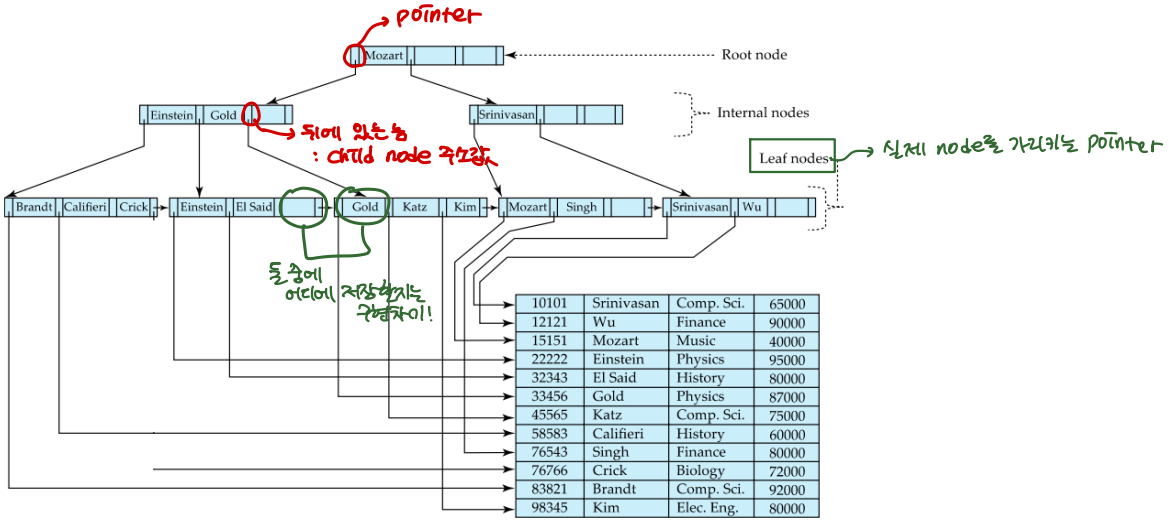
17. Indexing – B+-tree

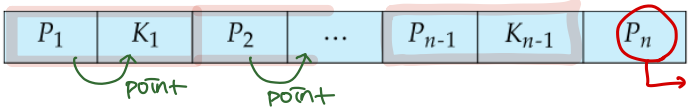
▼ B+ tree

leaf node → linked list로 이루어져 있으며 실제 데이터가 leaf node에만 저장됨



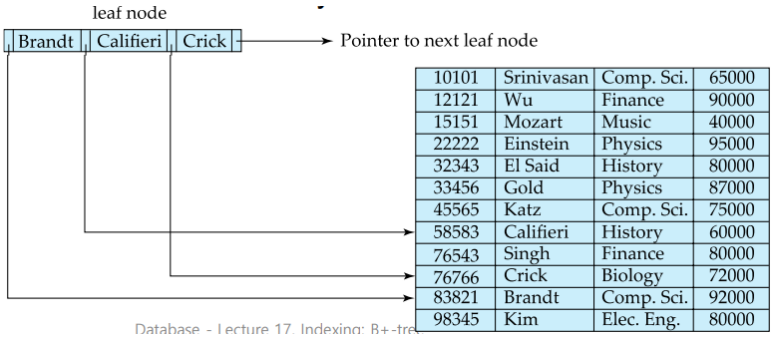
- 최대 n개의 child node를 가질 수 있는 rooted tree
  1. 다 같은 길이의 leaf node
  2. 각 노드는 $\lceil n/2 \rceil$ ~ n 개의 child node
  3. leaf node → 기본적으로 $\lceil (n-1)/2 \rceil$ ~ n-1개의 value를 가져야 함
  4. root가 leaf인 경우 → 0 ~ (n-1)개의 value
  5. root가 leaf가 아닌 경우 → 적어도 2개의 children

▼ node structure



- Ki: search-key value
  - K1 < K2 < K3 … Kn-1 ⇒ 모두 정렬되어 있음
- Pi: pointer to children 혹은 pointer to records(bucket of records)

