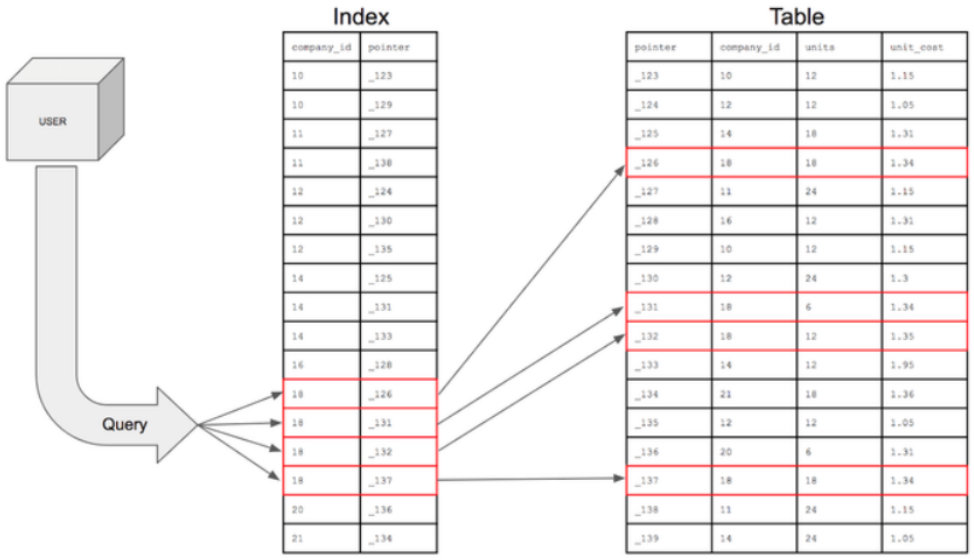
Indexing

♦ 인덱스란?

- index는 추가적인 쓰기 작업과 저장 공간을 활용하여 데이터베이스 테이블의 검색 속도를 향상 시키기 위한 자료구조
- 인덱스 사용 시 `SELECT`, `UPDATE`, `DELETE` 성능도 함께 향상
- 테이블의 특정 Column 에 인덱스를 생성하면, 해당 Column의 데이터를 정렬한 후 별도의 메모리 공간에 데이터의 물리적 주소와 함께 저장된다. Column의 값과 물리적 주소를 (key, value) 의 한 쌍으로 저장한다.



♦ 인덱스의 관리

- DBMS는 인덱스를 향상 정렬된 상태로 유지해야 원하는 값을 빠르게 탐색 가능
- INSERT, DELETE, UPDATE의 명령문이 들어오게 되면 index 관련 연산을 추가로 진행
  - `INSERT` : 새로운 데이터에 대한 인덱스 추가

- `DELETE` : 삭제하는 데이터의 인덱스를 사용하지 않는다는 작업을 진행
- `UPDATE` : 기존의 인덱스를 사용하지 않음 처리하고, 갱신된 데이터에 대한 인덱스 추가

♦ 인덱스의 장단점

- **장점**
  - 테이블을 검색하는 속도와 성능이 향상되어, 시스템 전반적인 부하를 줄일 수 있다.
- **단점**
  - 인덱스를 관리하기 위한 추가 작업이 필요
  - 추가 저장 공간 필요
  - 잘못 사용하는 경우, 오히려 검색 성능 저하
  - 인덱스의 수정이 잦은 경우, 인덱스를 제거하는 것이 아니라 '사용하지 않음' 으로 처리하고 남겨두기 때문에 수정 작업이 많은 경우, 실제 데이터에 비해 인덱스가 과도하게 커지는 문제점이 발생할 수 있다.
  - 성별과 같이 값의 range가 적은 컬럼인 경우, 인덱스를 읽고 나서 다시 많은 데이터를 조회해야 하기 때문에 비효율적이다.

