# 균형 트리

- 클러스터형 인덱스와 보조 인덱스는 모두 내부적으로 균형 트리로 만들어짐

- ‘자료 구조’에 나오는 범용적으로 사용되는 데이터의 구조

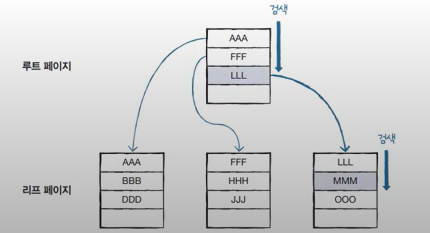
- 균형 트리는 무조건 루트 페이지부터 검색

- 균형 트리 구조에서 데이터가 저장되는 공간을 ‘노드’라고 함(나무가 뒤집어진 형태를 생각하면 쉬움)

1. 노드의 가장 상위 노드는 루트 노드

2. 중간에 끼인 노드들은 중간 노드

3. 제일 마지막에 존재하는 노드는 리프 노드



\* 균형 트리의 페이지 분할

-> 페이지 분할로 인해 SELECT는 빨라지지만 데이터 변경 작업(UPDATE, DELETE, INSERT) 시 성능이 나빠짐

# - 클러스터형 인덱스 구조

USE market\_db;

CREATE TABLE cluster -- 클러스터형 테이블

( mem\_id CHAR(8) ,

mem\_name VARCHAR(10)

);

INSERT INTO cluster VALUES('TWC', '트와이스');

INSERT INTO cluster VALUES('BLK', '블랙핑크');

INSERT INTO cluster VALUES('WMN', '여자친구');

INSERT INTO cluster VALUES('OMY', '오마이걸');

INSERT INTO cluster VALUES('GRL', '소녀시대');

INSERT INTO cluster VALUES('ITZ', '잇지');

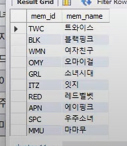
INSERT INTO cluster VALUES('RED', '레드벨벳');

INSERT INTO cluster VALUES('APN', '에이핑크');

INSERT INTO cluster VALUES('SPC', '우주소녀');

INSERT INTO cluster VALUES('MMU', '마마무');

SELECT \* FROM cluster;

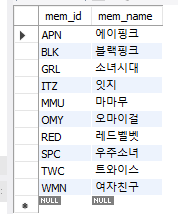
 -> 기본키가 지정되지 않았기 때문에 클러스터형 인덱스가 생성되지 않아서 정렬되지 않음

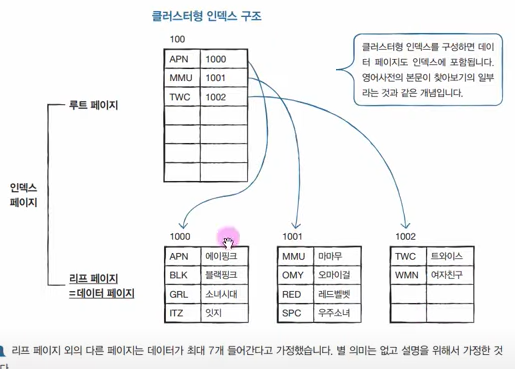
ALTER TABLE cluster

ADD CONSTRAINT

PRIMARY KEY (mem\_id);

SELECT \* FROM cluster;

 -> 클러스터형 인덱스가 생성된 후 영어 사전처럼 정렬



-> 내부적으로 생성된 클러스터형 인덱스 구조

# - 보조 인덱스 구조

USE market\_db;

CREATE TABLE second -- 보조 인덱스 테이블

( mem\_id CHAR(8) ,

mem\_name VARCHAR(10)

);

INSERT INTO second VALUES('TWC', '트와이스');

INSERT INTO second VALUES('BLK', '블랙핑크');

INSERT INTO second VALUES('WMN', '여자친구');

INSERT INTO second VALUES('OMY', '오마이걸');

INSERT INTO second VALUES('GRL', '소녀시대');

INSERT INTO second VALUES('ITZ', '잇지');

INSERT INTO second VALUES('RED', '레드벨벳');

INSERT INTO second VALUES('APN', '에이핑크');

INSERT INTO second VALUES('SPC', '우주소녀');

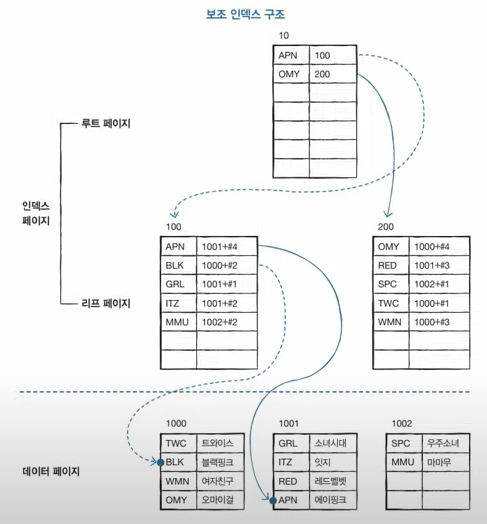
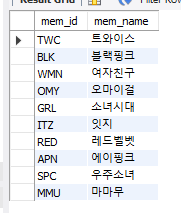
INSERT INTO second VALUES('MMU', '마마무');

ALTER TABLE second

ADD CONSTRAINT

UNIQUE (mem\_id);

SELECT \* FROM second;



====> 인덱스 검색을 통해 클러스터형 인덱스는 2페이지만 읽어서 원하는 결과를 검색, 보조 인덱스는 3페이지를 읽어서 원하는 결과를 검색 즉 두 인덱스 모두 검색이 빠르지만 클러스터형 인덱스가 조금 더 빠름