**11- Chapter 11: Indexing & Hashing**

**Basic Concepts**

- Search Key

파일에서 레코드들을 찾기 위한 attribute들.

- Index file

Records로 이루어져 있음. (index entry라고 불린다)



Index file들은 보통 파일보다 훨씬 작음.

인덱스는 두 개의 기본적인 종류가 있음

- Ordered Indices: search key들이 정렬된 순서로 저장됨. (tree index)

- Hash Indices: search key들이 hash function을 사용해서 bucket들 사이에 균등하게 분배됨.

**Index Evaluation Metrics**

Index가 어떤 query를 support하느냐에 따라 다름.

- Point queries: search key를 위한 특정 값. (두 인덱스 방법 모두 지원)

- Range queries: 특정 범위에 속하는 search key 값. (hash index는 지원 X)

Time: Access time, insertion time, deletion time. Insert와 delete에서 약간 희생을 하고 access에서 크게 이득을 봄.

저장 공간의 overhead가 있음.

**Ordered Indices**

- Primary Index

Search key가 파일의 순서로 특정된 index.

**Clustering index라고도 불림.**

Primary index의 search key는 항상은 아니지만 보통 primary key임.

- Secondary Index

Search key가 file의 순서와 다르게 특정되어 있는 index

Non-clustering index라고도 불림.

포인터가 더욱 중요한 역할을 함.

- Index-sequential file

정렬된 Sequential file+ primary index.

일반적으로 쓰이는 구조

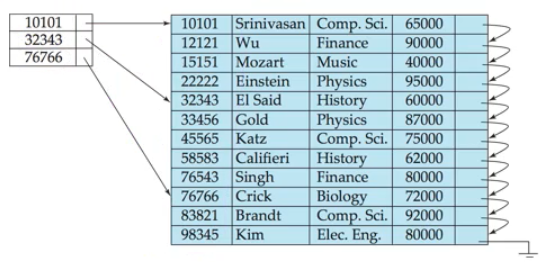
**Dense Index Files**

- Dense Index

모든 index가 search-key value에 나타남. 항상 primary index는 아님!

- Sparse Index

일부 search-key value들만 index record를 담고 있음.

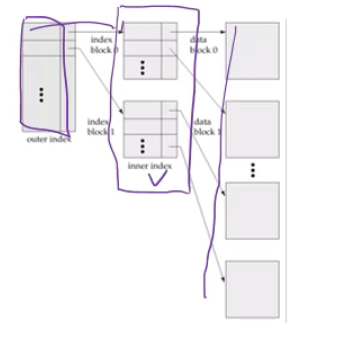


record들이 search-key의 순서대로 저장되어 있을 경우에 사용 가능.

Insert/delete에서 공간과 유지보수 overhead가 적음.

보통 record를 찾는 데 dense index보다 느림.

**Multilevel Index**



Primary index가 memory에 있지 않으면 access 비용이 비싸짐.

Disk에 있는 primary index를 sequential file로 취급하고, 그것에 대한 sparse index를 구성함.

Outer index – primary index에 대한 sparse index

Inner index – the primary index file.

만약 Outer index가 여전히 main memory에 fit하기에 크다면, 또 만듦.

모든 level의 index들은 insert/delete 시 update 되어야 함.

**Index Update: Insertion**

Search-key value를 이용해서 lookup을 수행함.

Dense Index

Search-key value가 index에 나타나지 않으면 insert함.

Sparse Index

만약 index가 파일의 각각의 block의 entry에 저장되어 있으면, 새 블락이 만들어지기 전까지는 변화가 필요 없음.

만약 새 블락이 만들어지면, 새 블락의 첫 search-key value는 해당 index로 삽입됨.

**Index Update: Deletion**

Index는 delete에 따라 update 되어야 함.

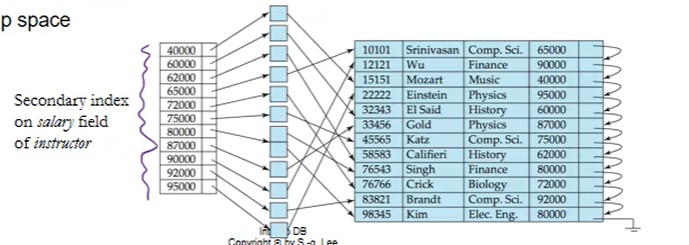
만약 삭제된 record가 파일에서 유일한 search-key value를 가지면, search-key 또한 index에서 없어져야 함.

Sparse index

만약 search key에 대한 entry가 index에 존재하면, 파일의 다음 search-key 값으로 대체됨.

다음 search-key가 index entry에 있으면 entry는 그냥 삭제됨.

