

인덱스의 필요성, B-트리

인덱스의 필요성

| 인덱스 | | EMPLOYEE | | | | | |
|------------|---------|----------|---------|-------|---------|---------|-----|
| EmpnoIndex | Pointer | EMPNO | EMPNAME | TITLE | MANAGER | SALARY | DNO |
| 1003 | | 2106 | 김창섭 | 대리 | 1003 | 2500000 | 2 |
| 1365 | | 3426 | 박영권 | 과장 | 4377 | 3000000 | 1 |
| 2106 | | 3011 | 이수민 | 부장 | 4377 | 4000000 | 3 |
| 3011 | | 1003 | 조민희 | 과장 | 4377 | 3000000 | 2 |
| 3426 | | 3427 | 최종철 | 사원 | 3011 | 1500000 | 3 |
| 3427 | | 1365 | 김상원 | 사원 | 3426 | 1500000 | 1 |
| 4377 | | 4377 | 이성래 | 사장 | ^ | 5000000 | 2 |

KEY(EID) - VALUE(pointer to record)로 이루어짐

장점

- 검색 속도를 크게 향상
- 인덱스는 실제 테이블의 데이터 크기에 비해 작아 메모리에 적재하기 쉽다.


단점

- 추가 저장 공간이 필요하다.(약 10%)
- insert, update, delete 등의 변동 사항이 있는 경우 성능이 저하됩니다. ⇒ 데이터 변경 시 인덱스도 수정되어 추가 비용이 발생

해쉬 함수

- A naïve hash function

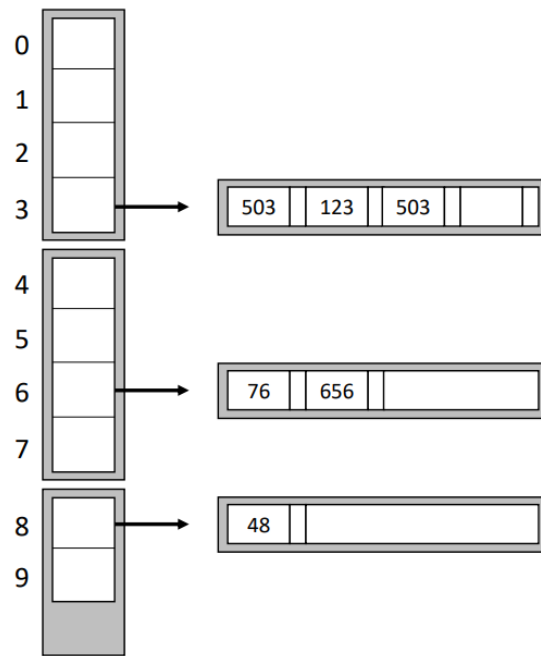
$$h(x) = x \bmod 10$$

 : A disk block

➔ Cost per lookup:

