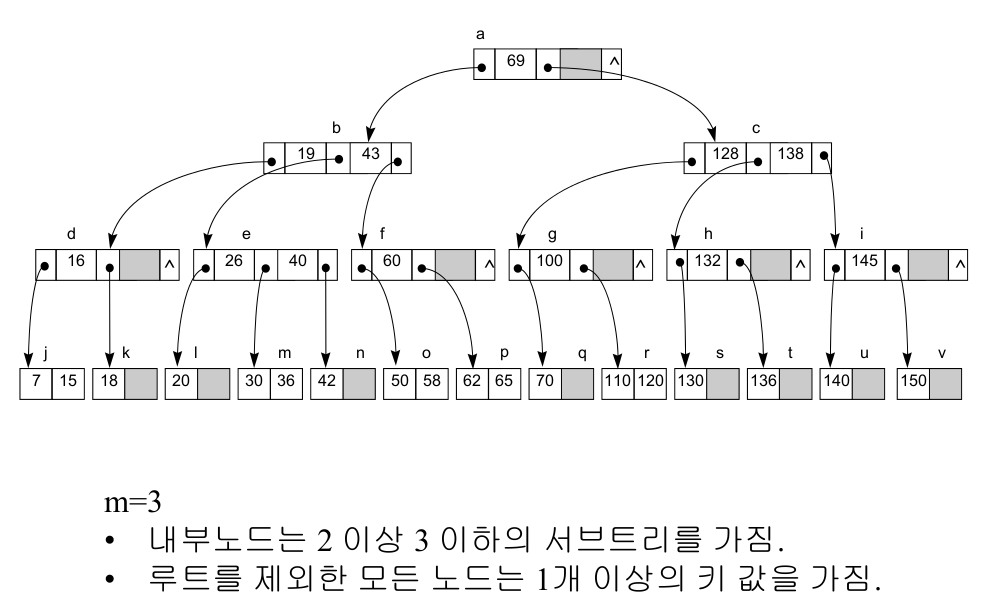
#### < B-Tree의 Node 구조 >

* **한 노드의 키 값 수** : ( m/2-1 <u><</u> n <u><</u> m-1)
* **서브트리 포인터** : Pi( 0 <u><</u> i <u><</u> n )
* **키 값** : Ki ( 1 <u><</u> i <u><</u> n )
* **키 값으로 Ki를 가진 레코드에 대한 포인터** : Ai ( 1 <u><</u> i <u><</u> n )

#### < B-Tree의 장점 >

* **최악의 경우** : O(logn(N + 1)) 접근
* 삽입 / 삭제시 균형 상태를 유지함 ... 재균형이 필요 없음

#### < B-Tree 예제 >



#### < B-Tree 검색 >

* MSB 처럼 검색한다. ( 좌소우대 ... 왼쪽이 작고 오른쪽이 큰 ... )
* 한 노드내의 Key 검색은 순차 검색이다.
* Inorder Traversal 로 검색한다.