**Primary and Secondary Indices**

****

**Secondary Index는 항상 dense 해야 함.**

index들은 record를 찾을 때 큰 이득을 제공함.

파일이 변경되면 모든 index는 업데이트 되어야 함. (overhead.)

Primary index에서 Sequential scan은 효율적이지만, secondary index에서 sequential scan은 해당 순서대로 정렬이 되어 있지 않기 때문에 비쌈.

Index는 공간을 많이 씀.

**Ordered Index Performance**

Access type supported: point queries, range queries

Time

Access time: index tree의 높이에 따라 다름.

Insertion/Deletion time: 마찬가지로 높이에 따라 다름.

N key values & k children/node

Best case: height = log\_k(n)

Worst: height = n

Balanced index tree를 원함.

**B+ Tree Index**

다음과 같은 성질들을 만족함.

- 모든 leaf 노드들은 root로부터 같은 길이를 가짐.

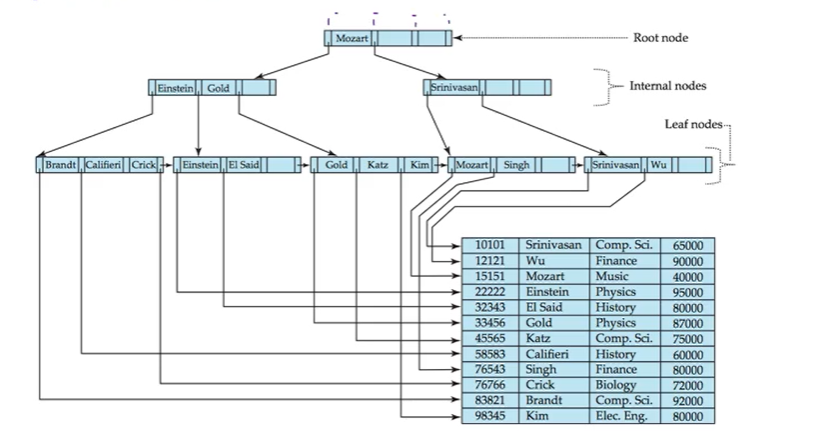
- root 이외의 각각의 노드는n/2와 n개 사이의 pointer를 가지고 있음.

(n-1)/2 <= ~~ <= n-1

특별한 케이스: root node

- leaf가 아니면 적어도 2개의 children만 있으면 됨.

- root가 leaf이면 0 ~ n-1을 가질 수 있음.



Leaf node는 record에 대한 dense index를 형성함.

레코드는 disk에 저장되어 있는데, index들도 disk에 저장됨.

모든 node는 block size와 동일한 크기를 가지게 함.

**Advantages of B+ tree index**

장점

Insertion/deletion에서 작은 local change로 성질 유지.

Performance를 수행하기 위해 전체 파일을 재조직할 필요가 없음.

AVL tree랑 비슷..

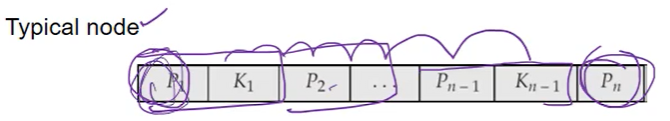
단점 (경미함)

Insertion/deletion에서 약간의 overhead.

Space overhead

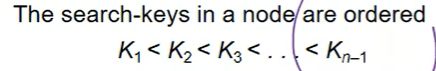
장점이 더 커서 광범위하게 쓰이고 있음.

B+ Tree Node Structure



K는 key-value, P는 children을 위한 pointer.

Leaf 노드면 P는 해당 record 혹은 bucket을 pointer.



**Leaf Nodes**

P\_i는 record 자체를 search-key 값인 K\_i에 대해서 point. 혹은 버킷.

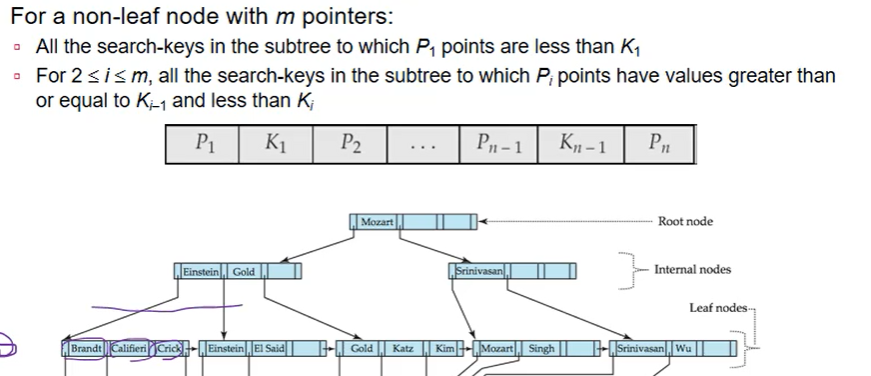
Search-key가 primary key가 아닐 경우만 bucket 구조가 필요함.