- One access in array
- One access in list

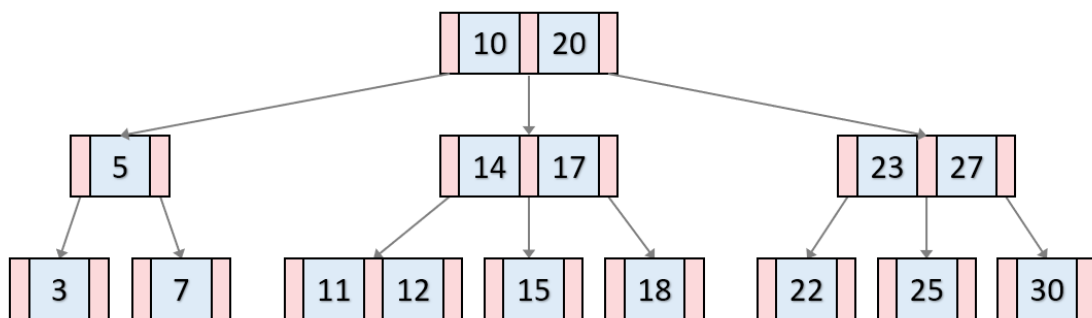
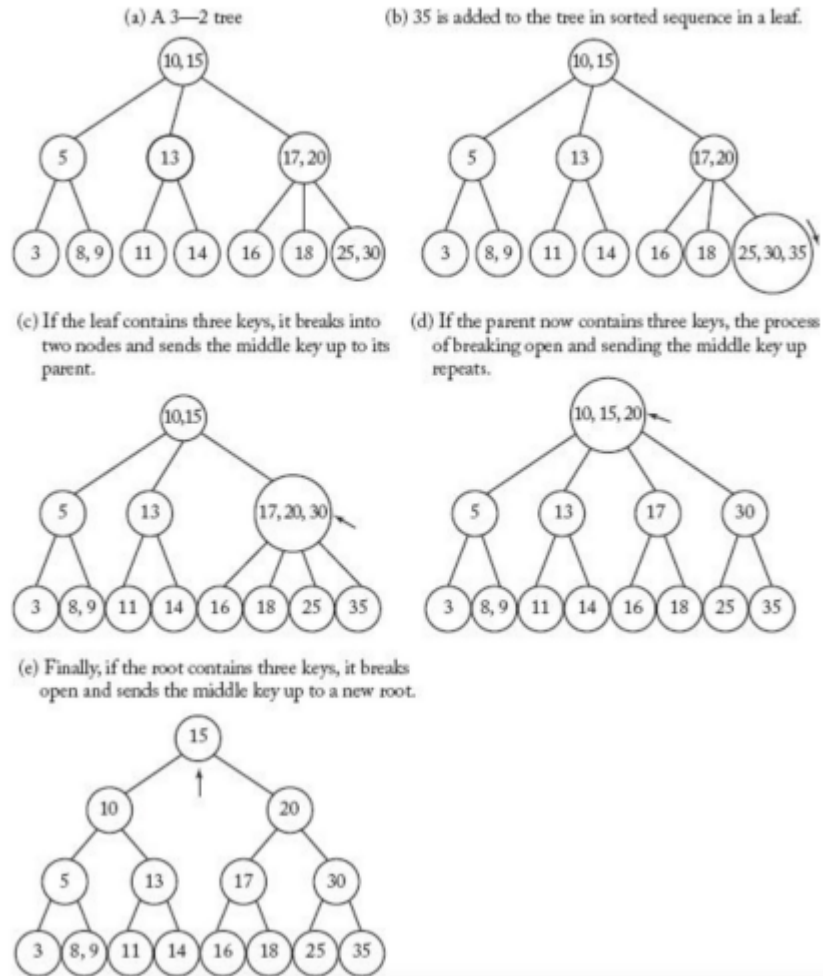
- No range queries



- 하나의 인덱스가 레코드로 연결
- 한번의 접근에 한번의 조회
- Range queries에 효율이 안좋다. ex) BETWEEN이나 in등

B - Tree

B-trees



3차 B-트리

- 노드는 최대 M개의 자식 노드를 가질 수 있다.
ex) 3차 B-트리라면 최대 3개의 자식 노드를 가질 수 있다.