① Leaf nodes



$(0 < i < n)$

- Pi : Ki을 point
- Pn : 다음 node point
- i<j → search-key value : Li≤Lj

② Non-Leaf nodes

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

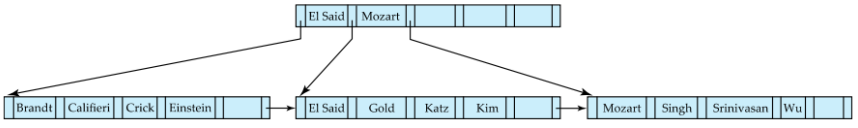
- leaf node의 multi-level sparse index 형식 → 모든 search key value로만 정렬됨

① i = 1 : P1에 있는 subtree에 있는 모든 값 ≤ K1 → 이후에 있는 값 다 작음

② 2 ≤ i ≤ n-1 : Ki-1 ≤ Pi '' ≤ Ki
↳ Pi에 있는 모든 subtree

③ i = n : Pn '' ≤ Kn-1

- example → instruction file, n=6



- Leaf node : 3개까지 value 가져야 함 $(\frac{6-1}{2} < 3)$
- Non-Leaf node : 3~6개의 children 가져야 함
- Root : 반드시 적어도 2개의 children

- Observation

① pointer에 의해 연결되어 있기에 물리적으로 붙어있음

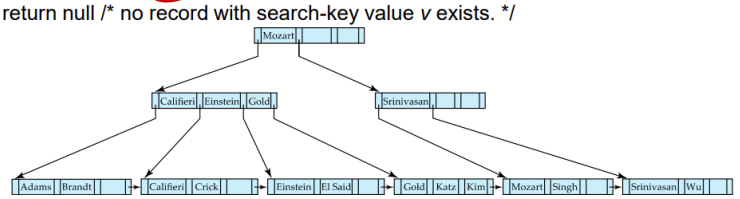
② non-leaf level → 계층 구조 + sparse index

③ 상대적으로 적은 수의 level 포함

⇒ search cost = tree height = $\log_{\lceil n/2 \rceil}(K)$

▼ Queries on B+ tree

```
function find(v)
1.  C=root
2.  while (C is not a leaf node)
1.      Let i be least number s.t. V ≤ K_i.
2.      if there is no such number i then
3.          Set C = last non-null pointer in C
4.      else if (v = C.K_i ) Set C = P_{i+1}
5.      else set C = C.P_i
3.  if for some i, K_i = V then return C.P_i
4.  else return null /* no record with search-key value v exists. */
```



- range queries 가능 → 범위를 설정해서 그 범위에 맞는 record 찾기
- node
  - 보통 4KB→ disk block size에 맞춤
  - 약 100개의 index entry가 들어있음(약 40byte크기)
  - search key value = 100만개 → 4개의 node만 접근하면 가능
    ↔ balanced binary tree : 20개의 node만 접근하면 가능

⇒ disk I/O 최적화

- Non-Unique Keys
  - search key a가 unique하지 않은 경우 → composite key 만들어서 사용
  - composite key → (a, Ap)
    - Ap: 기본 키 혹은 record ID 혹은 다른...거 → unique 보장
    - a : (v, -∞) to (v, ∞)

⇒ but, 실제 record를 가져오려면 더 많은 I/O 작업이 필요함

▼ Insertion

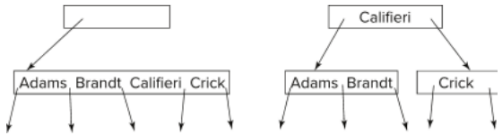
ptr: record의 pointer

k: record의 search key

1. search
   i. leaf node에 공간이 있다면 → insert (k,ptr)
   ii. leaf node에 공간이 없다면 → split + update parent nodes

2. split: node의 data 개수 = n , n개의 pa자가 정렬되어 있어야 함

↳ non-leaf example



i. leaf node 삽입하고 생각!