♦ 인덱스를 사용하면 좋은 경우

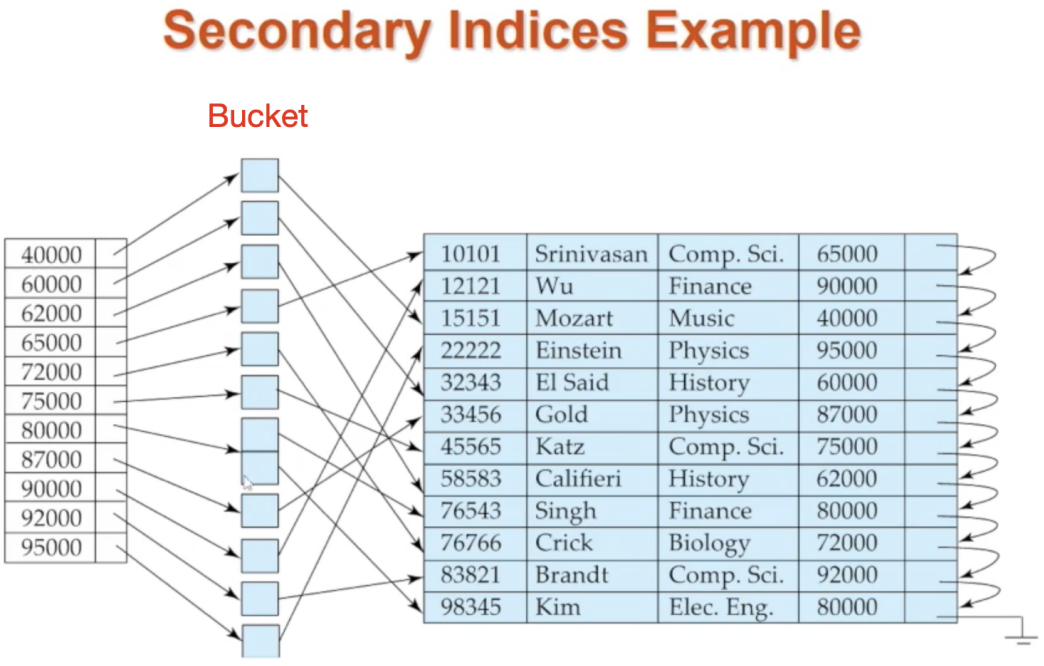
: 인덱스를 효율적으로 사용하기 위해선 데이터의 range가 넓고 중복이 적을수록, 조회가 많거나 정렬된 상태가 유용한 Column에 사용하는 것이 좋다.

- 규모가 큰 테이블
- 삽입, 수정, 삭제 작업이 자주 발생하지 않는 컬럼
- WHERE나 ORDER BY, JOIN 등이 자주 사용되는 컬럼
- 데이터의 중복도가 낮은 컬럼

♦ 인덱스의 분류

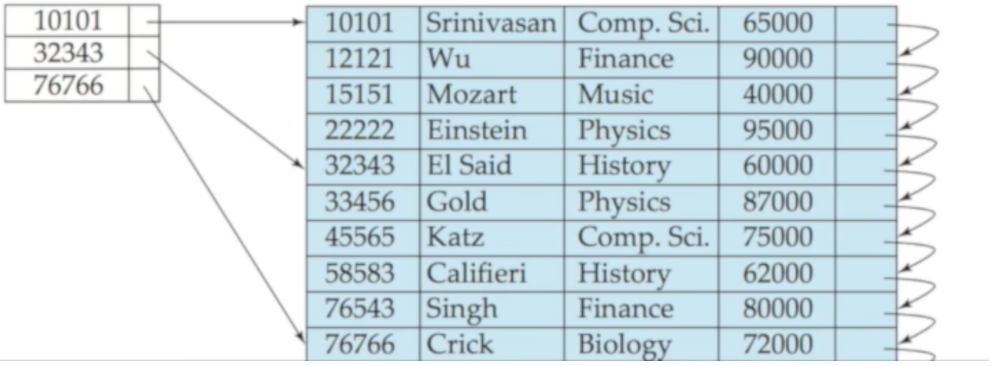
- **인덱스 대분류**
  - `Ordered Index` : search key를 책의 index처럼 정렬된 순서로 저장하는 것
  - `Hash Index` : search key를 해시 함수로 생성하여 저장하는 것
- **정렬 여부에 따른 분류**
  - `Primary Index` : search key의 순서와 실제 파일이 정렬된 순서가 동일한 경우를 말한다. 순차적으로 정렬된 파일의 순서를 정하는 기준이 되는 Attribute를 search key로 사용하는 index 이다. 같은 데이터끼리 뭉쳐지게 되므로 clustering index 라고도 한다. 주로, 기본 키가 search key가 되긴 하지만 꼭 그럴 필요는 없다.
    - **Index-sequential file** : primary index 를 가지고 순차적으로 정렬된 데이터 파일을 말한다. 즉, search key에 해당하는 Attribute에 대해 순차적으로 정렬되어 있다.
  - `Secondary Index` : search key의 순서와 실제 파일이 정렬된 순서가 다른 경우를 말한다. 파일의 순차적 정렬과 다른 정렬을 가지는 search key를 가지는 index이다.