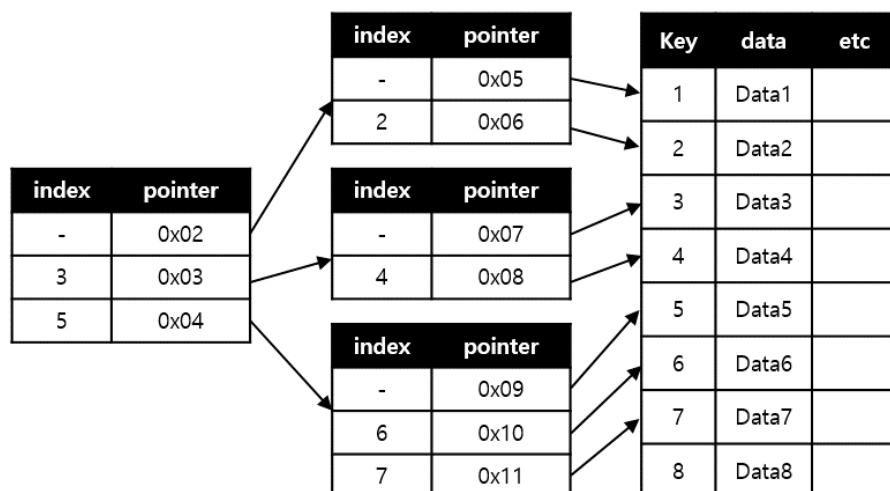
- 노드에는 최대 $M-1$ 개의 KEY를 가질 수 있다.
ex) 3차 B-트리라면 최대 2개의 KEY를 가질 수 있다.
- 각 노드는 최소 $\lceil M/2 \rceil$ 개의 자식 노드를 가진다. (루트 노드와 leaf 노드 제외)
ex) 3차 B-트리라면 각 노드는 최소 2개의 자식 노드를 가진다.
- 각 노드는 최소 $\lceil M/2 \rceil - 1$ 개의 키를 가진다. (루트 노드 제외)
ex) 3차 B-트리라면 각 노드는 최소 1개의 키를 가진다.
- internal 노드의 KEY가 x 개라면 자녀 노드의 수는 언제나 $x+1$ 개다.
- > 노드가 최소 하나의 KEY는 가지기 때문에 몇 차 인지에 상관없이 internal 노드는 최소 두 개의 자녀는 가진다.

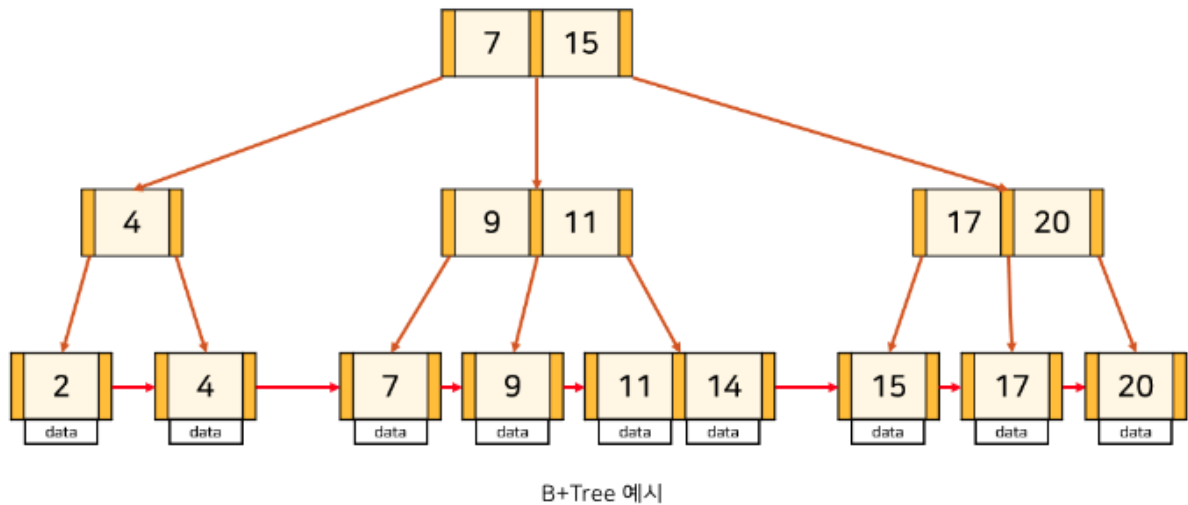
(m 이 정해지면 root노드를 제외하고 internal 노드는 최소 $\lceil M/2 \rceil$ 개의 자녀 노드를 가질 수 있게 된다.)

- 노드에 KEY들은 항상 정렬된 상태로 저장된다

B Tree 삽입, 삭제 과정 : <https://velog.io/@chanyoung1998/B트리>

B+ Tree





인덱스 구현시 B-Tree를 사용하지 않는 이유

B-Tree는 탐색을 위해서 노드를 찾아서 이동해야 한다는 단점이 있다.

