

INDEX

co2plant

2025.09.10

CONTENTS

SECTION 01

인덱스가 채용한 자료구조

1-1. B-Tree

 1-1-1. 인덱스의 탐색 과정(검색/삽입/삭제)

 1-1-2. B+Tree

 1-1-3-. B^{*}Tree

1-2. Hash

1-3. etc

SECTION 02

인덱스의 동작 원리

2-1. 옵티마이저와 인덱스 사용 결정 과정

SECTION 03

인덱스 종류

- 3-1. Primary vs Secondary
- 3-2. Cluster vs Non-Cluster
- 3-3. Column vs multi Column
- 3-4. Unique Index

CONTENTS

SECTION 04

특수 인덱스

- 4-1. 함수/표현식 기반 인덱스
- 4-2. 커버링 인덱스
- 4-3. 부분 인덱스
- 4-4. 전문 검색(Full-Text) 인덱스

SECTION 01

SECTION 01

B-tree

B-Tree란?

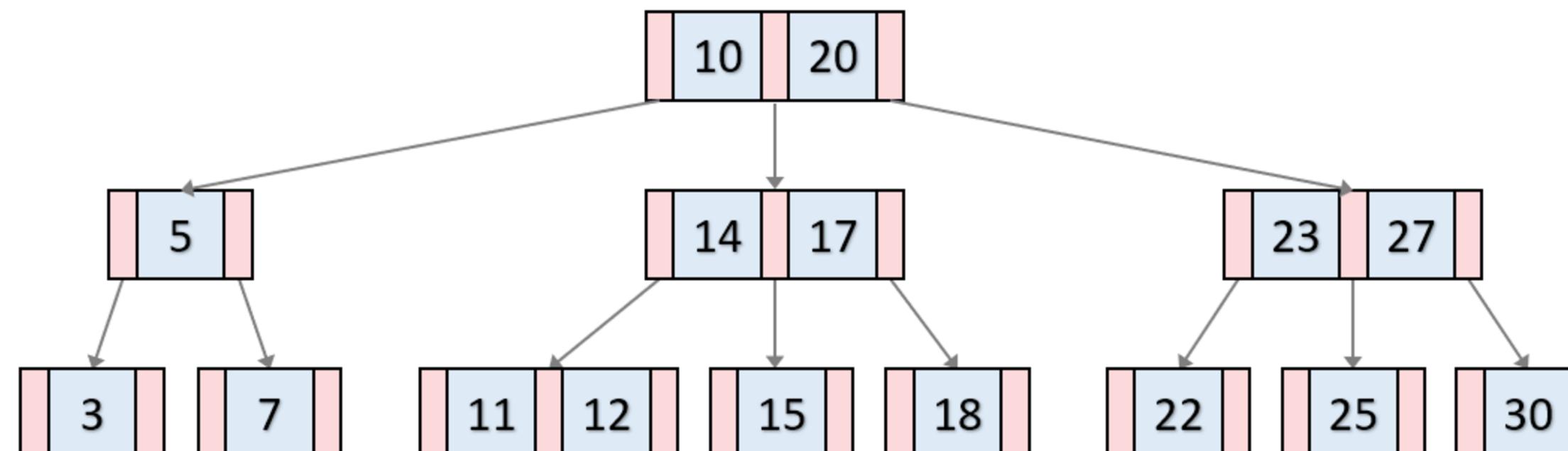
모든 리프노드들이 같은 레벨을 가질 수 있도록
밸런스를 맞춰주는 균형 탐색 트리