L\_i 와 L\_j (i < j인 경우)

L\_i의 search key value는 L\_j의 search key value보다 작다.

P\_n (마지막 포인터)는 search-key order의 다음 leaf node를 가리킴.

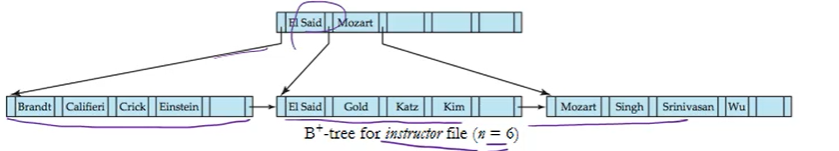
**Non-Leaf Nodes**

****

Non-leaf node들은 leaf node들에 대해서 multi-level sparse index를 형성함.

P1은 K1보다 작은 값들, P2는 K1보다는 크고 K2보다는 작은 값들.. 이런 순서대로 저장됨.

**Example of B+ tree**

****

N = 5일 경우에는 (포인터 개수)

루트가 아닌 모든 node들은

2개 ~ 4개의 key value를 가져야 함. (3~5개의 pointer)

루트가 아닌 Non-leaf node들은

3~5개의 children을 가져야 함 (포인터와 같은 의미)

Root는 최소 2개의 children을 가져야 함.

**Observations about B+ trees**

Inner-node 연결이 pointer에 의해 수행되기 때문에, logically 가까운 block들이 physically 가깝다는 의미는 아님.

Non-leaf level의 B+ 트리는 sparse index의 hierarchy를 구성함.

B+ 트리는 상대적으로 level이 적음.



Index가 log 시간으로 구성되기 때문에 insert/delete가 효율적임.

**B+ Tree: Search**

****

**B+ Trees: Search Performance**

K개의 search-key value가 있을 경우, 트리의 높이는 log\_(n/2)(k) 보다 작음.

Node는 주로 disk block의 크기와 같음.

보통 2~4KB.

그리고 n은 보통 50~100 (index entry당 40B)

100만개의 search key value와 n = 100에 대해서

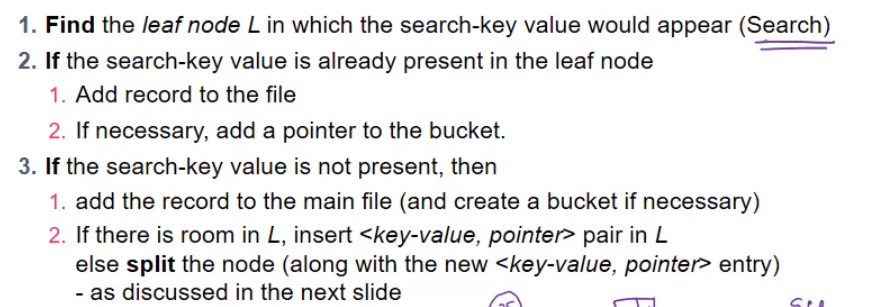
Log\_50(1000000) = 4 nodes가 lookup에 access됨.

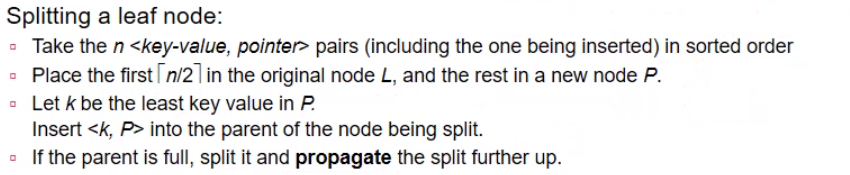
Balanced binary tree와 다른 점

20개의 node가 lookup에 사용됨.