즉, 모든 데이터를 순회하는 경우, 모든 노드를 방문해야함

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

장점

- **효율적인 탐색** : B+ 트리는 **균형 잡힌 트리**로서, 모든 리프 노드까지 도달하기 위한 경로의 길이가 동일합니다.(데이터 탐색 시간복잡도 $O(\log N)$)
- **범위탐색 유리** : leaf 노드끼리 **연결 리스트**로 연결되어 있어서 범위 탐색에 매우 유리함
- **순차 액세스 성능** : B+ 트리의 리프 노드는 **연결 리스트**로 구성되어 있으며, **순차 액세스 (Sequential Access)** 를 지원합니다.

단점

- B-tree의 경우 최상 케이스에서는 루트에서 끝날 수 있지만, B+tree는 무조건 leaf 노드 까지 내려가봐야 함
- **메모리 요구량** : B+ 트리는 대부분의 중간 노드를 메모리에 유지해야 하므로, 메모리 요구량이 크다는 단점이 있습니다. 트리의 크기가 커질수록 메모리 사용량도 증가하므로, 메

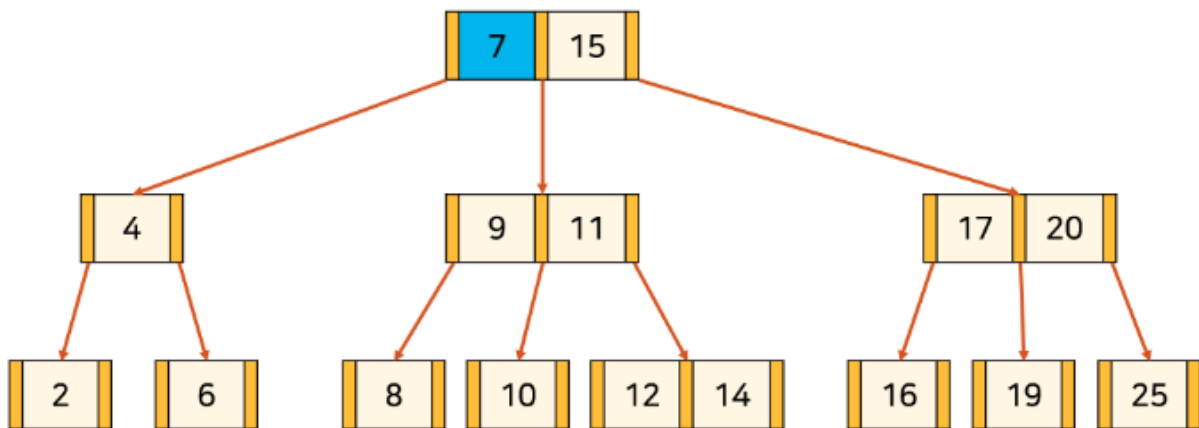
모리 제약이 있는 환경에서는 문제가 될 수 있습니다.

- **공간 사용 비효율성**: B+ 트리는 각 노드마다 포인터와 키 값을 저장해야 합니다. 이로 인해 트리의 크기가 실제 데이터 크기보다 커지며, 디스크 공간의 낭비를 초래할 수 있습니다.

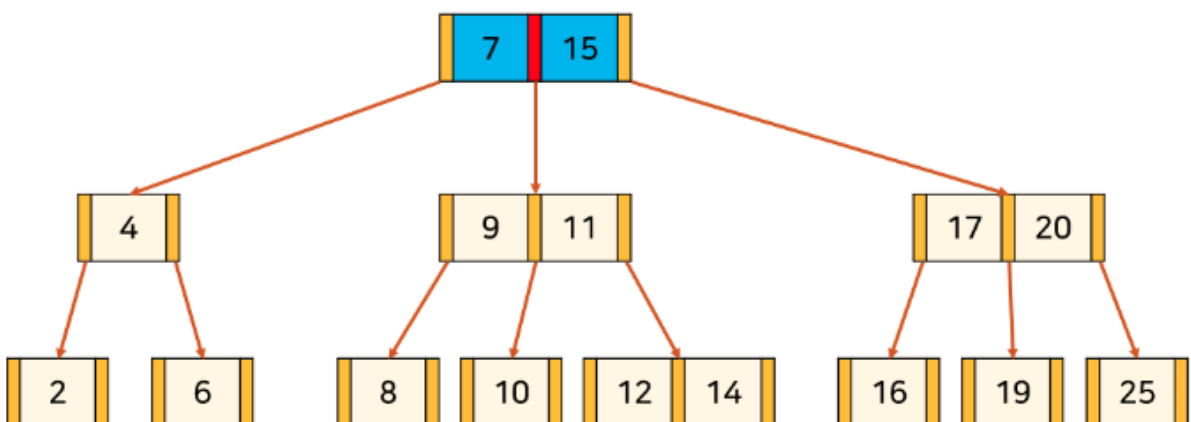
B+ Tree 검색 과정

- 14라는 key를 검색한다고 가정한다.