특징

균형잡힌 구조

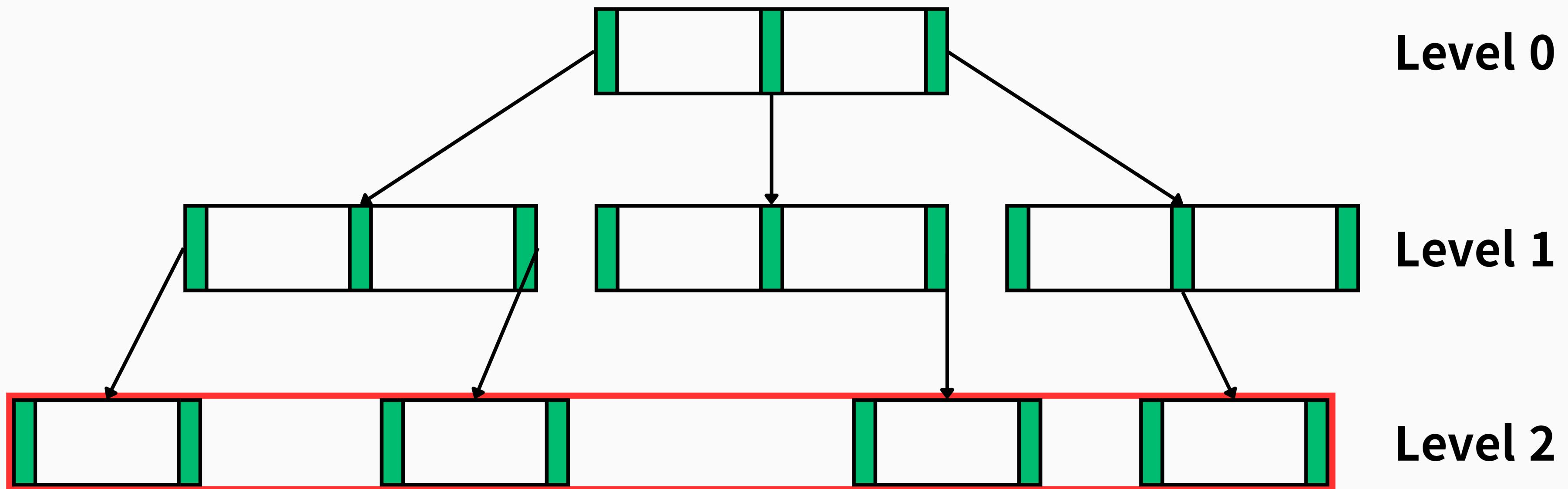
다중 경로 검색 트리

항상 정렬된 데이터



균형 잡힌 구조

리프 노드가 항상 같은 Level을 유지함

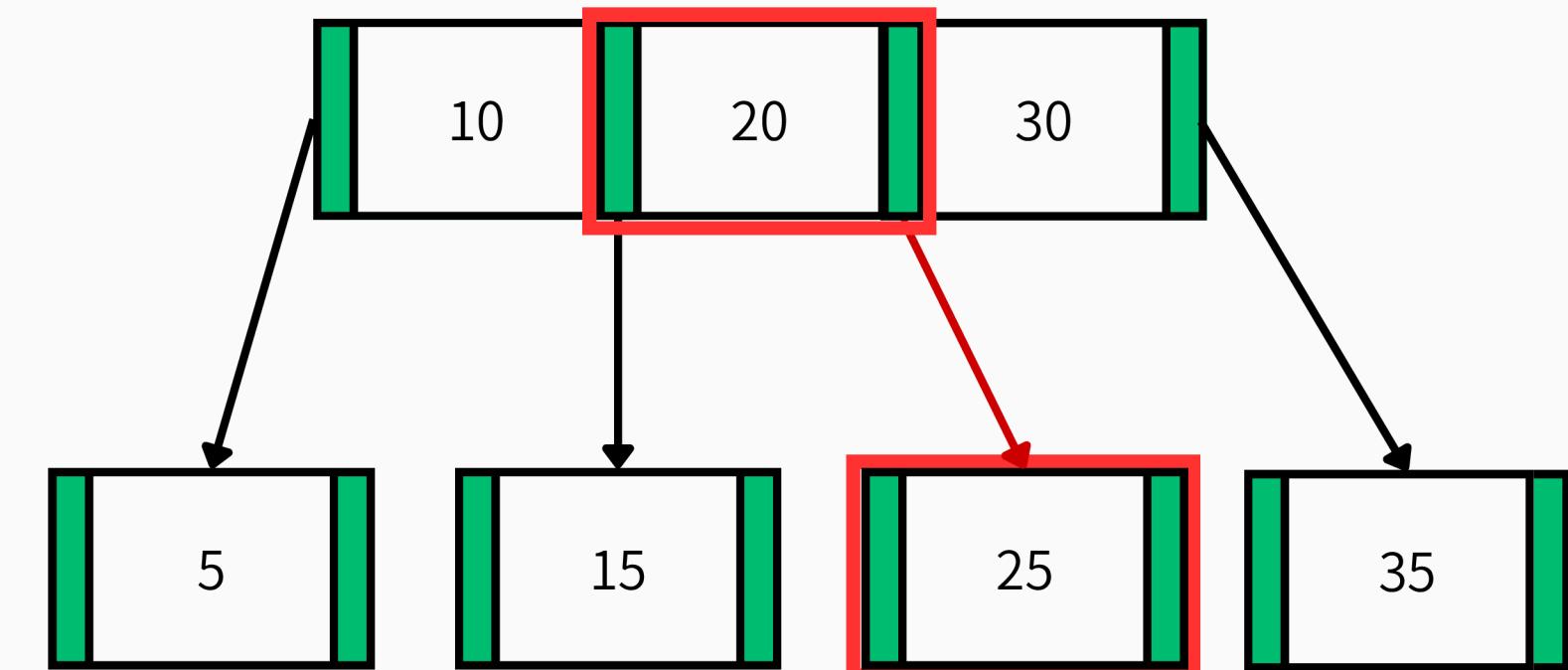
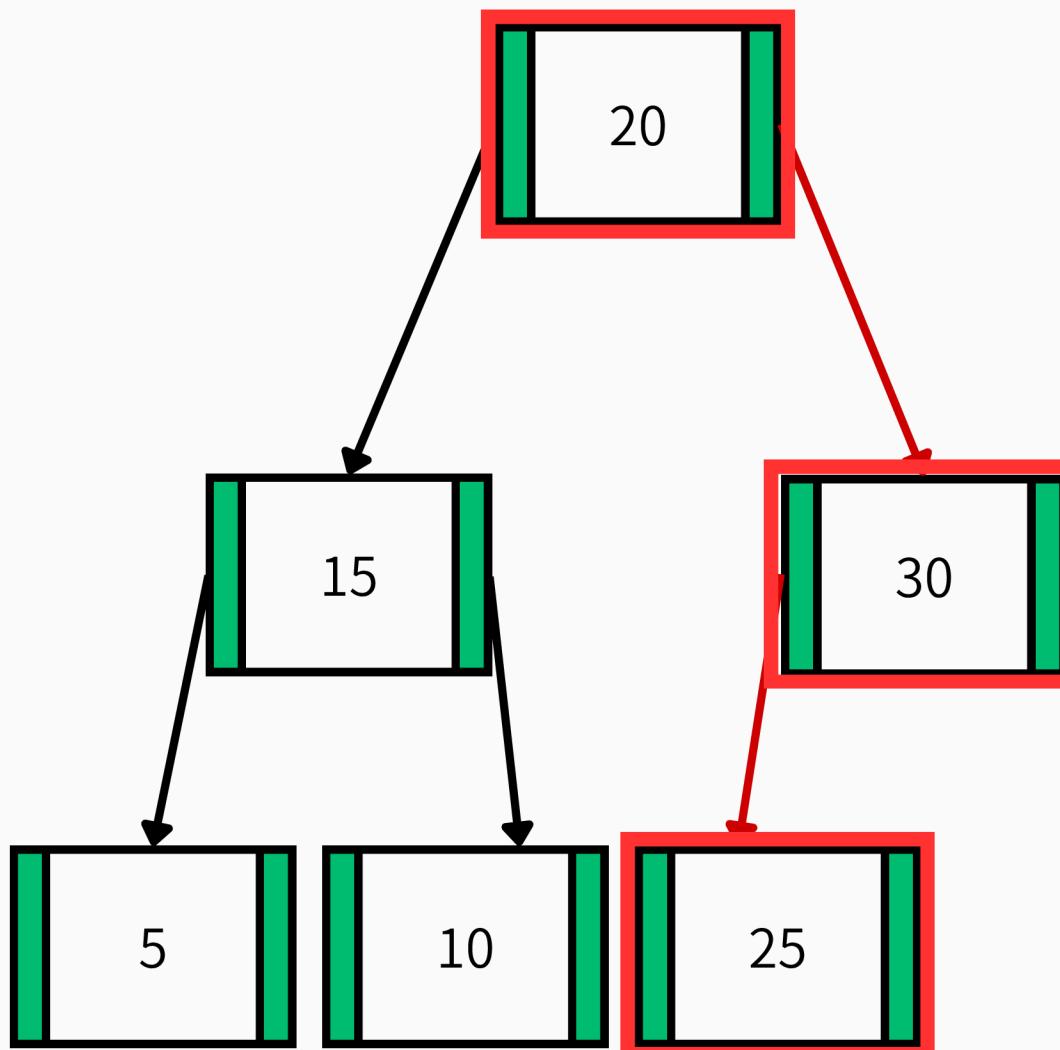