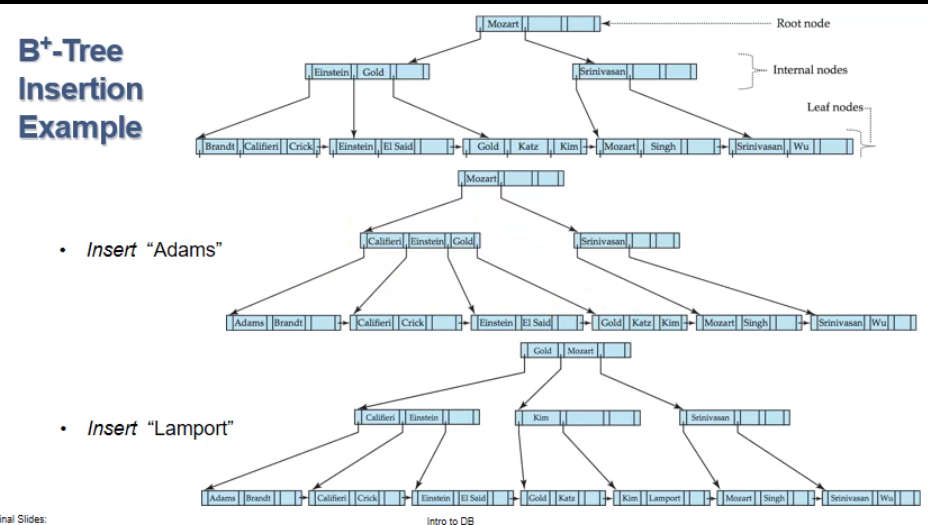
모든 노드가 disk I/O가 필요할 수도 있기 때문에 (20ms가 소요되는..) 차이가 큼!

**Updates on B+ Trees: Insertion**

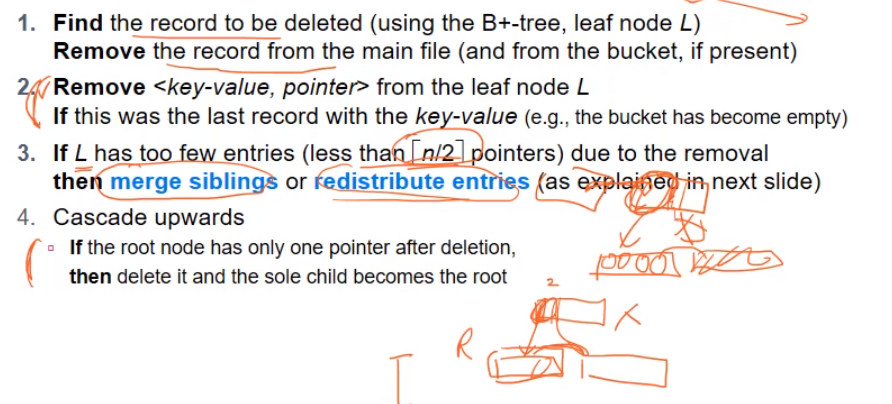


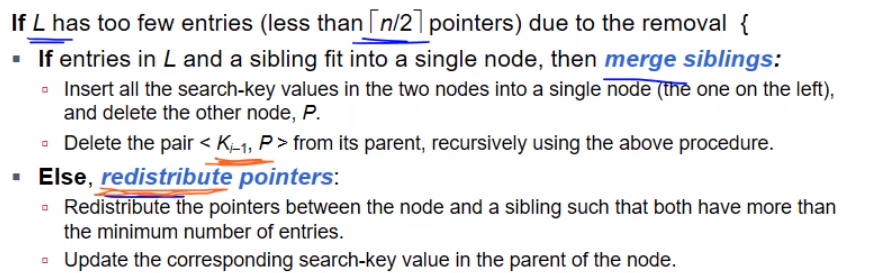


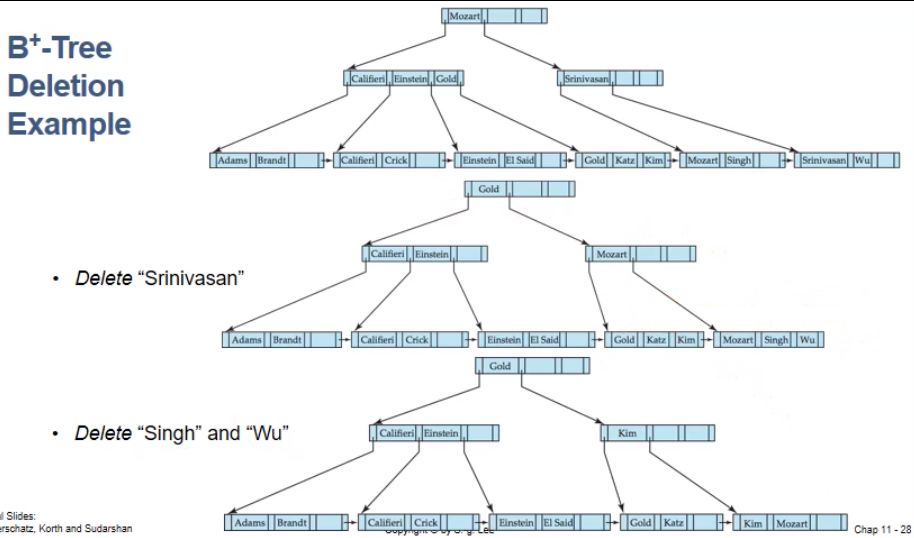
최악의 경우는 root node가 스플릿되면서 트리의 높이가 1 증가할 때임.