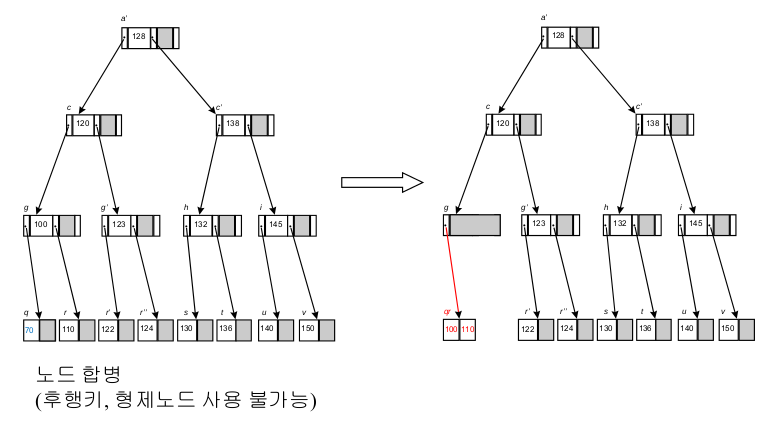
#### < B-Tree 삽입 > \*\* 실습 해보기

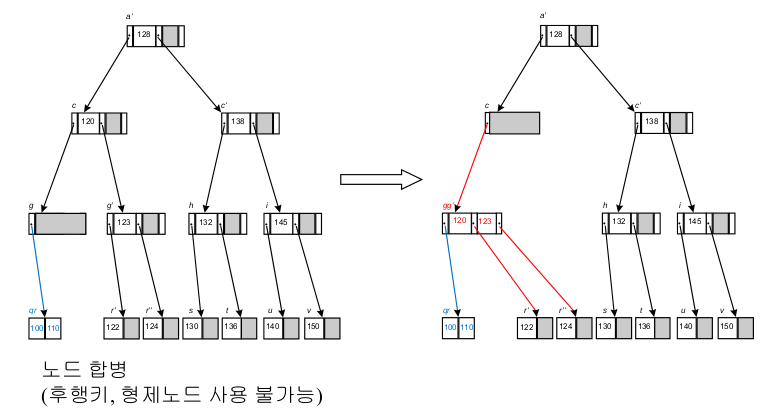
* 새로운 키값은 항상 리프 노드에 삽입된다.
* 삽입하는 두가지 케이스
  + 빈 공간이 있는 경우
    - 단순히 순서에 맞게 삽입
  + 빈 공간이 없는 경우 ( overflow 발생 )
    - 두 개의 노드로 분할한다.
    - 중간 키 값 ( m/2 번째 키 )를 기준으로 [왼쪽 키들] / [중간 키] / [오른쪽 키들] 로 나눈다.
    - 중간 키는 **분할된 노드**의 부모 노드로 이동하여 삽입되며, 왼쪽 오른쪽 키들을 가르키는 포인터도 같이 이동한다.
    - 만약 부모 노드로 올라간 키가 다시 오버플로가 난다면 다시 반복 수행한다.

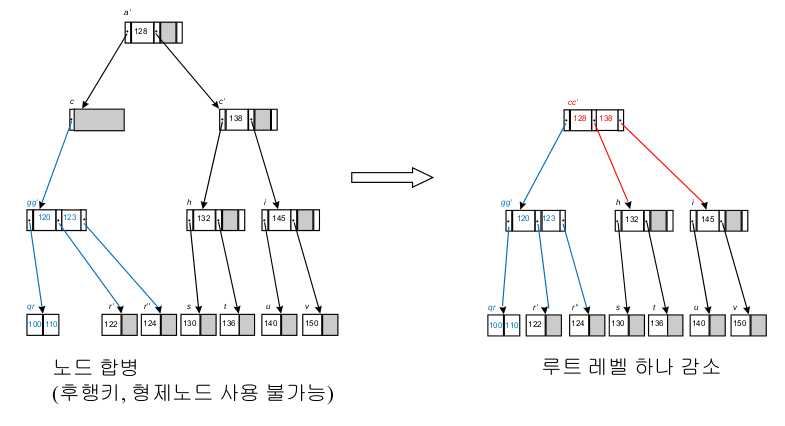
#### < B-Tree 삭제 > \*\* 실습 해보기

* 키 삭제에는 세가지 방법이 있다.
  + 키가 내부노드에 있는 경우
    - 후행키 방식
      * 후행키와 바꾼뒤 삭제한다.
  + Underflow가 발생하지 않는 경우
    - 평범하게 key를 하나 삭제한다.
  + Underflow가 발생한 경우
    - 키 재분배 ( Key Redistribution ) : 트리 구조를 변경하지 않는다.
      1. 왼쪽이나 오른쪽 형제 노드에 최소 키 값 수보다 많은 키 값들이 있는 노드를 선택
      2. 그 노드로부터 한 개의 키값을 차출하여 이동
    - 노드 합병 ( Node Merge ) : 재분배가 불가능할 때 사용한다.
      * 형제 노드들이 최소의 키(m/2-1)값만을 가지고 있는 경우에 적용
      * 왼쪽 노드와 먼저 합친다.
      * 합병 결과로 공백이 된 노드는 제거 --> 트리의 구조가 변경
* Best Sibling 선택
  + 양쪽 형제 노드 모두 사용 가능하다.
  + 노드의수가 많은 쪽을 선택하는 것이 Underflow 발생 회수를줄이는데 도움이 된다.
  + 재분배의 경우
    - Underflow가 아닌 쪽을 선택한다.
    - 양쪽 모두 underflow 라면, 노드의 수가 많은 쪽을 선택한다.
  + 노드 합병의 경우
    - m이 홀수인 경우 : 좌우 모두 m/2-1 이므로 아무거나 사용한다.
    - m이 짝수인 경우 : 노드의 수가 m/2인 쪽을 선택한다.