장점 - 균형 잡힌 구조

- 트리 높이가 낮다 = 리프 노드까지 도달하는데 I/O 횟수가 적다
- 트리 높이가 같다 = 일관된 성능 보장

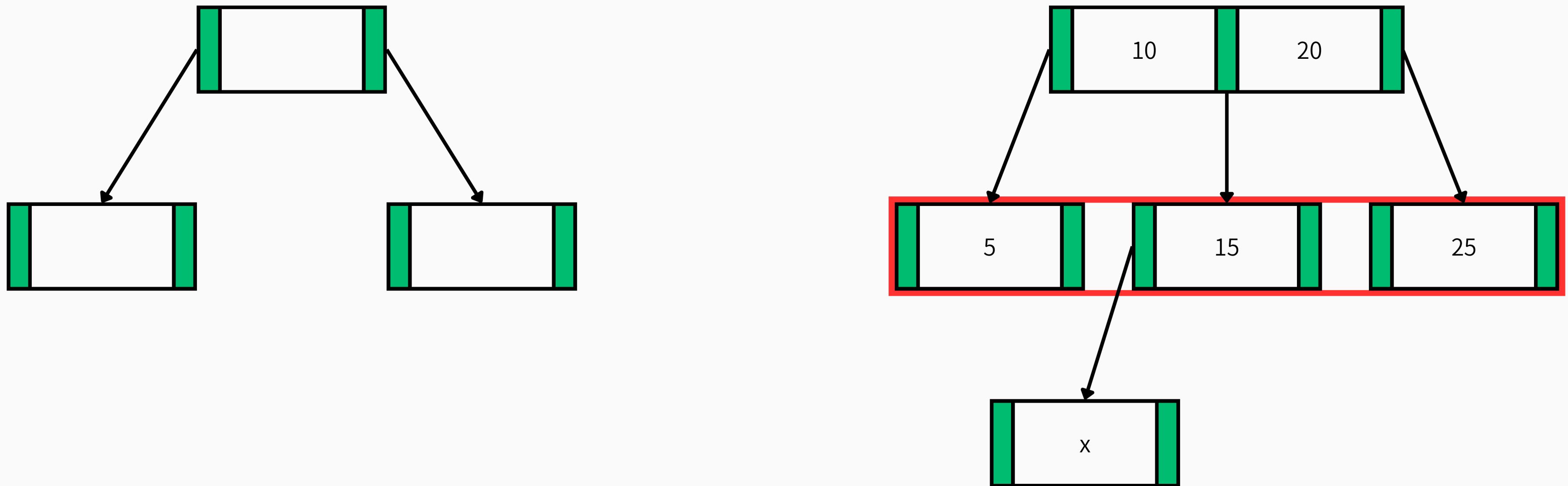
장점 - 균형 잡힌 구조

25를 찾는 경우 7개의 노드지만 이진트리는 3회 탐색해야함



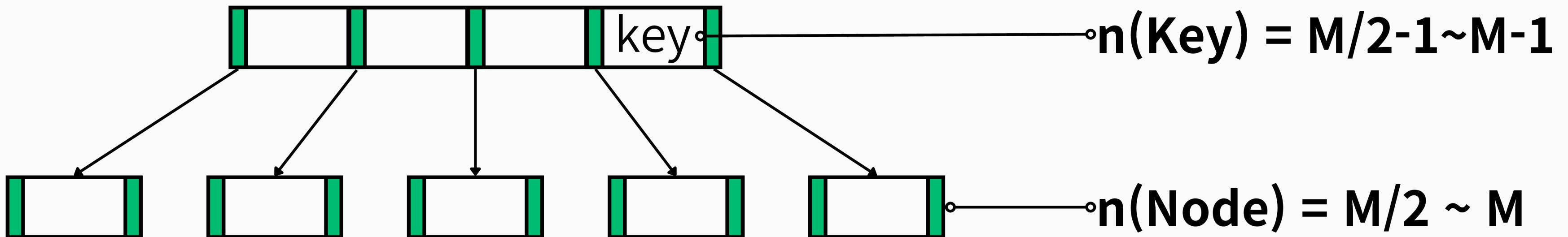
다중 경로 검색 트리

하나의 노드에 두 개 이상의 자식을 가짐



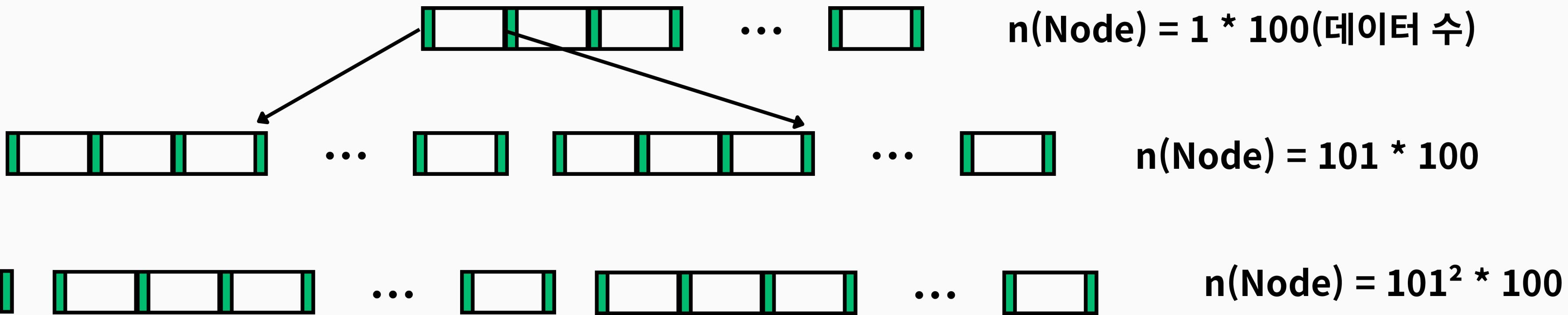
M차 B-tree

5차 B-tree를 예시로



장점 - 다중 경로 검색 트리

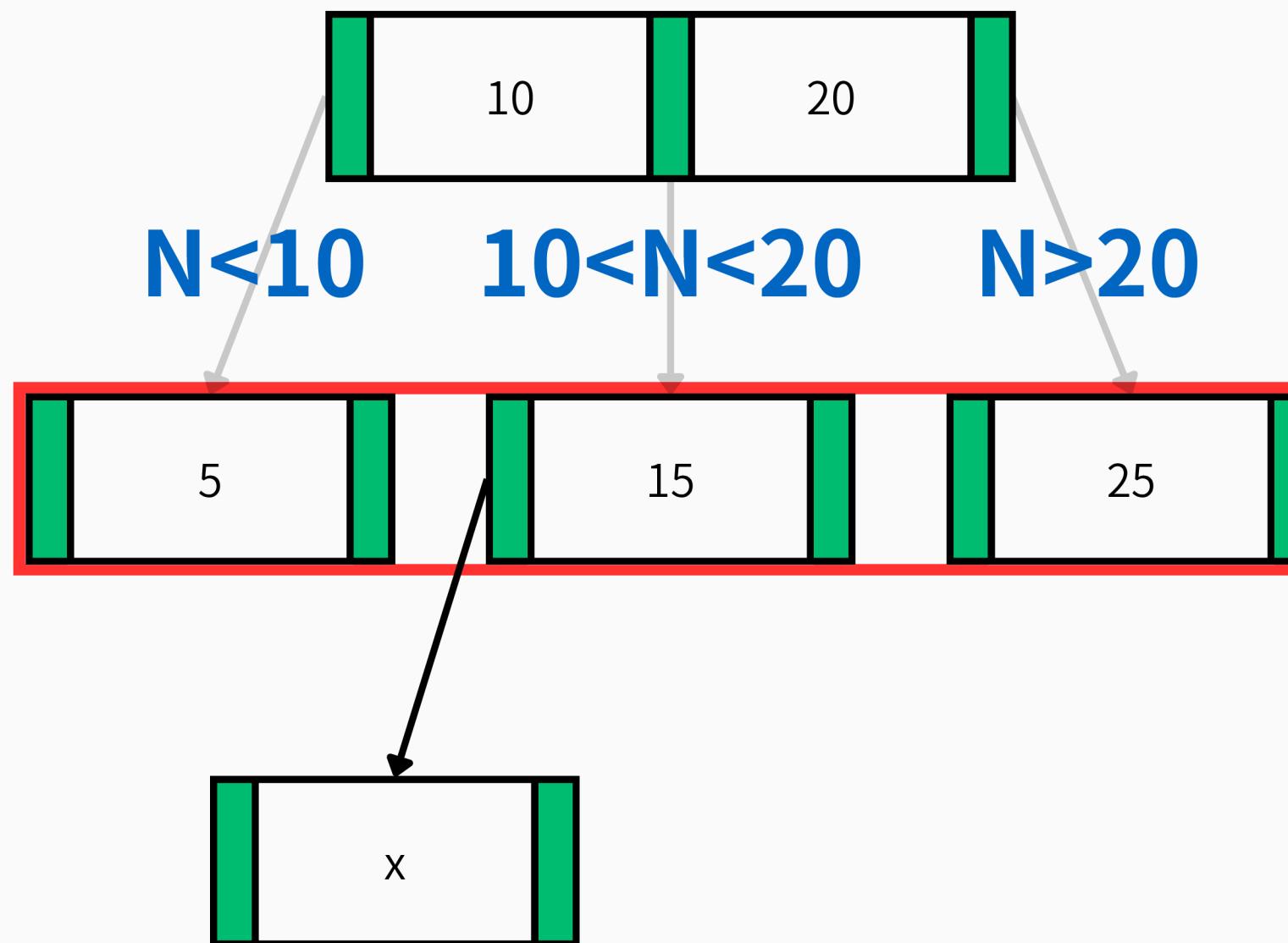
101차 B-tree



- 대용량 데이터 처리에 적합

장점- 데이터 정렬

검색, 순차 접근 시간 복잡도 $O(\log N)$ + Range 검색



DB의 특징

HDD or SSD 등 보조 기억 장치에 데이터를 저장함

- 보조기억장치 접근 회수 ↓ 성능 ↑
- 데이터를 **Block 단위로 저장함**

B-tree는 노드의 크기를 디스크 블록 크기와 일치 시키거나 크게 설계 가능함 → 한번에 블록 하나를 통째로 가져옴

단점

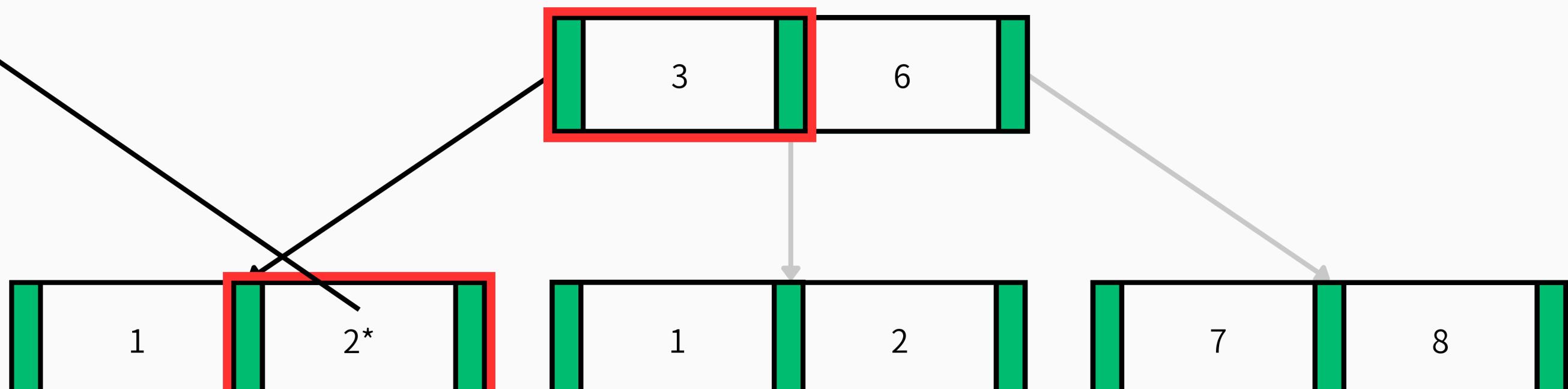
- 삽입, 삭제, 수정 시 오버헤드 발생 - 디스크 I/O증가
- 작은 테이블에서 상대적 비효율성

B-tree

ID	data
1	John
2	Sara
3	Tom
4	Jerry
5	Alo
6	San
7	Lee
8	Kim
9	Wang

동작 원리 - 검색

Where ID = 2(ID가 2인 행 검색 시)