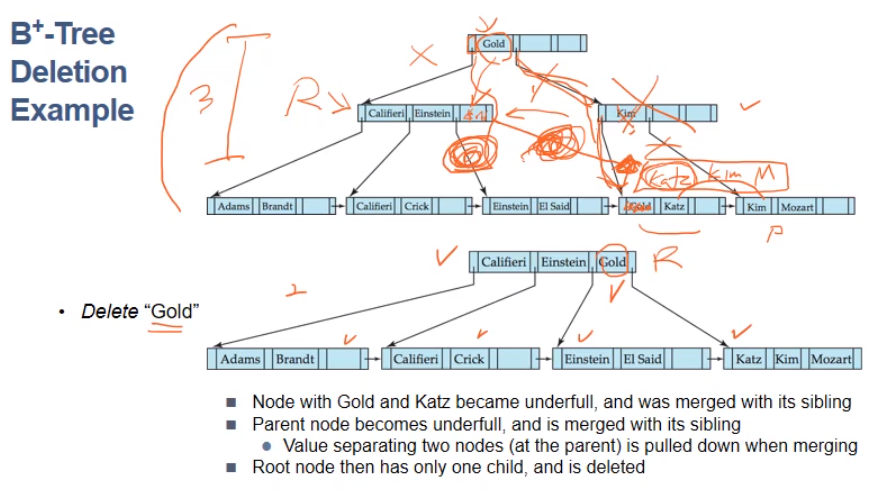
**Updates on B+ Trees: Deletion**







1개만 합치는 것이 아닌 건 양쪽을 반반으로 나눠주기 위해. 예를 들어 100개 50개가 있어서 50개 쪽이 줄면 99 / 50 뭐 이런 식으로 되기 때문임.



신기한건 Gold는 delete했는데 중간 node에는 골드가 남아 있게 됨.

**B+ Tree File Organization**

Leaf node에 포인터 대신에 레코드를 둠.

Data file organization 문제가 이 방법을 사용해서 해결될 수 있음.

Index file degradation 문제가 B+ 트리 index를 이용해서 풀리는 것과 같음.

Leaf node들은 여전히 half full이어야 함.

Record가 포인터보다 크기 때문에, leaf node에 저장된 record의 최대 개수는 nonleaf node의 pointer 개수보다 적음.

Insertion/deletion은 기존과 같음.

Space utilization이 중요. 레코드가 포인터보다 더 크기 때문에.

Split과 merge를 할 때 더 많은 sibling들을 참여시킴.

각각 2n/3 entries를 가지게 invlove함.

**Static Hashing**

- bucket

하나 이상의 record를 담고 있는storage의 단위.

주로 disk block임.

- hash file organization

Hash function을 이용해서 search-key value로부터 record의 bucket을 바로 얻을 수 있음.

- Hash function h

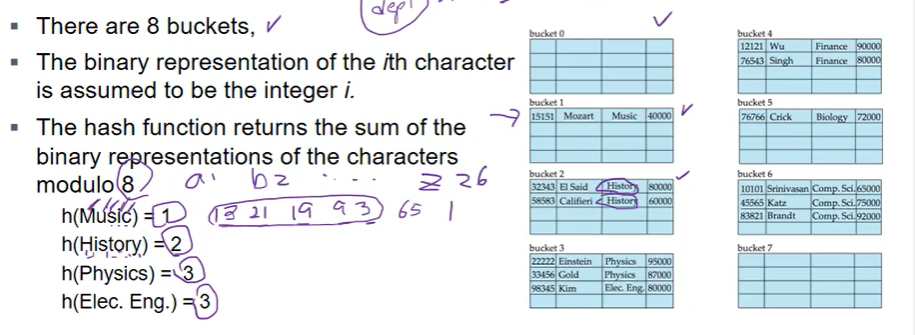
h: K -> B.

K는 search-key value, B는 bucket address.

다른 search-key value를 가진 레코드가 같은 bucket에 저장될 수도 있음.

**Hash File Organization**

각 알파벳을 숫자 i로 나타내서 그 i를 다 더한 값을 8로 나눠서 bucket에 넣음.



**Hash Functions**

Worst hash function = 모든 search-key value를 같은 bucket에 매핑. Access time이 O(n)

Ideal hash function은 uniform (각 bucket에 저장되어 있는 search-key value의 수가 같아야 함), random (file의 search-key value의 실제 분포와 상관 없이 각 bucket이 같은 수의 레코드를 가짐)해야함.

일반적으로 hash function은 search-key의 내부 binary 표현을 계산.

**Handling of Bucket Overflows**

Bucket overflow는 bucket 수가 적거나, record 분배에 skew가 있을 때 생김.