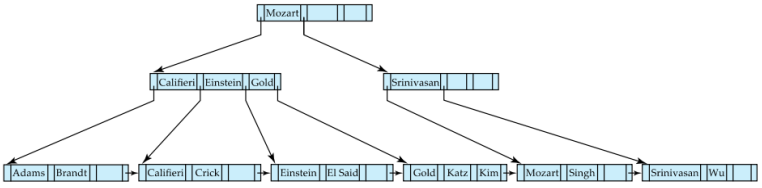
- 기존 node : $1 \cup \lceil \frac{n}{2} \rceil$
- new node : $(\lceil \frac{n}{2} \rceil + 1) \cup n$
- parent node에 첫 번째 node 삽입
  - 만약 parent node가 꽉 찼다면 다시 split

↳ non-leaf이기 때문에 다시 split 해야함.

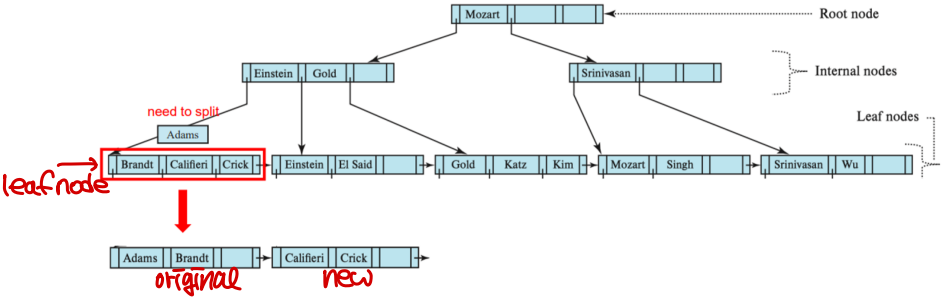
ii. non leaf node

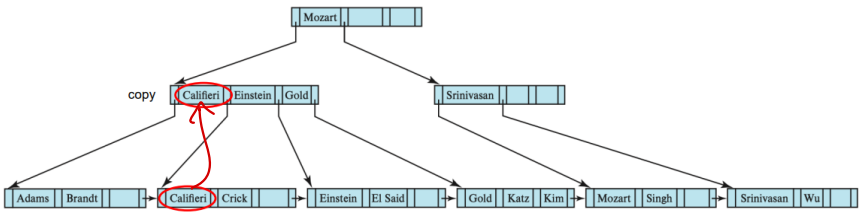
- 기존 node : $1 \cup (\lceil \frac{n}{2} \rceil - 1)$
- new node : $(\lceil \frac{n}{2} \rceil + 1) \cup n$
- parent node에 n/2번째 data 삽입