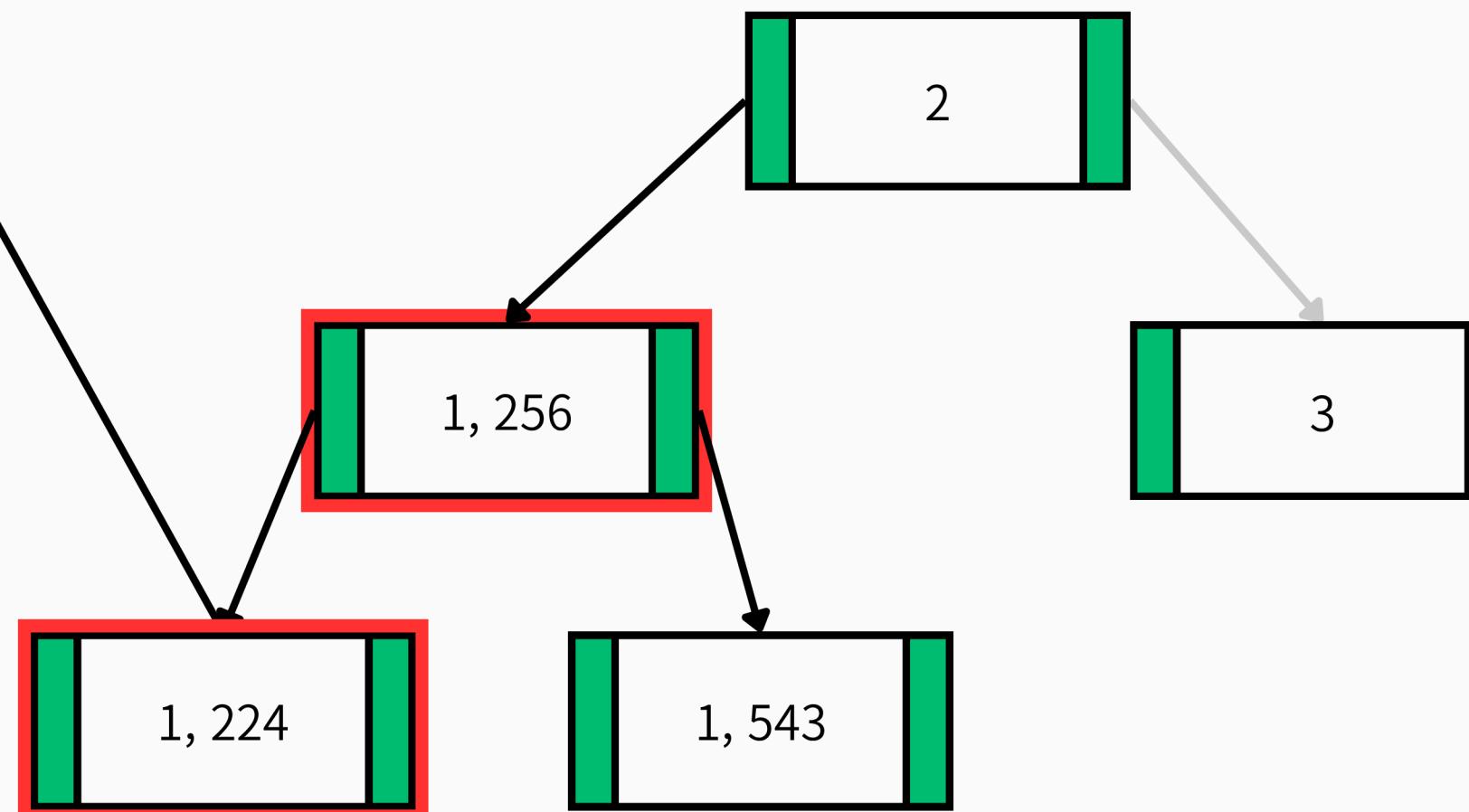
리프노드는 pointer
데이터를 가지고 있다.

B-tree

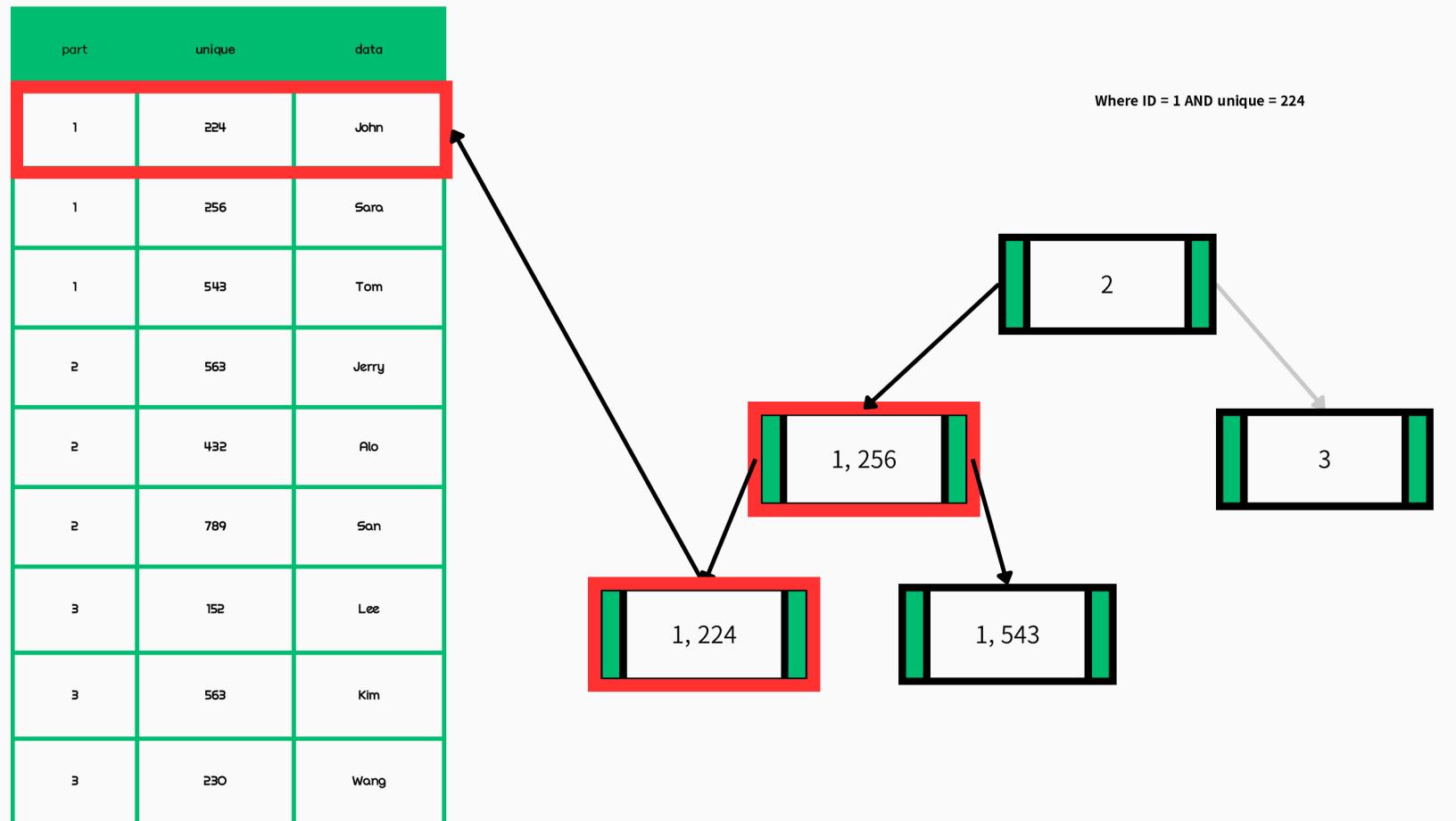
part	unique	data
1	224	John
1	256	Sara
1	543	Tom
2	563	Jerry
2	432	Alo
2	789	San
3	152	Lee
3	563	Kim
3	230	Wang

동작 원리 - 검색

Where ID = 1 AND unique = 224



일단 먼저오는 part를 기준으로 정렬 →
unique를 기준으로 정렬해서 검색함

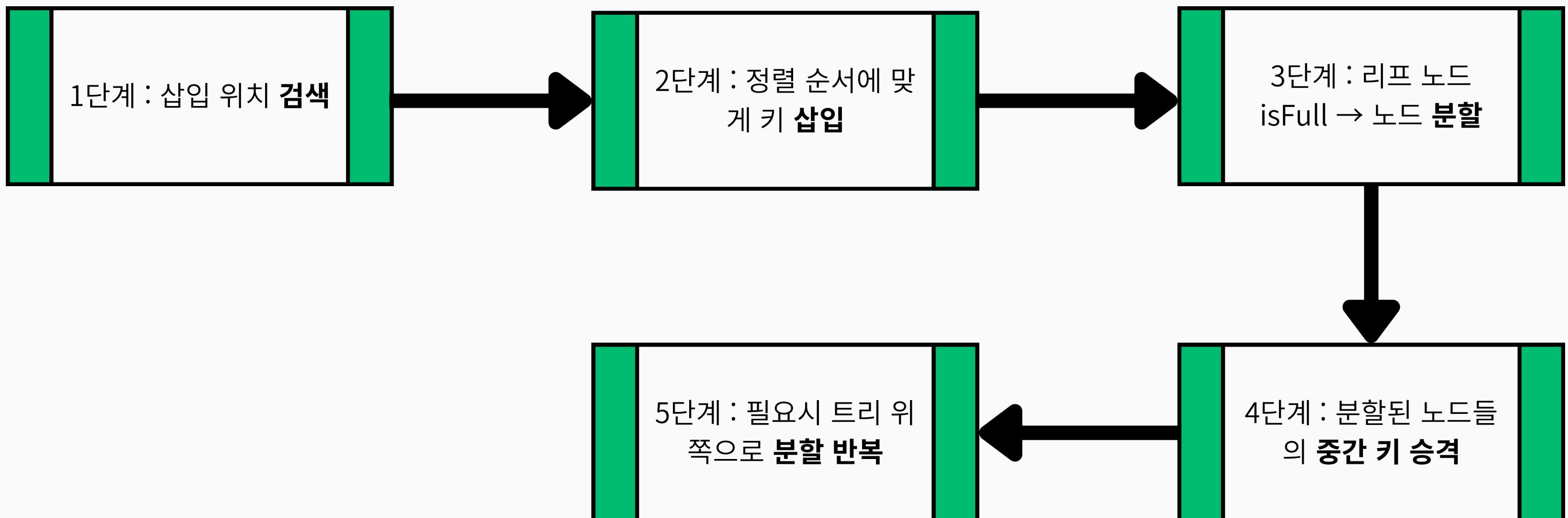


인덱스 왼쪽 접두사 원칙

복합 인덱스는 정의된 컬럼의 왼쪽부터
순서대로 조건이 사용될 때만 효율적으로 동작함

삽입, 삭제, 수정이 느린 이유

정렬상태를 유지하기 때문에...