- 많은 record가 같은 search-key value를 가짐.

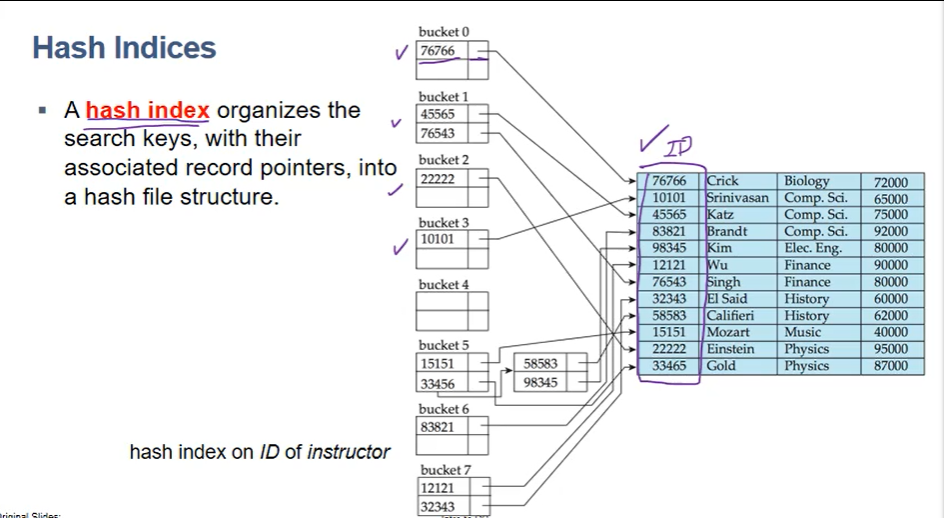
- 선택된 hash-function이 key-value에서 균등하지 않은 분배를 생산함.

Bucket overflow를 줄일 수는 있어도 없앨 수는 없음. 따라서 overflow buckets를 이용해서 다룸.



**Hash Indices**

Hash index는 search key와 연관된 record 포인터를 관리함.



**Deficiencies of Static Hashing**

해쉬 함수 h가 정해진 bucket address B의 집합에 대해 매핑함.

하지만 Database는 시간이 지나며 커지거나 작아지거나 함.

- bucket의 수가 너무 작으면 overflow가 많아서 성능 저하.

- 너무 많으면 시간 낭비.

하나의 방법: 주기적으로 file을 re-organization 함.

더 좋은 방법: dynamic hashing.

**Dynamic Hashing**

bucket의 수를 dynamic하게 변경될 수 있게 허용함.

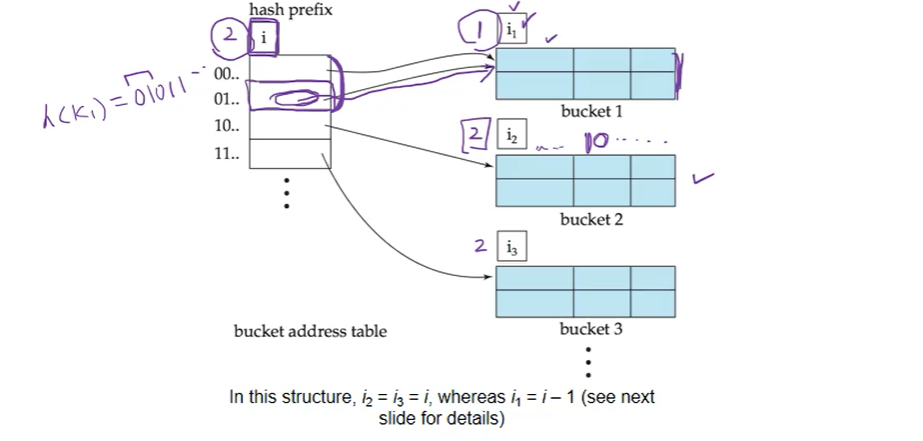
Extendable Hashing – dynamic hashing의 한 종류.

- hash function이 더 넓은 range의 value를 생성함. – 주로 b-bit integer, b = 32.

- hash function의 prefix만을 사용.

- 

**General Extendable Hash Structure**



I가 늘어나면 bucket을 split함. 늘어나면 hash prefix를 보고 테이블을 나눔. 나눠지지 않은 bucket들은 같은 entry를 가짐.

**Extendable Hashing**

- 각 bucket j는 value i\_j를 가짐.

- 같은 bucket을 가리키는 모든 엔트리는 첫 i\_j bit가 같다.

- search key K\_j를 포함하는 bucket을 locate하려면…

1. h(K\_j) = X를 계산한다.