    즉, key의 순서와 레코드 순서가 상관이 없다. non-clustering index 라고도 불린다.



- **커버리지에 따른 분류**
  - `Sparse Index` : 원본 relation의 일부 search key 값만 저장하고 있는 index 파일을 말한다.

    - **Sparse Index**: contains index records for only some search-key values.
      - Applicable when records are sequentially ordered on search-key
    - To locate a record with search-key value *K* we:
      - Find index record with largest search-key value < *K*
      - Search file sequentially starting at the record to which the index record points

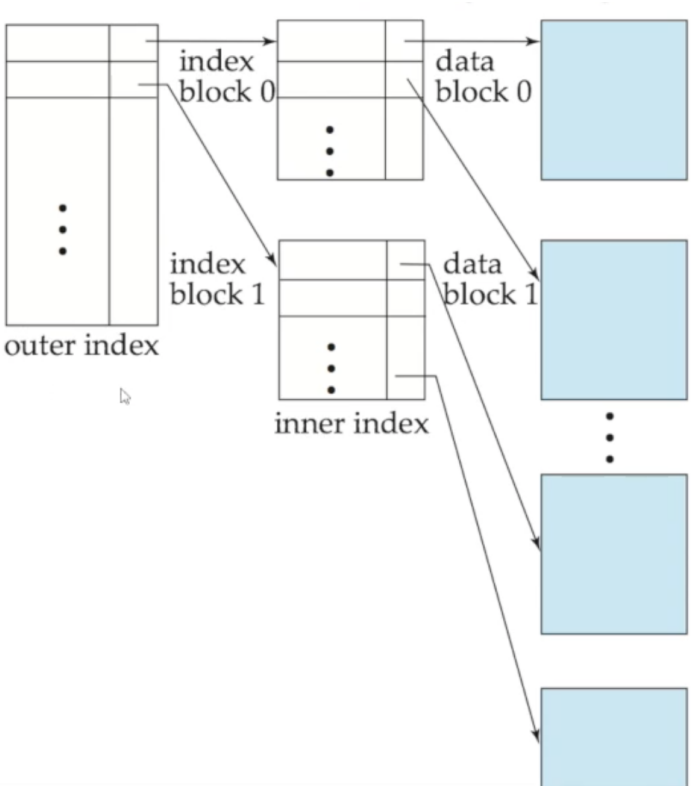


    예로 15151을 찾는다면, 우선 index 파일을 뒤져 10101과 32343 사이에 있는 것을 확인한다. 10101에 저장된 포인터로 이동하여 32343이 나올 때까지 한 칸씩 아래로 이동하며 15151을 찾아나간다.

    이러한 특징으로 sparse Index 파일을 사용하려면 반드시 primary index를 사용하여 원본 relation 정렬되어 있어야 한다.

  - `Dense Index` : 원본 relation에 존재하는 모든 search key attribute 값을 가지고 있는다. primary/ secondary 모두 가능하지만, dense index 파일의 구성이 서로 다르다. 위의 두 번째 사진처럼 primary index 여서 clustering이 되어 있는 상태라면 index 파일에는 각 search key당 포인터를 하나씩만 저장하면 된다. 반면, clustering이 되어 있지 않은 secondary index라면 모든 레코드들에 대한 일대일 포인터를 가지는 bucket이 있어야 한다.