삽입, 삭제, 수정이 느린 이유

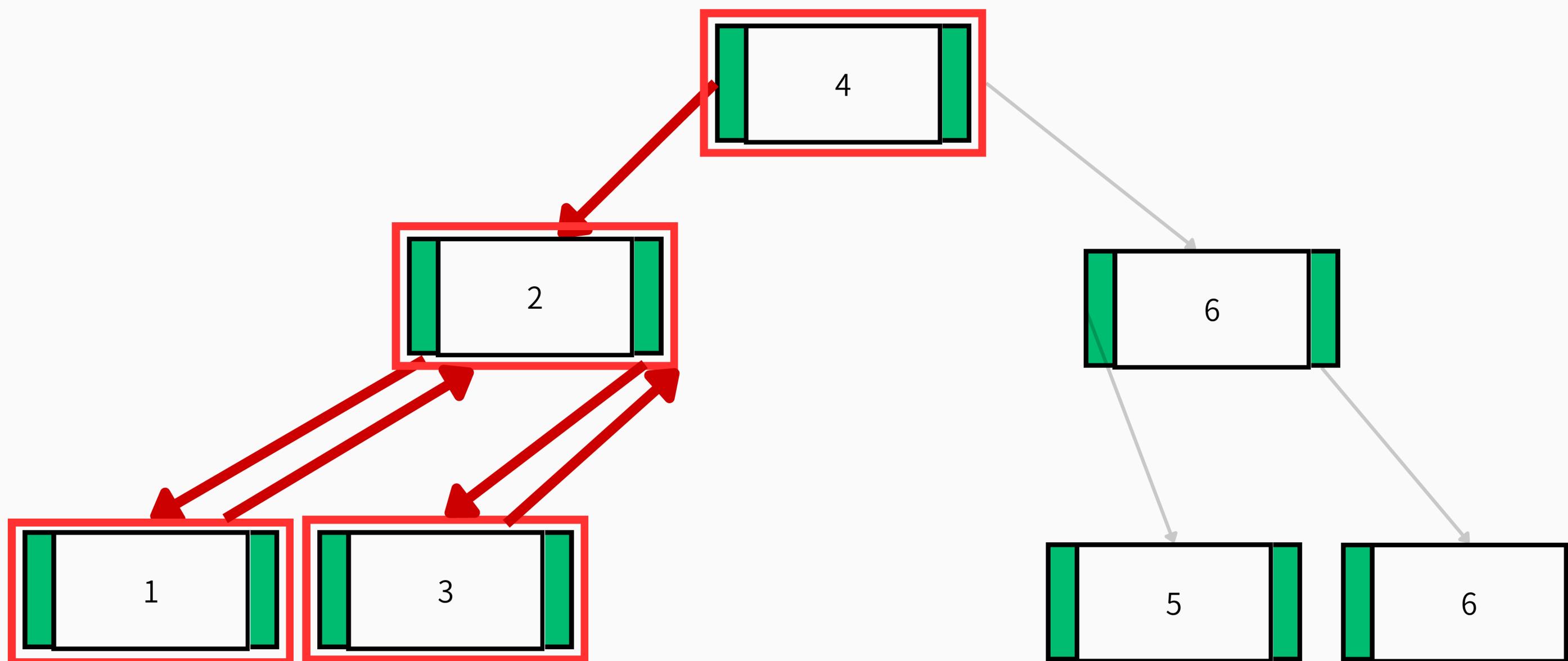
<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