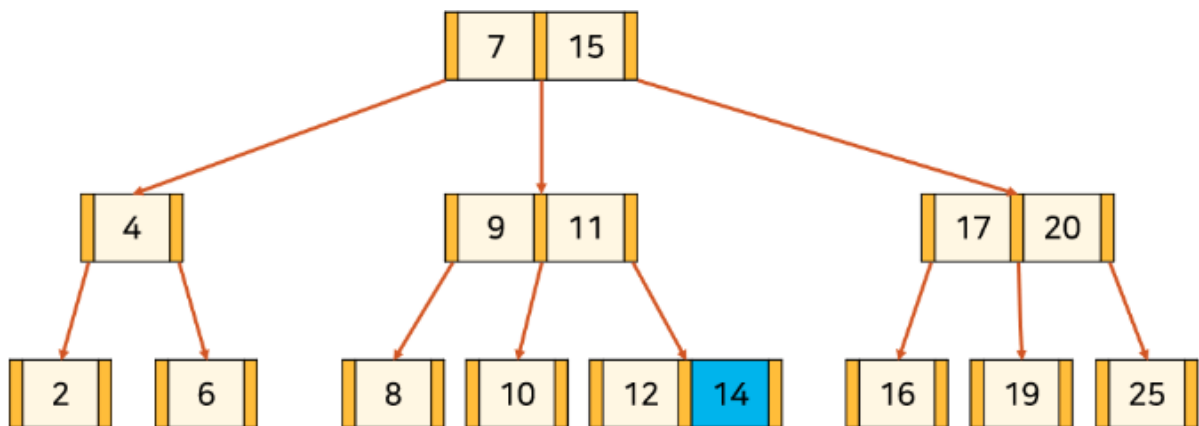
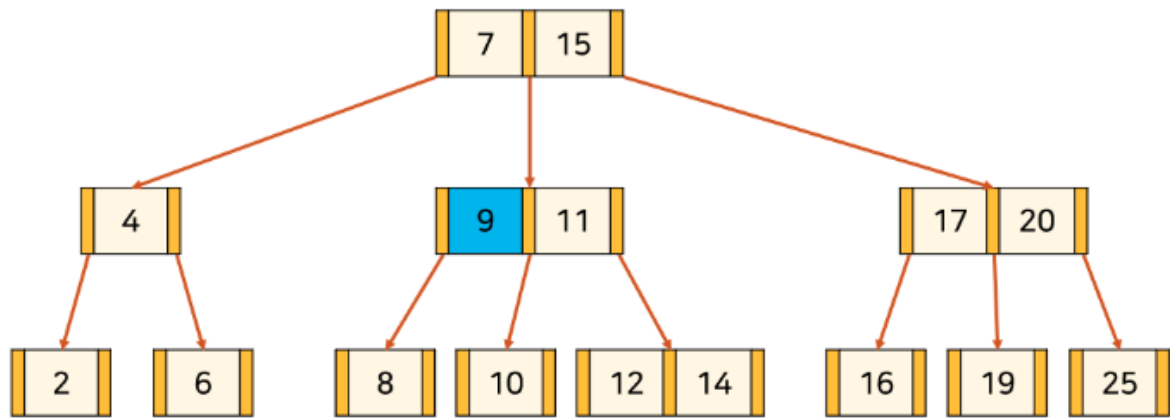
1. root node의 key를 순서대로 확인



2. 7보다 크므로 다음 key 확인. 15보다 작으므로 사이에 있는 포인터가 가리키는 자식으로 이동.