### < B-Tree 노드 합병 참고 >







* 더 빠른 사용을 위해 B-Tree를 사용한다.

-> B-Tree를 이용하여 DISK접근을 줄인다. ( BST는 접근 횟수가 B-Tree보다 크다 )

# B\*-트리

* B-트리의 문제점
  + B-Tree의 구조를 유지하기 위해 추가적인 연산이 필요하다.

-> 탐색 연산의 성능을 높이기 위해 지부해야 하는 대가로서 너무 비싸다.

* B\*-Tree는 B-Tree의 오버헤드를 줄이고 삽입과 삭제 연산의 성능을 개선하기 위해 고안된 B-Tree의 변형이다.

#### < B\*-Tree 특징 >

1. B\*-Tree 는 공백이거나 높이가 1 이상인 m-원 탐색 트리
2. root != leaf 인 경우 최소 2개, 최대 2 \* (2m - 2) / 3 + 1개의 서브트리를 갖는다.
3. 루트와 리프를 제외한 모든 노드는 적어도 (2m-2)/3+1 개의 서브트리를 갖는다.
4. 모든 리프는 같은 레벨에 있다.

# Trie ( 발음 = Try / reTRIEval의 약자 )

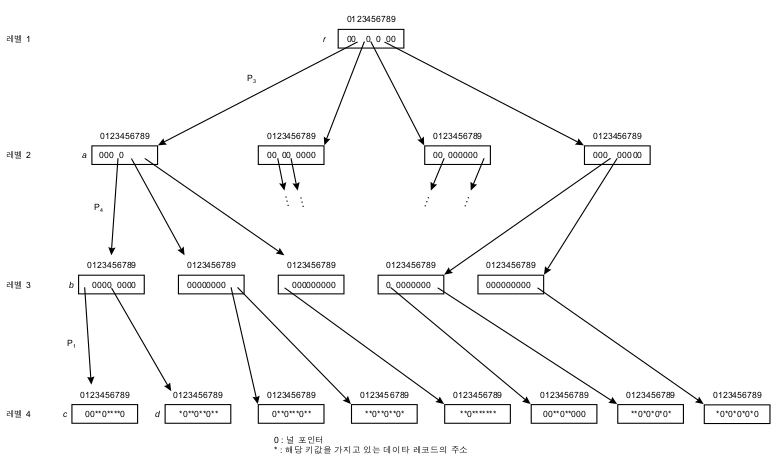
**B-Tree** : 트리 구조를 이용한 키 값 비교

**Trie** : 검색을 위해 키 값을 그대로 이용하지 않고, <u>\*\*키를 구성하는 문자나 숫자의 순서를 이용해 키 값을 검색\*\*</u>할 수 있는 자료 구조. **m-진 트리** 이지만 **m-원 탐색 트리**는 아니다. ( 키 값의 순서가 m원 탐색 트리 규칙과 다르다 )

**m진 트라이(m-ary trie)**

* 차수 m : 키 값을 표현하기 위해 사용하는 문자의 수, 즉 기수(radix)
* 숫자 : 기수가 10이므로 m=10, 영문자 : m = 26
* m진 트라이 : m개의 포인터를 표현하는 1차원 배열
* 트라이의 높이 : 키 필드(스트링)의 길이

**높이가 4인 10진 트라이**



**트라이 연산**

1. 탐색
   * 탐색 끝 : 리프 노드에서, 중간에 키 값이 없을 때
   * 탐색 속도 ≈ 키 필드의 길이 = 트라이의 높이
   * 최대 탐색 비용 <u><</u> 키 필드의 길이
   * 장점 : 균일한 탐색시간(단점 : 저장 공간이 크게 필요)
   * 선호하는 이유 : 없는 키에 대한 빠른 탐색 때문에
2. 삽입
   * 리프 노드에 새 레코드의 주소나 마크를 삽입
   * 리프 노드 없을 때 : 새 리프 노드 생성, 중간 노드 첨가
   * 노드의 첨가나 삭제는 있으나 분열이나 병합은 없음
3. 삭제
   * 노드와 원소들을 찾아서 널 값으로 변경
   * 노드의 원소 값들이 모두 널(공백노드) : 노드 삭제