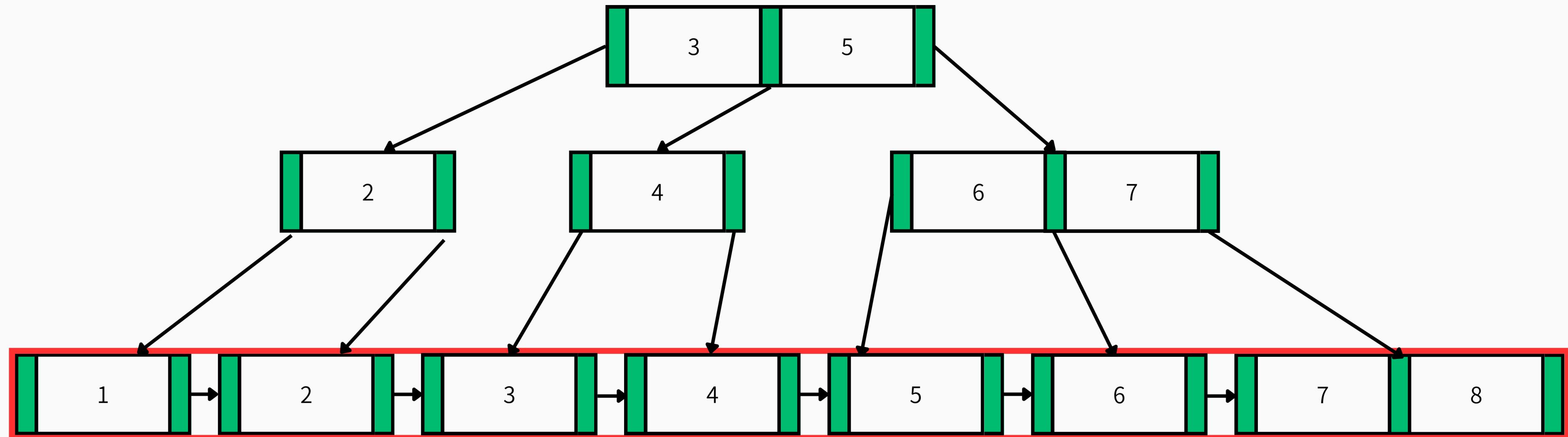
SECTION 01

B+tree

B-Tree는 탐색을 위해서 상위 노드로 이동해야함

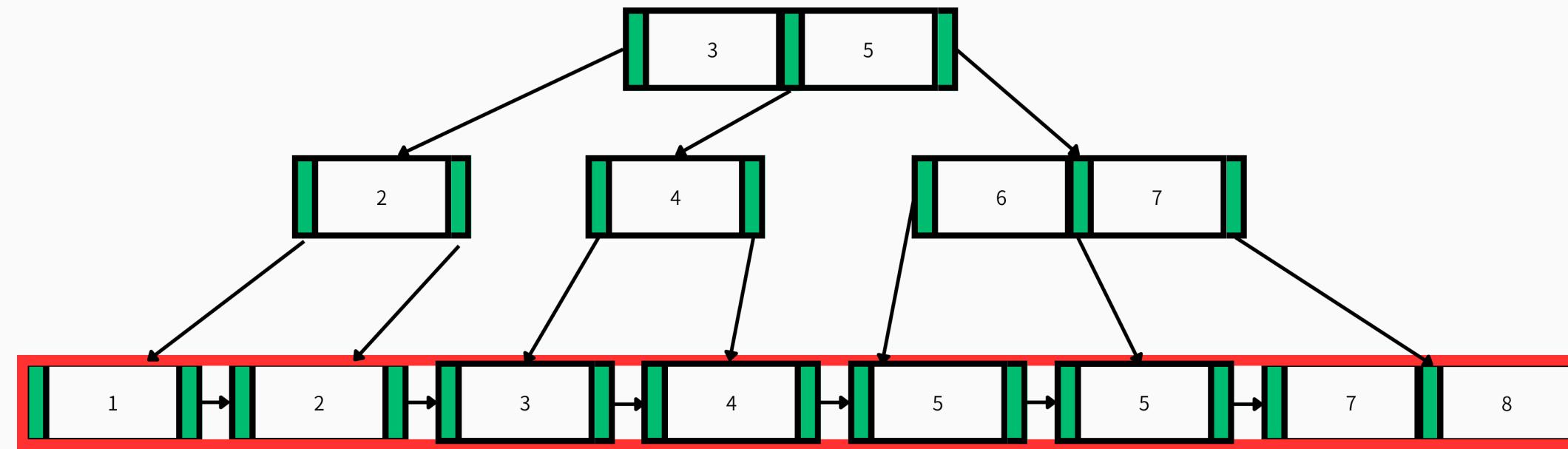


리프 노드의 연결과 키 중복



같은 레벨의 형제 노드는 **연결리스트 형태**로 이어져있음

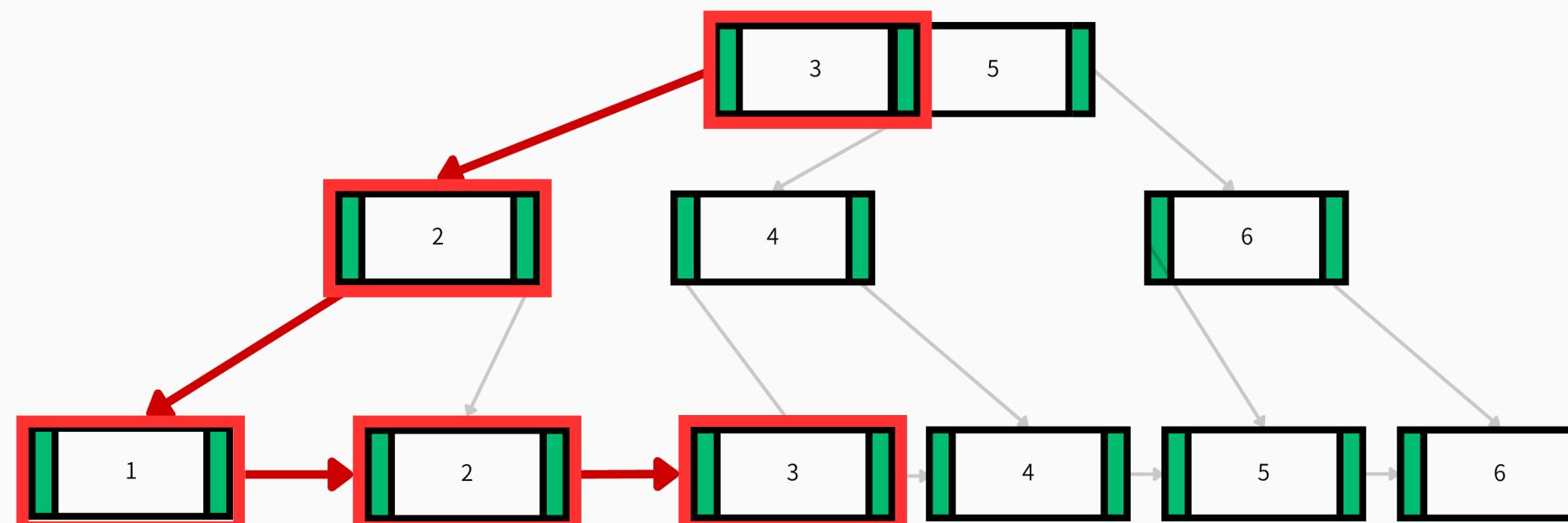
왜 이런 이상한 구조를 사용하는가?



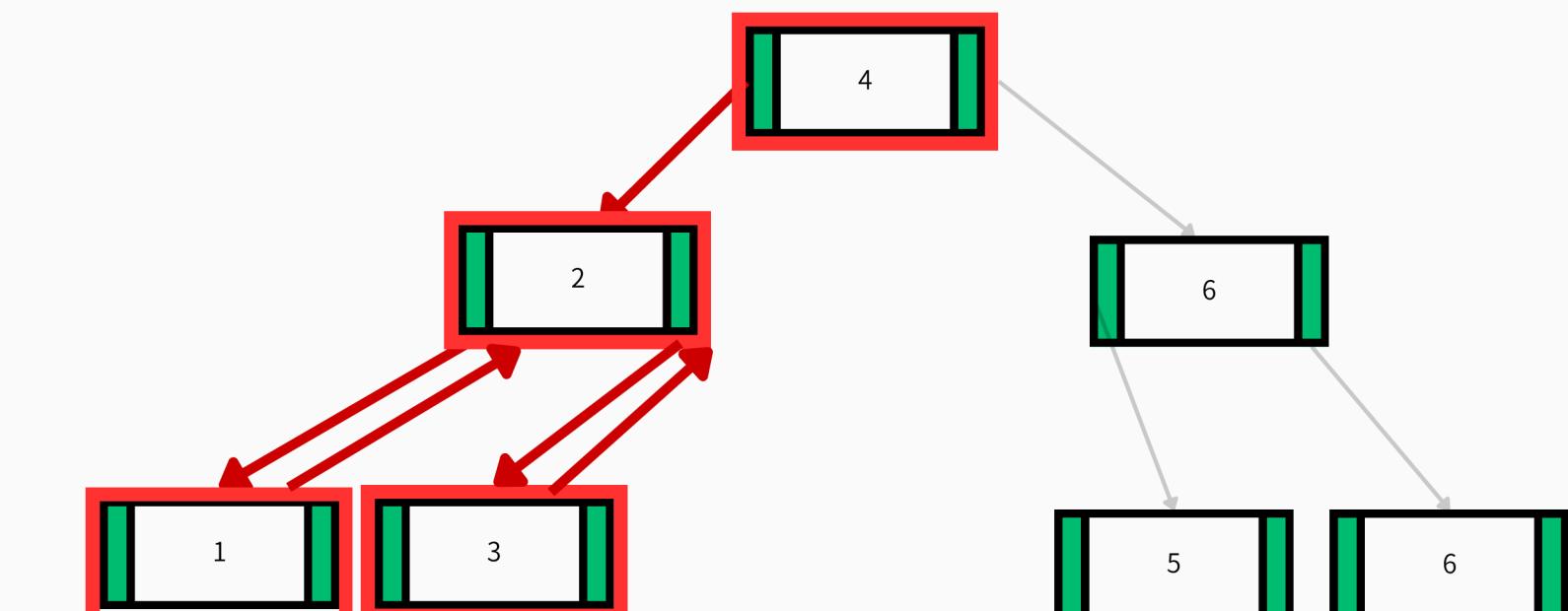
인덱스를 사용하는 이유는 **조회, 검색 속도**

B-Tree보다 더 빨리 조회하기 위해

왜 빠르다는 거죠?



4회



5회

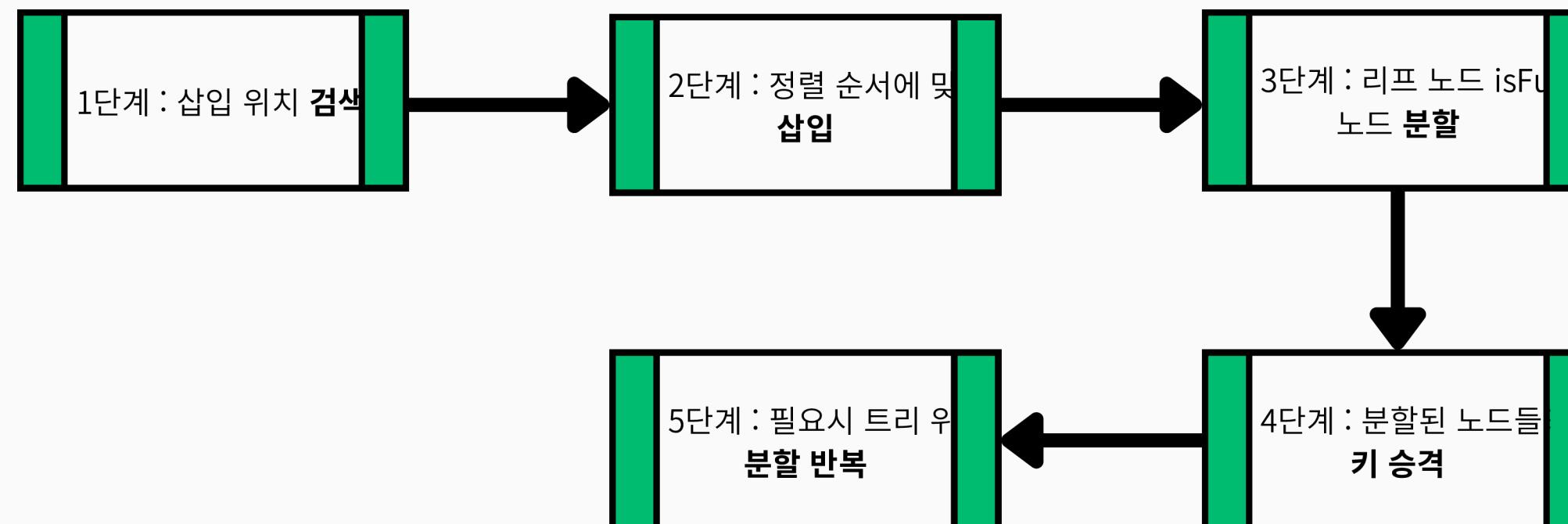
“인덱스를 사용하는 이유는 조회, 검색 속도“

SECTION 01

B*tree

B-Tree는 구조 유지를 위해 추가 연산, 새로운 노드 생성

정렬상태를 유지하기 때문에...



B^{*}Tree란?

리프 노드에만 실제 데이터, 포인터를 저장

→ 내부 노드는 더 많은 키를 저장

→ 트리의 높이가 더욱 낮게 유지됨

변경점

- B-Tree

최소 $M/2$

노드 가득 차면 분열

- B^{*}Tree

최소 $2M/3$ 개

노드가 가득 차면 이웃한 형제 노드로 재배치

SECTION 01

B-tree 결론

B-Tree란?

모든 리프노드들이 같은 레벨을 가질 수 있도록
밸런스를 맞춰주는 균형 탐색 트리

Inno DB는 B-Tree에 리프 노드 간 연결을 더한 B+Tree를 사용

B-Tree는 언제 써야할까?

범위 검색을 사용

동등 검색을 사용

카디널리티(**중복되지 않는 값이 많은**)가 높은 컬럼

B-Tree는 언제 쓰지 말아야 할까?