- **Multi-level index**



: 만약 index 파일이 커서 전체를 한 번에 메모리에 유지할 수 없다면, 디스크와의 상호작용이 필요하므로 검색 비용이 많이 들게 된다.

디스크 상호작용을 최소화하기 위해 index 파일 자체를 한번 더 indexing 하는 방법이 있다. 기존 primary index 파일을 순차 파일로 취급하여 inner index로 정의하고, 그에 대한 sparse index를 만들어 inner index로 정의한다.

♦ 인덱스의 자료구조

: 인덱스는 여러 자료구조를 이용해서 구현할 수 있는데, 대표적으로 `해시 테이블`과 `B+Tree`가 있다.

1. **해시 테이블 (Hash Table)**

   : 해시 테이블은 key와 value를 한 쌍으로 데이터를 저장하는 자료구조이다. `(key, value)`로 쌍을 표현하며, key값을 이용해 대응되는 value 값을 구하는 방식이다. 해시 충돌이라는 변수가 존재하지만 평균적으로 **O(1)**의 매우 빠른 시간 만에 원하는 데이터를 탐색할 수 있는 구조이다.

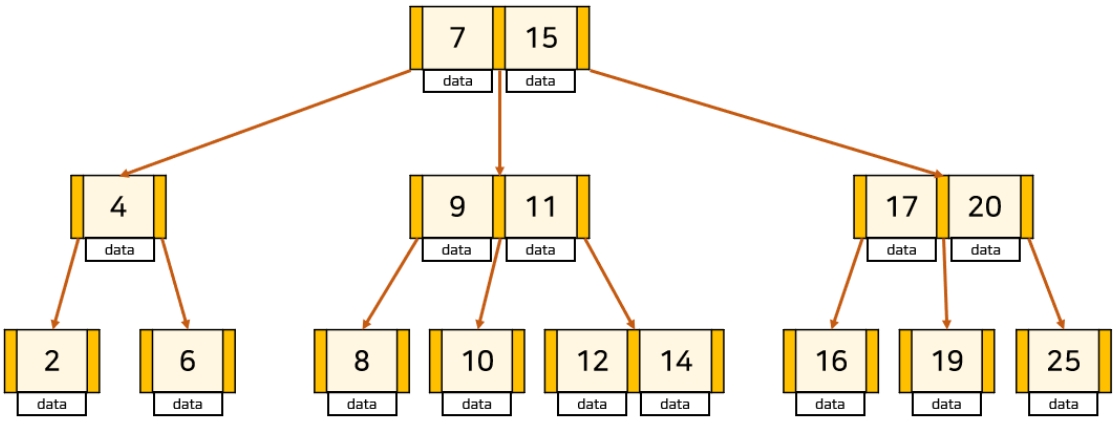
해시 테이블을 이용한다면 인덱스는 **(key, value) = (컬럼의 값, 데이터의 위치)**로 구현하는데, 해시 테이블은 실제로 인덱스에서 잘 사용되지 않는다.
그 이유는, 해시 테이블은
등호 (=) 연산에 최적화되어 있기 때문이다.
데이터베이스에서는
부등호 연산(<. >)이 자주 사용되는데, 해시 테이블 내의 데이터들은 정렬되어 있지 않으므로 특정 기준보다 크거나 작은 값을 빠른 시간 내에 찾을 수 없다.