- example

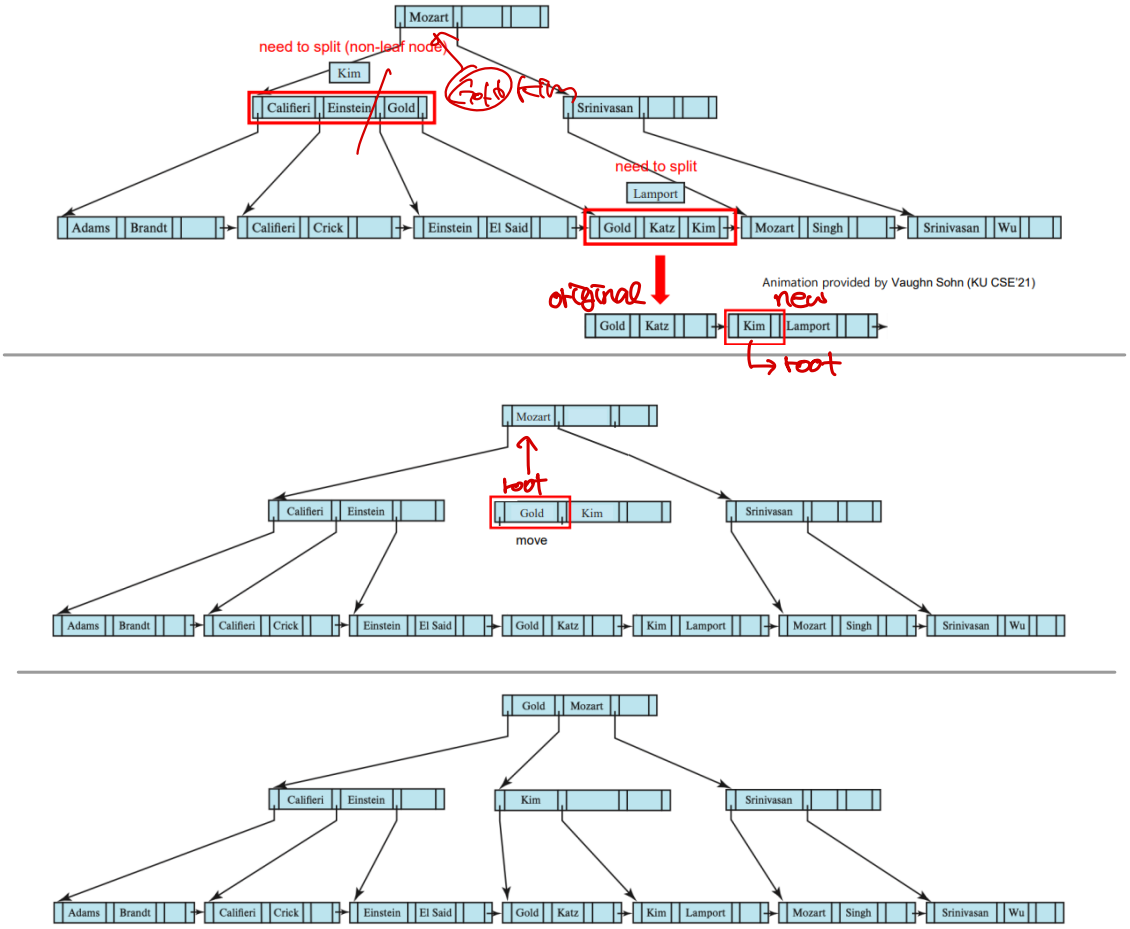


1. left split : insert 'Adams'





2. insert ‘Lamport’



▼ Deletion

ptr : record의 pointer

k : record의 search key

1. leaf node에서 (k, ptr) 제거
2. case
   i. 애초에 leaf node에 너무 적은 entry → sibiling node not fit

   ⇒ merge sibilings
   - 두 node의 모든 key → 왼쪽 node에 삽입 + 다른 node 삭제
   - 삭제된 node를 가리키는 (k-1, p)를 parent node에서 재귀적으로 삭제

   ii. 삭제로 인해 leaf node에 너무 적은 entry → sibiling node not fit

   ⇒ redistribute pointers

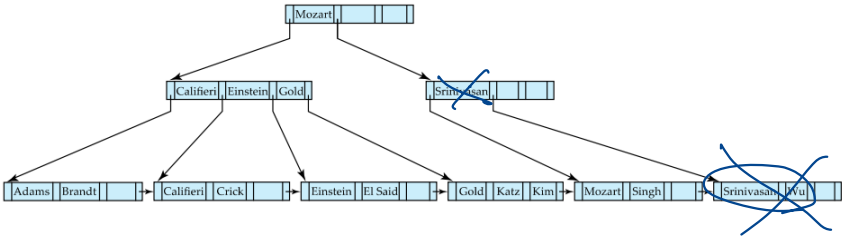
- 최소 entry 수 이상이 되도록 sibiling node의 entry 재배치
- parent node의 해당 key update