쓰기 작업이 매우 빈번한 테이블

데이터 중복도가 높은 컬럼(**예시 : 성별**)

데이터 규모가 작은 테이블

SECTION 01

Hash

Hash기반 Index란?

hash table을 이용해 구현하는 index

ID	Key
0	24
1	
2	20
3	
4	
5	II



- 시간복잡도 $O(1)$
- range비교는 불가능
- multicolumn index
 - index를 구성하는 전체 attributes에 대한 조회만 가능
- rehashing이 부담스러움

단점

range비교는 불가능

ID	Key
0	24
1	
2	20
3	
4	
5	II

Rehashing

hash table을 이용해 구현하는 index

ID	Key
0	24
I	II
2	20

원래는 임계값을 넘으면 바로
rehashing한다

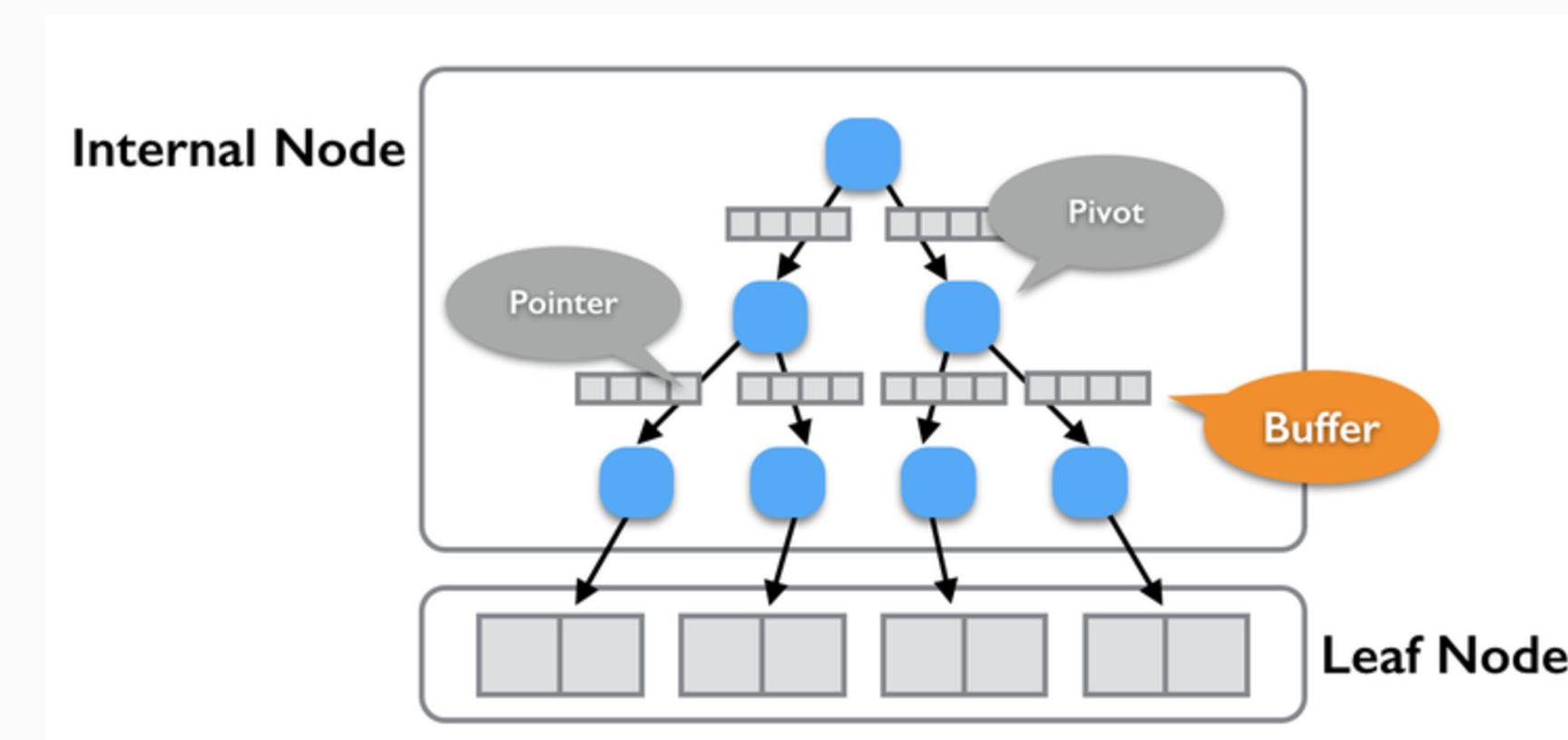


ID	Key
0	24
I	
2	20
3	
4	
5	II

SECTION 01

etc

프랜탈 트리 인덱스

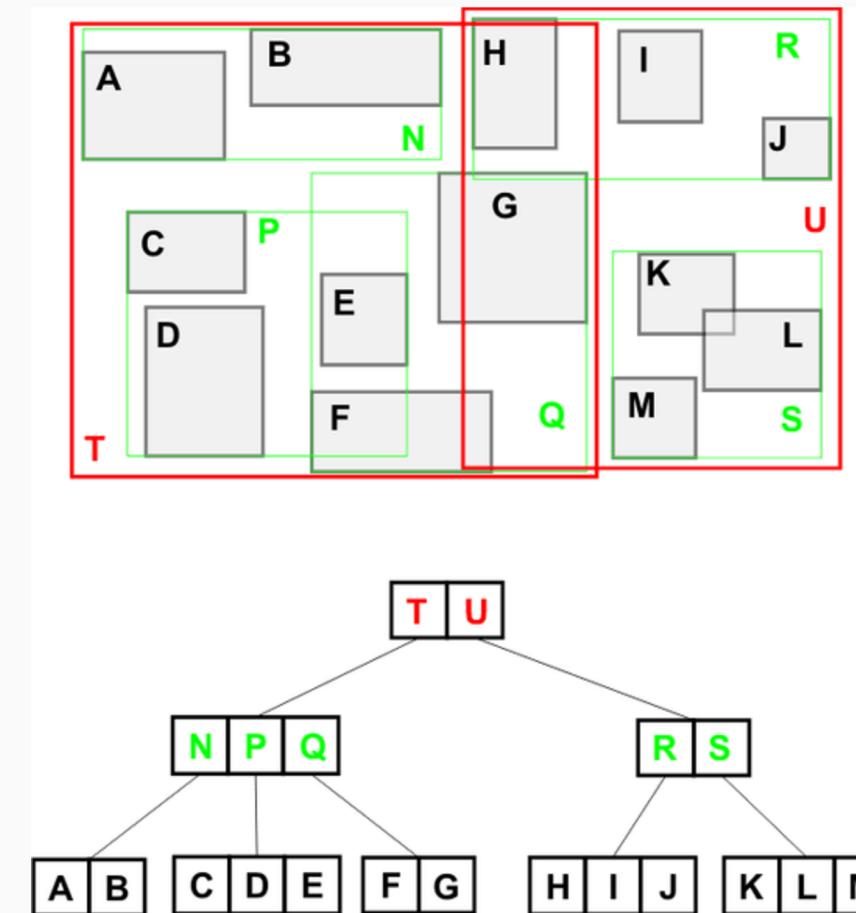


- 프랙탈-트리 인덱스
데이터 저장, 삭제 시 처리 비용을 줄인 설계

빠른 삽입 및 삭제 - 기존 MySQL보다 100배 이상 빠른속도가 나오기도 함

B-Tree보다 큰 4MB의 내부 노드 버퍼를 사용함

R-Tree



- **R-Tree**
CAD 및 지리 데이터에서 요구되는 공간 데이터를 효율적으로
저장하기 위한 인덱스 구조
동적 구조를 가짐

전체 자료 구조 비교

자료구조	범위 검색 (Range Scan)	동등 검색 (Equality Search)	정렬 지원 (Sorting)	주요 특징
B-Tree	0	0	0	스스로 균형을 맞추며, 디스크 I/O에 최적화됨
B+Tree	◎ (매우 우수)	0	0	모든 데이터가 리프 노드에 있고, 리프 노드끼리 연결되어 범위 검색에 탁월
B*Tree	0	0	0	최소 점유율 2/3, B-Tree보다 저장 공간 효율이 높음
Hash	X	◎ (매우 우수)	X	O(1)의 매우 빠른 동등 검색 속도, 범위 검색은 불가능
Bitmap	0	0	X	성별, 등급 등 카디널리티가 매우 낮은 컬럼에 적합, 비트 연산으로 동작
GiST/R-Tree	0	0	△ (제한적)	지도, 도형 등 2D 이상의 공간/다차원 데이터를 위한 인덱스
Fractal Tree	0	0	0	B-Tree 구조에 캐시 친화적인 프랙탈 레이아웃을 적용한 구조

SECTION 02

SECTION 02

옵티マイ저와 인덱스 사용 결정 과정

옵티マイ저란?