2. **B+Tree**

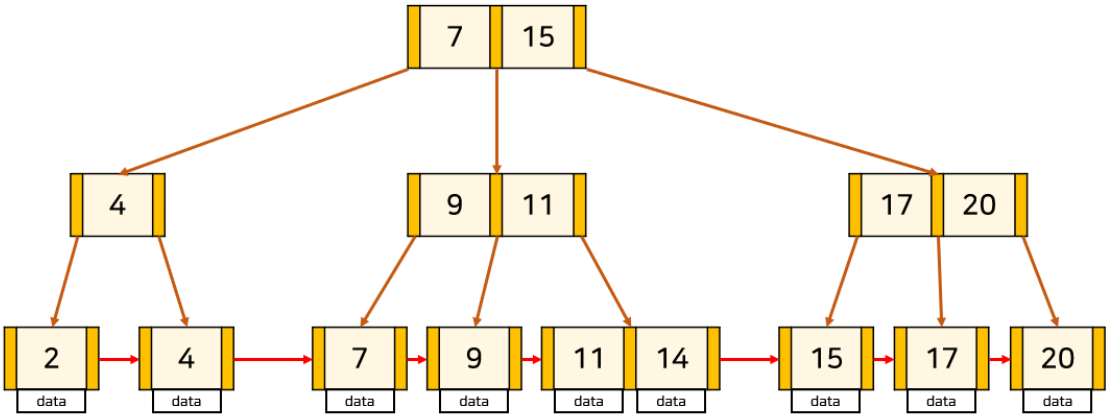
: 기존의 B-Tree는 어느 한 데이터의 검색은 효율적이지만, 모든 데이터를 **한 번 순회**하는 데에는 **트리의 모든 노드를 방문**해야 하므로 비효율적이다. 이러한 B-Tree (Indexed-sequential file) 의 단점을 개선 시킨 자료구조가 B+Tree이다.

B+Tree는 오직 leaf 노드에만 데이터를 저장하고, leaf 노드가 아닌 노드에는 자식 포인터만 저장한다. 그리고 leaf 노드끼리는 Linked list로 연결되어 있다.

또, B+Tree에서는 반드시 leaf 노드에만 데이터가 저장되기 때문에 중간 노드에서 key를 올바르게 찾아가기 위해서 key가 중복될 수 있다.



B-Tree 예시



B+Tree 예시

B+Tree의 장점

1. leaf 노드를 제외하고 데이터를 저장하지 않기 때문에 메모리를 더 확보할 수 있다. 따라서, 하나의 노드에 더 많은 포인터를 가질 수 있기 때문에 트리의 높이가 더 낮아지므로 검색 속도를 높일 수 있다.
2. Full scan을 하는 경우 B+Tree는 leaf 노드에만 데이터가 저장되어 있고, leaf 노드끼리 Linked list로 연결되어 있기 때문에 선형 시간이 소모된다. 반면, B-Tree는 모든 노드를 확인해야 한다.
3. 삽입/ 삭제가 발생하더라도 상대적으로 적은 양의 비용으로 재구성이 가능하다.

반면, B-Tree의 경우 최상의 경우 특정 key 를 root 노드에서 찾을 수 있지만, B+Tree의 경우 반드시 특정 key에 접근하기 위해서 leaf 노드까지 가야 하는 단점이 있다.

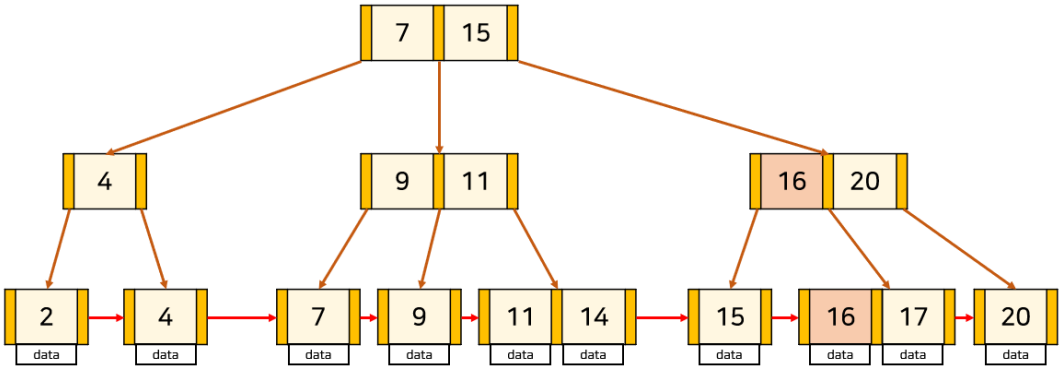
인덱스에서 B-Tree 대신 주로 B+Tree 를 사용하는 이유

: 해시 테이블에서 언급했듯이 인덱스 컬럼은 부등호를 이용한 순차 검색 연산이 자주 발생한다. 따라서, B+Tree의 LinkedList를 이용하면 순차 검색을 효율적으로 할 수 있게 된다.