2. X의 첫 i개의 high order bits를 bucket address table의 displacement로 사용한다. 그리고 포인터가 적절한 bucket을 가리키도록 함.

- search-key value K\_j를 가지고 있는 record를 삽입하려면…

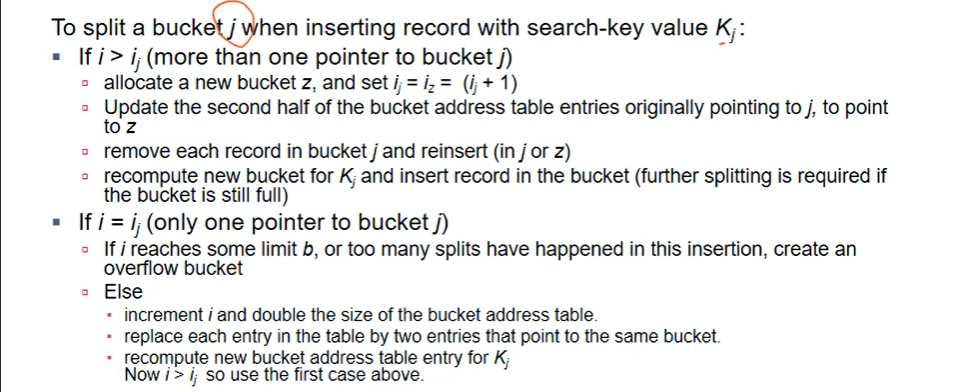
1. bucket을 look-up하고 locate하는 것과 같은 절차를 따름. j라고 하자

2. bucket j에 공간이 있으면 그냥 넣으면 됨.

3. 없으면 bucket은 split되어야 하고 다시 insert를 시도함.

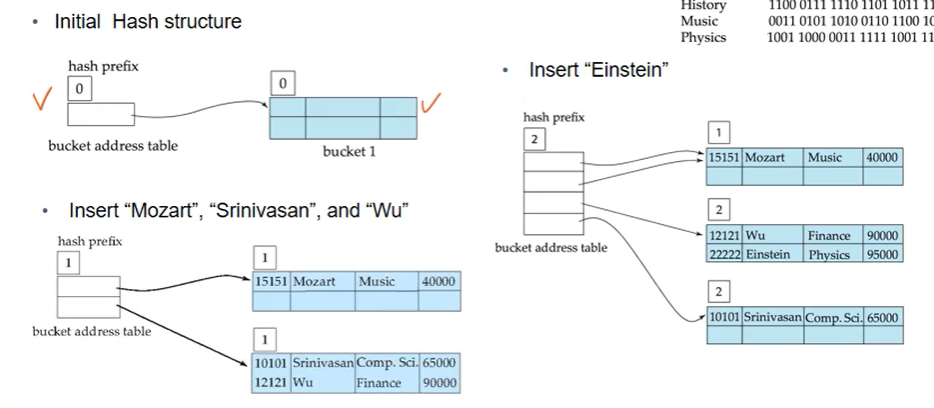
- 때에 따라 overflow bucket을 사용하는 경우도 있음.

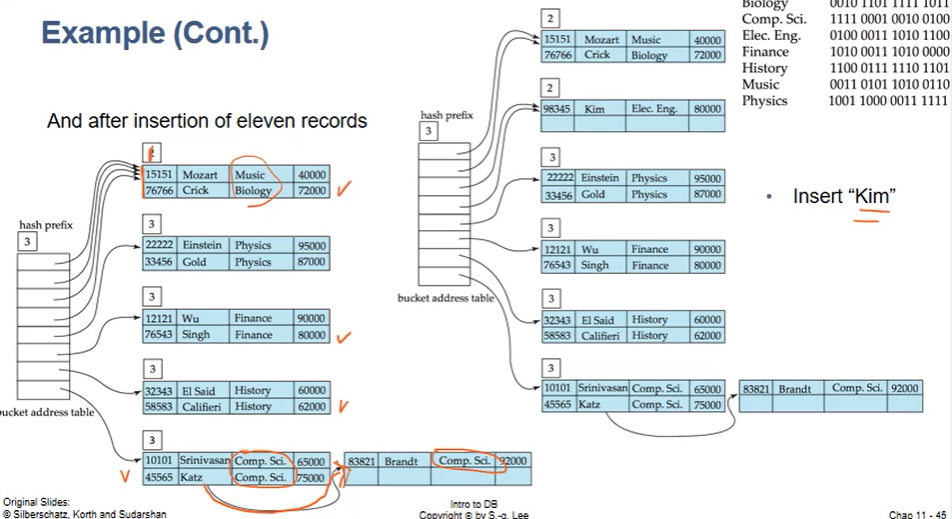
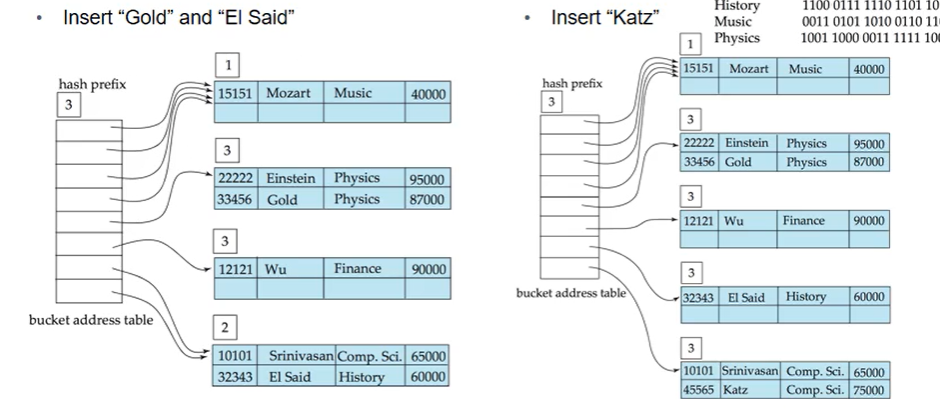
**Extendable Hashing: Insertion**