SQL 쿼리를 받아 가장 효율적인 실행 계획을 생성하는
데이터 베이스 엔진의 핵심 구성요소

인덱스 사용 결정 4단계

SECTION 03

SECTION 03

Primary vs Secondary

Primary Index(주 인덱스)

테이블의 **기본키**에 대해 자동으로 생성되는 인덱스

기본키가 없다면 UNIQUE면서 NOT NULL인 컬럼으로 생성

Secondary Index(보조 인덱스)

기본키가 아닌 컬럼에 생성되는 인덱스

여러개 생성 가능하고 물리적 저장과 별개로 논리적으로 관리됨

Index의 종류

비교

구분	Primary Index	Secondary Index
개수	테이블당 1개	테이블당 여러 개
생성	자동 생성	수동 생성(Unique 제약조건 적용시 자동 생성)
NULL	허용 안함	허용 (조건부)
중복	허용 안함	허용 (조건부)
고유성	Unique && NOT NULL	Unique/Non-Unique 가능
용도	데이터 고유 식별	검색 성능 향상

SECTION 03

Cluster vs Non-Cluster

clustered Index

물리적 페이지를 인덱스 키 순서대로 정렬하는 인덱스

non-clustered Index

별도의 인덱스 구조를 가지며, 실제 데이터 위치를 가리키는 포인터를 저장
예시. 책 맨뒤의 부록

Index의 종류

비교

구분	Clustered Index	non-clustered Index
개수	테이블당 1개	테이블당 여러 개
생성	자동 생성	수동 생성(Unique 제약조건 적용시 자동 생성)
NULL	허용 안함	허용 (조건부)
중복	허용 안함	허용 (조건부)
고유성	Unique && NOT NULL	Unique/Non-Unique 가능
용도	데이터 고유 식별	검색 성능 향상

InnoDB에서는
Primary Index = Clustered Index
Secondary Index = non-clustered Index
라고 봐도 무방

SECTION 03

Column vs multi Column

Column Index

하나의 열에 대해 생성된 인덱스

- 단일 열의 값을 기반으로 빠른 검색 지원
- 특정 열에 대한 검색, 필터링이 빈번할 때 사용

- 인덱스에 포함된 열의 조합을 기반으로 검색 최적화
- 열 순서에 민감해서 주의해야한다.

Multi-Column Index

두 개 이상의 열을 조합하여 생성된 인덱스.
열의 순서가 매우 중요

SECTION 03

Unique Index

Unique Index

인덱스된 열 또는 열 조합에 중복을 허용하지 않는 인덱스

- 고유성 제약 조건을 강제하여 데이터 부결성 보장
- Unique Index는 DBMS에 따라 다르지만 NULL 값을 허용 한다.

전체 자료 구조 비교

인덱스 종류	특징	주요 용도	장점	단점
Primary Index	기본 키에 생성, 고유성 보장	기본 키 검색, 무결성 유지	빠른 검색, 무결성 보장	쓰기 작업 시 비용 발생
Secondary Index	기본 키 외 열에 생성, 중복 허용	추가적인 검색 최적화	유연한 검색 가능	추가 저장 공간, 쓰기 성능 저하
Cluster Index	데이터 물리적 정렬	범위 검색, 정렬	빠른 검색, 효율적 데이터 접근	쓰기 작업 시 재정렬 비용
Non-Cluster Index	데이터와 별도의 인덱스 구조	특정 열 검색	유연성, 다중 인덱스 가능	추가 I/O 및 저장 공간 필요
Column Index	단일 열에 생성	단일 열 조건 검색	간단, 특정 열 검색 최적화	복합 조건에서 효율 낮음
Multi-Column Index	다중 열 조합	다중 열 조건 검색	복합 조건 검색 최적화	열 순서 민감, 저장 공간 증가
Unique Index	고유성 제약	고유 값 검색, 무결성 유지	데이터 무결성, 빠른 검색	고유성 검증으로 쓰기 성능 저하

SECTION 04

SECTION 04

함수/표현식 기반 인덱스

Function/Expression-Based Index

테이블의 열에 직접 적용되지 않음
열에 함수나 표현식을 적용한 결과에 대해 생성된 인덱스

```
ALTER TABLE employees ADD upper_name VARCHAR(50) GENERATED ALWAYS AS (UPPER
(last_name)) STORED;
CREATE INDEX idx_upper_name ON employees (upper_name);
```

- 복잡한 조건 검색 최적화 가능
- 함수나 표현식이 포함된 쿼리 성능이 크게 향상됨
- InnoDB에서는 가상 열을 사용해서 구현한다.

SECTION 04

커버링 인덱스

Covering Index

인덱스를 활용하는 방법 중 하나
쿼리에 필요한 모든 컬럼이 인덱스에 포함되어 있음
인덱스만으로 쿼리를 완전히 처리 가능

```
CREATE INDEX idx_covering ON users (email, name, phone);
```

- 주로 Non-Clustered Index로 구현됨
- 물리적으로 직접 접근 X → I/O 비용 대폭 감소
- InnoDB에서는 복합 인덱스로 구현한다.

SECTION 04

부분 인덱스

Partial Index

특정 조건을 만족하는 행에 대해서만 생성된 인덱스
WHERE절을 사용해 인덱스 대상을 제한함

PostgreSQL

```
CREATE INDEX idx_active_users ON users (email) WHERE status = 'active';
SELECT * FROM users WHERE email = 'test@example.com' AND status = 'active';
```

- 데이터의 하위 집합만 자주 검색할 때 사용
 - 인덱스 크기 감소, 특정 쿼리에서 성능 향상

InnoDB에서 직접적인 지원은 없으나 가상 열, 뷰로 유사 구현은 가능하다

SECTION 04

전문 검색 인덱스

Full-Text Index

텍스트 컬럼을 효율적으로 검색하기 위해 설계된 인덱스
단어 단위로 텍스트를 분리하여 인덱싱

```
CREATE FULLTEXT INDEX idx_fulltext ON articles (content);
SELECT * FROM articles WHERE MATCH(content) AGAINST('database optimization');
```

- 데이터의 하위 집합만 자주 검색할 때 사용
 - 인덱스 크기 감소, 특정 쿼리에서 성능 향상

InnoDB에서 자연어 및 Boolean검색이 가능

INTERVIEW





A cartoon illustration of a person with dark hair, wearing a blue suit and red tie, standing behind a podium and pointing towards a large whiteboard. The whiteboard has a black border and contains the text 'B-Tree와 B+Tree의 차' on top and '이점은 무엇인가요?' below it. The background is a solid teal color, and the foreground shows the dark silhouettes of an audience seated in rows.

B-Tree와 B+Tree의 차
이점은 무엇인가요?

B-Tree와 B+Tree의 차이점은 무엇인가요?

Keywords

- 데이터 저장 위치(내부노드, 리프노드)
- 리프 노드 간 연결 리스트
- 범위 검색 효율성

**Index에서 B-Tree를 사용하는 이유는 무엇인가
요?**



Index에서 B-Tree를 사용하는 이유는 무엇인가요?

Keywords

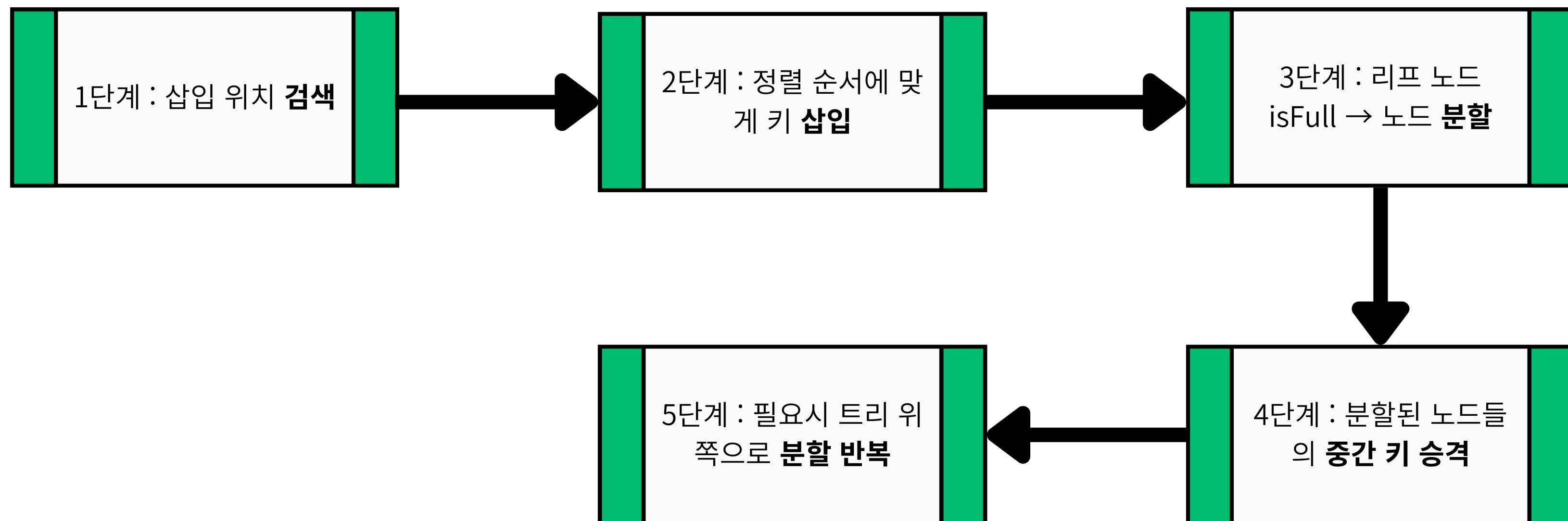
- 디스크 I/O
- 페이지/블록 단위 접근
- 트리의 높이
- 대용량 데이터 접근

인덱스가 있는 테이블에
INSERT나 DELETE 작
업이 발생할 때 내부적으
로 어떤 일이 일어나는가?



인덱스가 있는 테이블에 INSERT나 DELETE 작업이 발생할 때 내부적으로 어떤 일이 일어나는가?

Remind



출처

<https://velog.io/@emplam27> - 그림으로 알아보는 B-Tree