1) 삽입

- **key의 수가 최대보다 적은 leaf node에 삽입하는 경우**

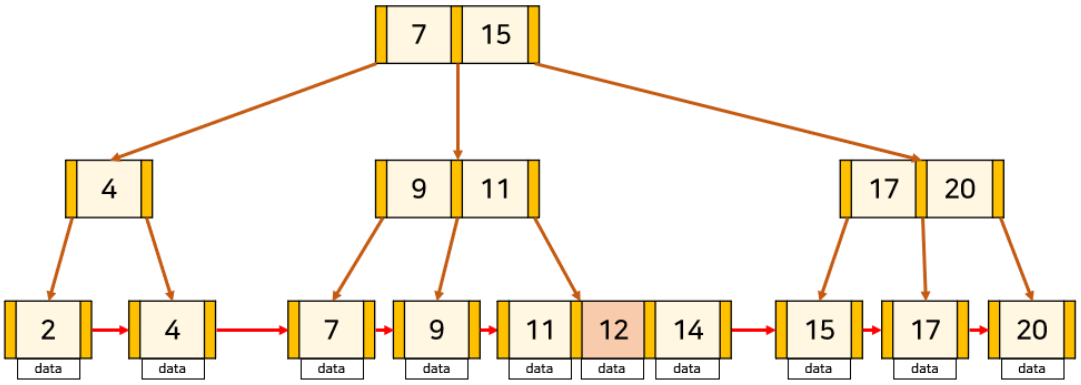
: 해당 node의 가장 앞이 아닌 곳에 삽입되는 경우는 단순히 삽입해 주면 된다. 하지만, leaf node의 가장 앞에 삽입 되는 경우는, 해당 node를 가리키는 부모 node의 포인터의 오른쪽에 위치한 key를 K로 바꿔준다. 그리고 leaf node 끼리는 Linked list로 이어줘야 하므로 삽입된 key에 LinkedList로 연결한다.

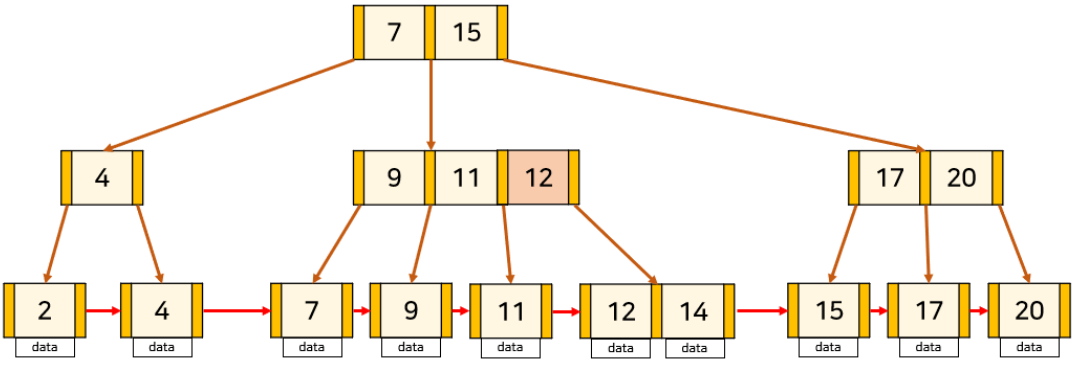


- **key의 수가 최대인 leaf node에 삽입하는 경우**

  : key의 수가 최대이므로 삽입하는 경우 분할을 해주어야 한다. 만약, 중간 node에서 분할이 일어나는 경우는 B-Tree와 동일하게 해주면 된다.

  leaf node에서 분할이 일어나는 경우는 중간 key를 부모 node로 올려주는데 이때, 오른쪽 node에 중간 key를 붙여 분할한다. 그리고 분할된 두 node를 Linked List로 연결해준다.