iii. deletion은 n/2개 이상의 pointer를 가진 node를 찾을 때까지만 가능

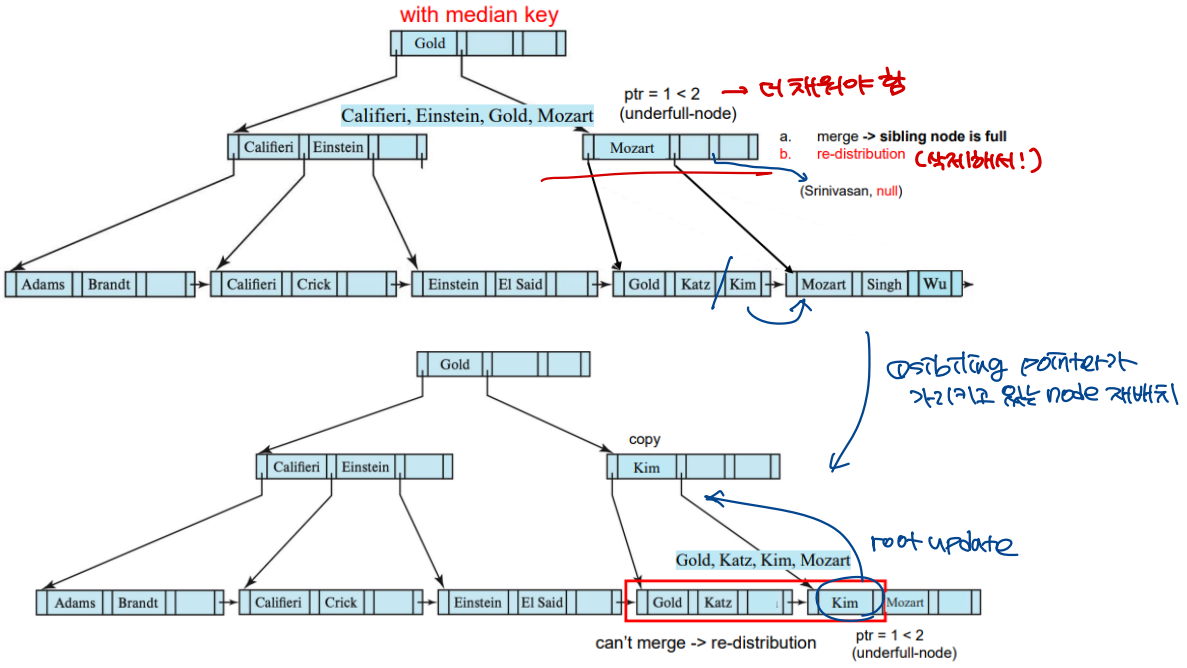
iv. root node에 삭제 후 하나의 pointer만 남은 경우

→ root node 삭제 + child가 새로운 root가 됨

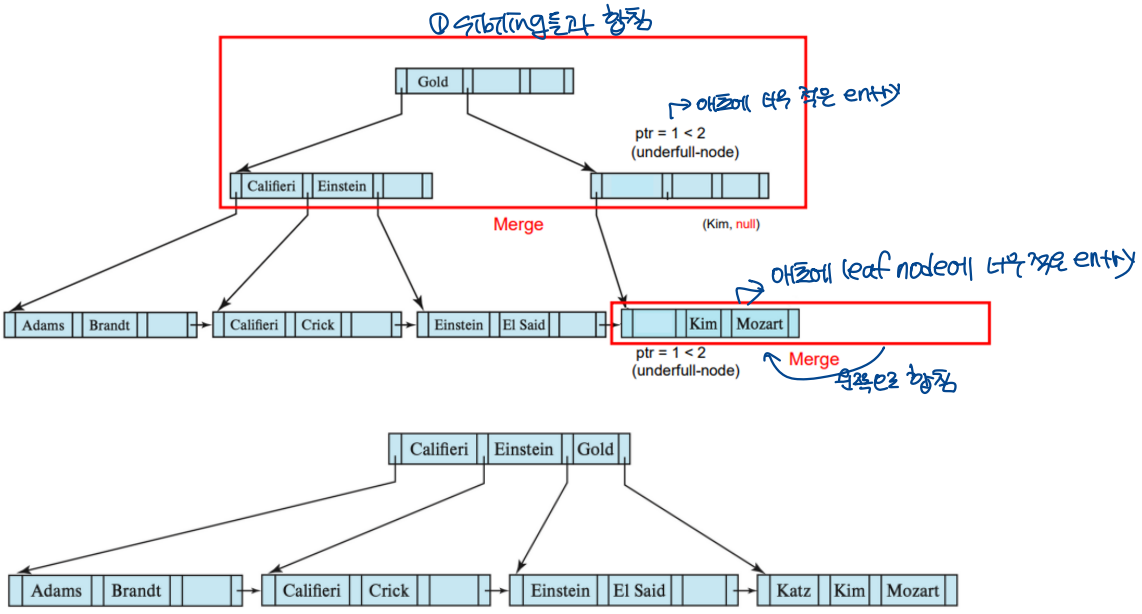
- example



1. Deleting “Srinivasan” → **merging**



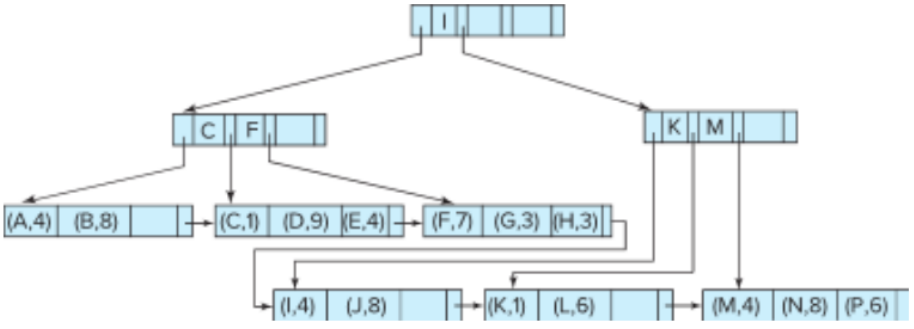
2. Delete "Gold"



▼ Complexity of Updates

- Cost
  - insertion, deletion의 I/O 작업 cost → tree의 height와 관련있음
  - K개의 entry + max fanout of n →
- 실제로는 i/o 작업 수가 적음
  - internal node가 buffer에 있는 경향이 있음
  - 분할, 병합이 드물며, 대부분 leaf node에만 영향 미침
- insertion에 의한 평균 node occupancy
  - 2/3 ln 2 (random)
  - 1/2 (sorted order)

▼ File organization



- leaf nodes → pointer 대신에 record 저장
  - insertion, deletion, update할 때에도 data record를 clustered하게 keep하도록 도와줌

- record가 pointer보다 더 많은 공간 차지하기에 good space utilization이 중요
- space utilization을 향상시키기 위해서는 split과 merge를 하는 동안 더 많은 sibling node들이 redistribution해야 함

▼ Indexing

- Record relocation + secondary index
  - record가 지워지면 record pointer를 저장하고 있는모든 secondary index들이 update 되어야 함
  - node split → 매우 비쌈

⇒ record pointer 대신 search_key 이용 (in secondary index)

- indexing Strings
  - key의 길이가 생각보다 길어질 수 있음
    - split하는데 space 사용

↳ prefix compression

- inter node의 key value가 full key의 prefix 가능
  - sub string만 저장해서 압축해서 표현

- Bulk Loading and Botton-Up Build
  - entry 당 여러 개의 I/O를 필요로 할 경우 너무 비효율적임

    ⇒ bulk loading : key의 정보 모두 다 알고 있어야 함

    ↳ 한 번에 많이 load ⇒ 너무 비효율적

  - solution
    1. 정렬 먼저 하기 → 정렬된 순서대로 insert 하기
    2. **Bottom-up B+tree construction**
       - sort entry → layer마다 tree 생성

(정렬된 순서로 계속 삽입)

↳ leaf level에서 시작

+) B tree와의 차이점
B+tree의 장점

+) Bottum up, topdown build.