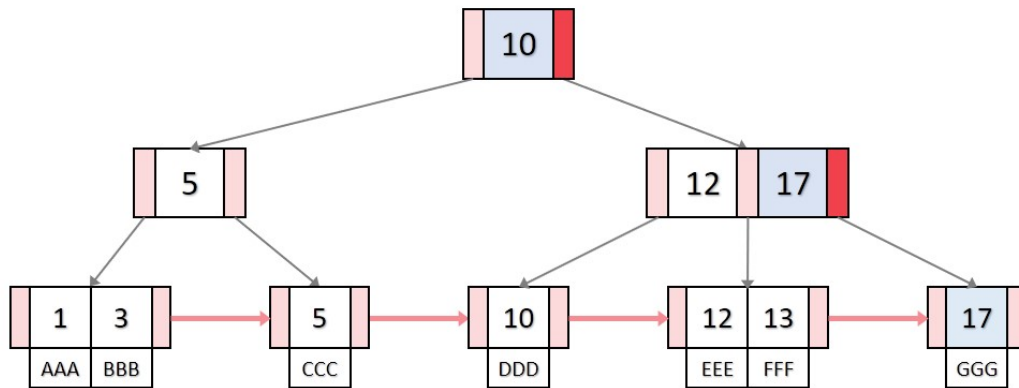
3. 자식으로 이동한 후 key 순서대로 확인



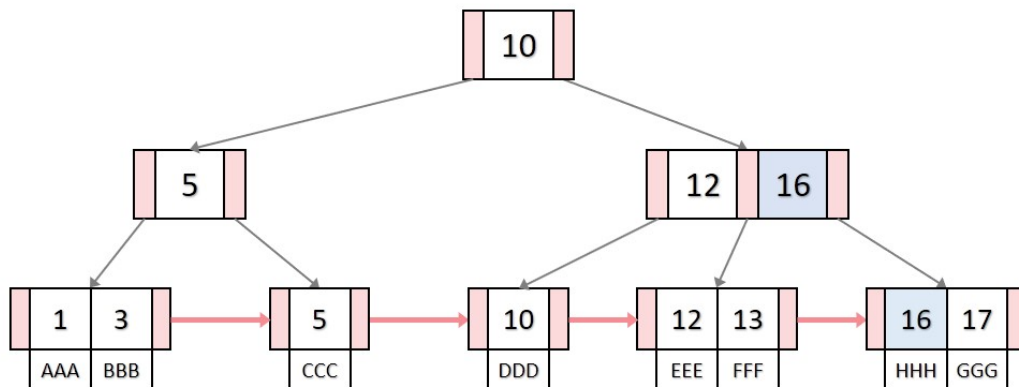
B+ Tree 삽입 과정

Case 2. 분할 x

- 1 (key: 16, data: HHH) 삽입, B-Tree와 동일하게 삽입될 자리 탐색

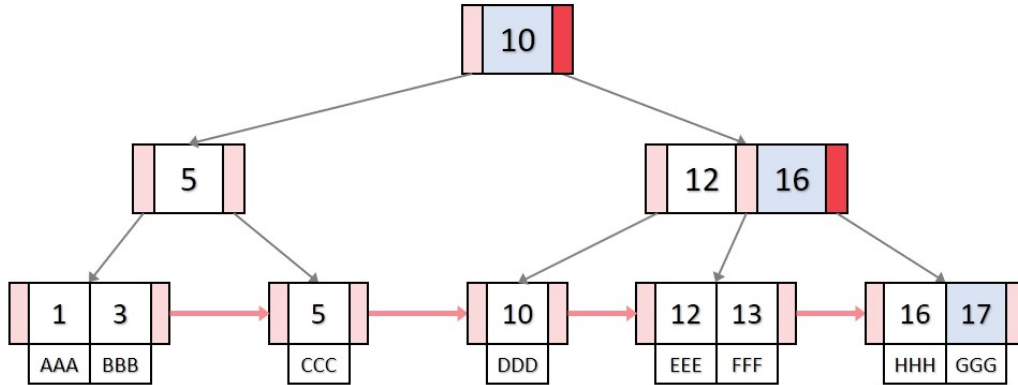


- 2 (key: 16, data: HHH) 삽입 후 부모 key를 바꿈