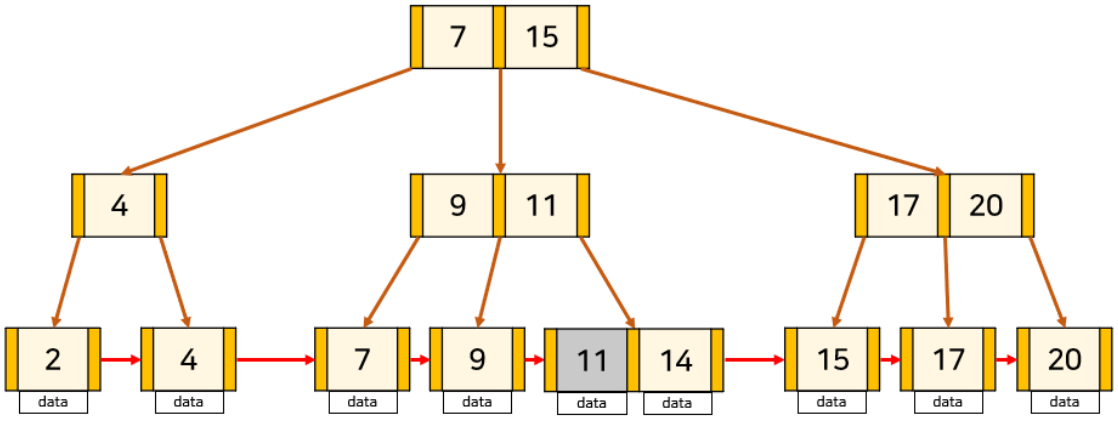


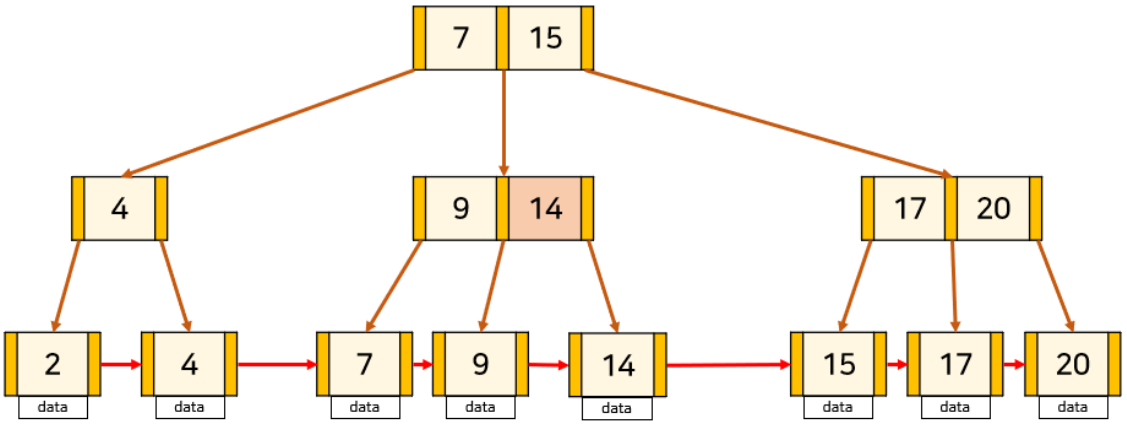


2) 삭제

- **삭제할 key가 leaf node의 가장 앞에 있지 않은 경우**
  : B-Tree와 동일한 방법으로 삭제
- **삭제할 key가 leaf node의 가장 앞에 위치한 경우**

  : 이 경우는 leaf node가 아닌 node에 key가 중복해서 존재한다. 따라서 해당 key를 노드보다 오른쪽에 있으면 가장 작은 값으로 바꾸어 주어야 한다.





- 참고자료

[CS] 데이터베이스 - 5. Normalization / Indexing / Transaction

Reference내용전반: edwith정규화란, 하나의 relation에 하나의 의미만 존재할 수 있도록 relation을 분해해 가는 과정입니다. 이 과정에서 데이터의 중복이 있을 경우 이상 현상이 발생할 가능성이 있는데, 이런 불필요한 중복을

https://velog.io/@yohanblessyou/CS-데이터베이스-5.-Normalization-Indexing-Transaction

velog