# 7장 인덱스된 순차 파일

**간단한 설명** : 순차 파일 + Index

**INDEX** : 임의 접근에 효율적

인터넷 검색? ----- 대부분이 Random Access

### 순차파일 vs 인덱스 파일

1. 순차파일
   * 순차 검색에 특화
2. 인덱스 파일 (AVL Tree, B-Tree)
   * 임의 검색에 특화
   * in-order 방식으로 순차 검색이 가능 but 키의 개수가 많아지면 Stack overflow 가 발생해 실질적인 순차 검색이 불가능하다.
3. 인덱스된 순차 파일
   * 순차 검색과 임의 검색 모두 효율적으로 가능

### 인덱스된 순차 화일의 구조

< 구조 = 순차구조 + 인덱스 >

* 순차 데이터 파일 : 전체 레코드를 순차 접근할 때
  + 데이터 블록으로 구성 ... 서로 Linked List 로 연결되어 있다.
  + 순차적으로 저장된 데이터 레코드와 자유공간이 존재함
  + 레코드의 추가를 위해 남겨둔 여분의 공간을 **자유 공간**이라고 한다.
  + **자유 공간**은 화일 생성시 각 블록에 만들어진다.
* 인덱스 파일 : 한 특정 레코드를 직접 접근할 때
  + 인덱스 블록으로 구성
  + 트리 구조
  + 최상위 레벨의 인덱스 블록을 **마스터 인덱스**라고 한다.

#### 삽입 방법

1. 자유 공간에 우선 삽입한다.
2. 자유 공간이 꽉 찬 경우 다음과 같이 분할을 한다.
   1. 반은 그대로 두고, 반은 새로운 데이터 블록에 넣는다.
   2. 기존 반을 가지고 있는 데이터 블록이 새로은 데이터 블록을 가르키게 한다.
   3. 새로운 데이터 블록에서 가장 큰 키를 인덱스 블록에 추가한다.
   4. 꽉 차면 상위 인덱스로 전파한다.

#### 인덱스된 순차 파일의 문제점

* **데이터 파일**이 <u>순차적으로 정렬</u>되어 있어야 한다는 점이 <u>매우 큰 제약 사항</u>이다.
* 데이터 블록들이 정렬되어 연결되어 있어야 한다.

# B+-Tree

#### B+-Tree 란?

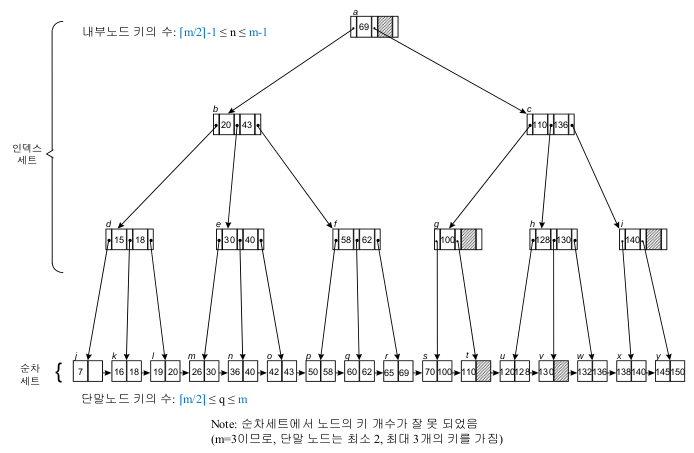
* 직접 접근 및 순차 접근 모두의 속도 향상을 위해 사용한다
  + 직접 접근 및순차 접근 모두가 필요한 응용에 적합하다.
* Knuth

#### B+-Tree의 구조

인덱스된 순차파일의 **인덱스 파일**이두 부분으로 나뉜다.

* 인덱스 파일
  1. Index Set
     + Tree의 내부 노드이다.
     + key값만 존재한다.
     + Leaf에 있는 key들에 대한 경로 정보만 제공한다.
  2. Sequence Set
     + Tree의 Leaf 노드이다.
     + Key값과Record Address가 같이 존재한다.
     + Leaf Node 들은 순차적으로 연결되어있다.
* 데이터 파일
  + 정렬되어 연결되지 않음 ( 레코드가 랜덤하게 저장된다 )

#### 다음은 차수가 3인 B+-Tree 이다.



#### 차수가 m인 B+-Tree 의 노드 구조

B+-Tree의 노드 구조는 B-Tree의 노드 구조와비슷하다.