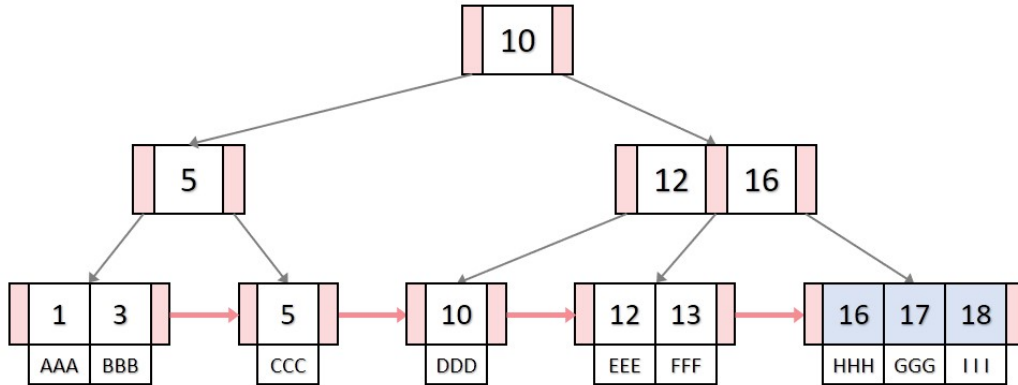


Case 3. 분할이 일어나는 경우

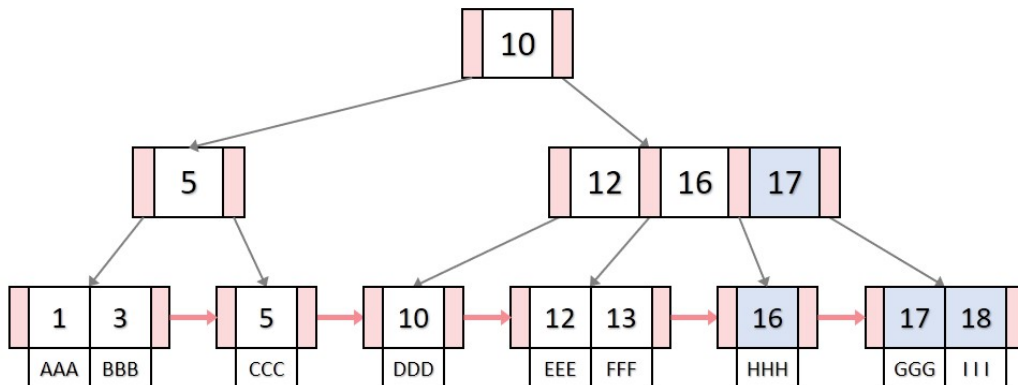
- 1 (key: 18, data: III) 삽입, B-Tree와 동일하게 삽입될 자리 탐색



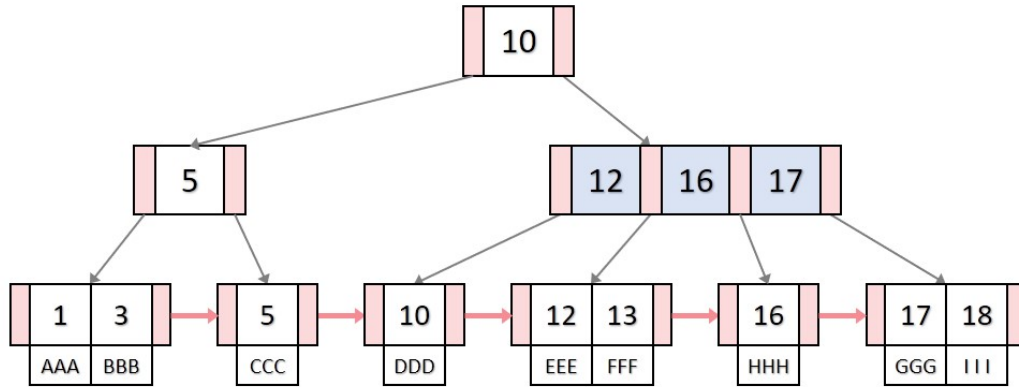
- 2 (key: 16, data: HHH) 삽입. 리프노드가 최대 key 개수를 초과, 분할을 수행