i가 어떠한 한계 b에 도달하거나 너무 split이 많이 일어난다 싶으면 overflow를 사용할 수도 있음.

**Extendable Hashing: Example**



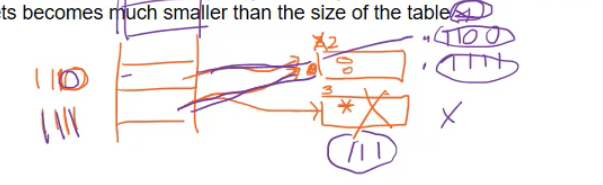


**Extendable Hashing: Deletion**

Key value를 찾아가서 remove 함.

Bucket도 empty가 되면 삭제될 수 있음.

Coalescing bucket이 이루어질 수도 있음.



- “buddy” bucket들끼리만 합침. 즉 같은 value i\_j와 같은 i\_j – 1 prefix를 가져야 함.

- 위의 그림과 같은 식으로 하나는 더 깊게 들어가있고 이러면 못함.

Bucket address table의 크기를 줄이는 것 또한 가능함.

- bucket address table의 크기를 줄이는 것은 비싼 연산이고 bucket의 수가 테이블의 크기보다 훨씬 적을 때만 수행해야 함.

**Extendable Hashing vs Other Schemes.**

Extendable hashing의 장점:

- hash performance는 파일이 커지면서 줄어들지 않음.

- minimal space overhead.

단점:

- record를 찾기 위해 indirection의 레벨이 늘어남.

- bucket address table 자체는 엄청 커질 수도 있음. (메모리보다 더!)

- 디스크에도 매우 큰 연속된 공간을 할당할 수는 없음.

- bucket address table의 크기를 바꾸는 건 비싼 연산.

- 따라서 너무 큰 테이블에는 적용하지 않음.

**Ordered Indexing vs Hashing**

Periodic re-organization

- static hashing이 worst. 나머지는 필요 X

Relative frequency of insertions and deletions

- 자주 없다면 static hashing이 좋음.

Average access와 worst-case 중 어느 것에 최적화?

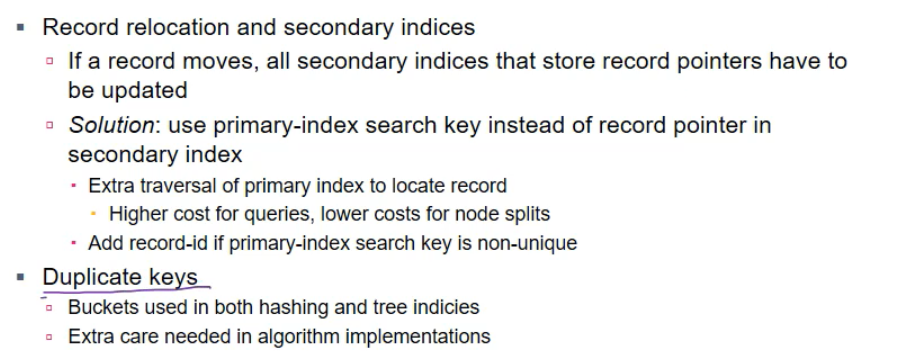
- B+는 worst도 로그이고.. 해쉬는 최악이 n임

Expected type of queries:

- Hashing은 특정한 값을 가지는 record를 찾는 데는 보통 더 나음.

- range queries가 일반적이면, ordered indices가 나음.

**Other Issues in Indexing**



**Index Definition in SQL**