* 내부 노드 구조
  + 내부 노드 내의 키 값의 수 n : [m/2]-1 <u><</u> n <u><</u> m-1
  + 키 값 : Ki (1 <u><</u> i <u><</u> n)
  + 서브트리에 대한 포인터 : Pi ( 0 <u><</u> i <u><</u> n )
* 리프 노드 구조
  + 리프 노드 내의 키 값의 수 q : [m/2] <u><</u> q <u><</u> m
  + 키 값 : Ki (1 <u><</u> i <u><</u> q )
  + 키 값으로 Ki를 가진 레코드에 대한 포인터 : Ai (1 <u><</u> i <u><</u> q )
  + Pnext : 순차 set의 다음 블록에 대한 포인터
* B-Tree와의 구조 차이?
  + 인덱스 세트와 순차 세트의 구분이 있다
    - 인덱스 세트는 키 값만 저장한다. // 데이터를 찾는 경로로 사용
    - 순차 세트는 키 값과 레코드 포인터를 저장한다. // 실제 데이터를 접근하는데 사용
  + Pnext는 순차 세트에만 있다.
    - 모든 순차 세트의 노두가 Linked List로 연결되어 있어, 효율적인 순차 접근이 가능하다.
  + Pi는 인덱스 세트에만 있다.
    - 분기를 위한 포인터이다.

#### B+-Tree 연산

* 검색
  + Random Access
    - Index Set 검색 ... MST Search Algorithm
  + Sequential Access
    - Sequential Set (Leaf) 검색
* 삽입
  + B-Tree와 유사
  + 오버플로우(분할) -> 부모 노드, 분열노드 모두 키값이 존재한다.
  + 리프 노드 분할 시
    - 중간 키 값의 **복사본**이 부모(Index Node)로 올라가 삽입된다.
  + 인덱스 노드 분할 시
    - B-Tree와 똑같은 알고리즘을 수행하고, 중간 키는 부모 노드로 올라간다.
* 삭제
  + Underflow 가 발생하지 않는 경우
    - Leaf 에서만 삭제
  + Underflow가 발생하는 경우
    - 재분배를 해야함
      * Index의 키값도 삭제한다
  + B-Tree와 유사하나, 키 값의 삭제는 <u>리프 노드에서만 수행</u>한다.
  + 인덱스 세트의 키 값을 굳이 삭제할 필요가 없다.
    - 삭제하지 않고, 분리자(seperator)를 이용하여 다른 키 값을 탐색하는데이용한다.
  + Underflow가 되는 키 개수의 기준
    - 내부노드 : [m/2]-1 <u><</u> n <u><</u> m-1
    - 리프노드 : [m/2] <u><</u> q <u><</u> m

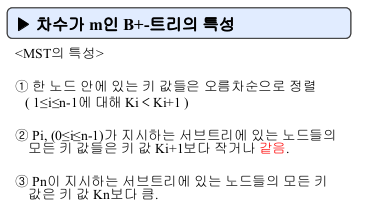
#### Range Search ( 범위 검색 )

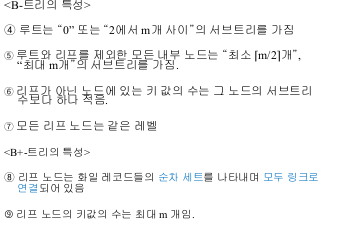
* 범위 검색 = 임의 검색 + 순차검색

#### B+-Tree의 성능

* 특정 키 값의 검색
  + 단점 : 검색이 항상 리프 노드까지 내려가야 종료된다.
    - 검색하고자 하는 값이 인덱스 세트에서 발견되더라도 리프 노드까지 내려가야 실제 레코드의 포인터를 알 수 있다.
  + 장점 :인덱스 세트 노드는 포인터를 저장하지 않으므로, 트리 공간이 절약된다.
* 순차 검색
  + 연결 리스트를 순회한다 ... 효율적이다
* 그런 의미에서 직접 처리와 순차 처리를 모두 필요로 하는 응용에서 효율적이다.

#### 이쯤에서 살펴보는 각 트리의 특징





B트리에서 Key는 디스크에 대한 포인터 역할을 했다.

B+트리에서 Key는 단순하게 교통 신호 역할을 한다.