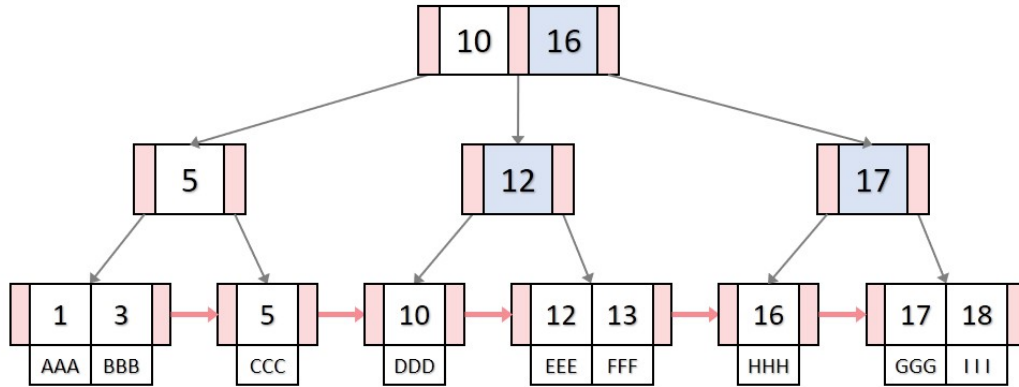
- 3 노드를 2개로 분할 후 오른쪽 노드의 가장 작은 값을 부모 key로 설정, 분할된 노드를 자식노드로 설정
오른쪽 자식을 왼쪽 자식의 오른쪽 노드로 설정하여 연결리스트 형태를 유지



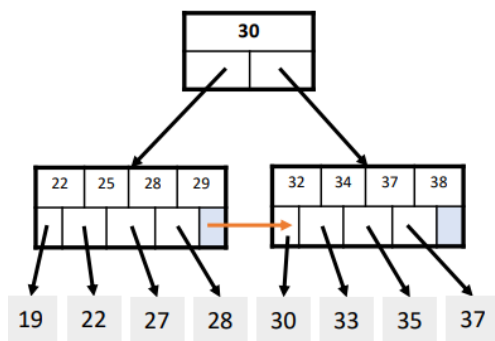
- 4 리프노드가 아닌 노드가 최대 key 개수를 초과, 분할을 수행



- 5 중간 노드를 부모 key로 설정, 분할된 노드를 자식노드로 설정. 삽입 종료.



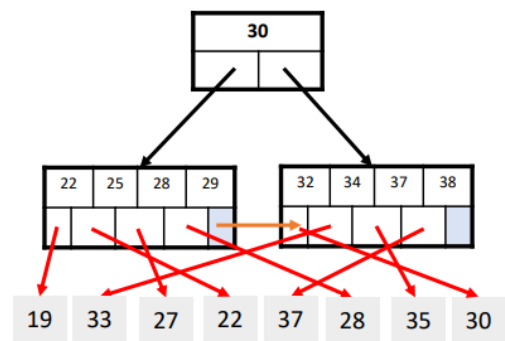
클러스터 인덱스



Clustered

- ➔ A table can have at most 1 clustered index.
- Default: cluster by primary key

Index File



Data file

Unclustered

- ➔ A table can have many unclustered indexes.

- Recall) Sequential IO is much faster than random IO
- For exact search:
 - No difference between clustered / unclustered.
- For range search over R values:
 - Clustered: 1 random IO + R sequential IO
 - Unclustered: R random IO
 - ➔ Unclustered index can be worse than the sequential scan, when searching a range over values.
 - ➔ Then, when is the unclustered index good?
 - When the insertion/deletion is frequent.
 - When indexing is needed to many attributes.
 - When the table is too large to manage the clustered index.

장점

클러스터 인덱스 (순차 탐색과 범위기반 탐색(Range Queries)에 강점이 있다),

단점

언 클러스터 인덱스(테이블이 크거나, 속성이 많을 때, 삽입 삭제가 빈번할 때)

