Index Structures 강의

강의 내용

- Binary Trees (Ch. 6 in “**화일 구조**”)
- B-tree, B*-tree, Trie (Ch. 6 in “**화일 구조**”)
- B^+-tree (Ch. 7 in “**화일 구조**”)
- Indexed Sequential File (Ch. 7 in “**화일 구조**”)

색인, 인덱스 (Index)

특징

- 파일의 레코드들에 대한 효율적 접근 구조
- <레코드 키 값, 레코드(에 대한) 포인터> 쌍

종류

- 키에 따른 인덱스 분류
  - 기본 인덱스(Primary Index): 기본 키(Primary Key)를 포함하는 인덱스 (키의 순서가 레코드의 순서를 결정지음)
  - 보조 인덱스(Secondary Index): 기본 인덱스 이외의 인덱스 (키의 순서가 레코드의 순서를 의미하지는 않음)
- 파일 조직에 따른 인덱스
  - 집중 인덱스(Clustered Index): 데이타 레코드의 물리적 순서가 그 파일에 대한 인덱스 엔트리 순서와 동일하게(유사하게) 유지되도록 구성된 인덱스
  - 비집중 인덱스(Unclustered Index): 집중 형태가 아닌 인덱스
- 데이타 범위에 따른 인덱스 분류
  - 밀집 인덱스(Dense Index): 데이터 레코드 각각에 대해 하나의 인덱스 엔트리가 만들어진 인덱스
  - 희소 인덱스(Sparse Index): 레코드 그룹 또는 데이터 블록에 대해 하나의 엔트리가 만들어지는 인덱스

Binary Search Tree (BST) (1/2)

Binary Tree (이진 트리)

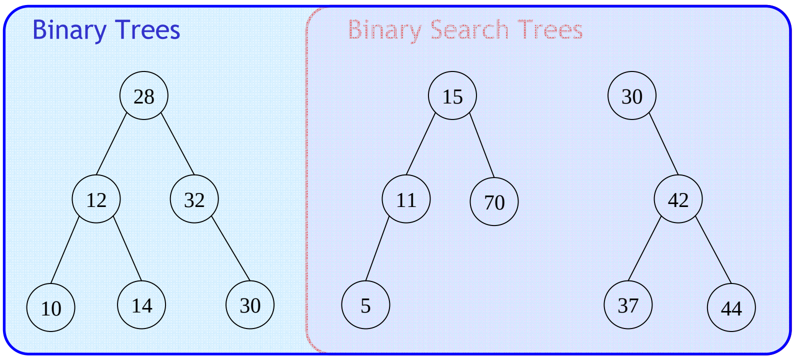
- 유한한 수의 노드를 가진 트리
- 왼쪽 서브 트리와 오른쪽 서브 트리로 구성

Binary Search Tree (BST, 이원 탐색 트리)

- 이진 트리
- 각 노드 N_i: 레코드 키 K_i와 이 키가 가지고 있는 레코드 포인터를 포함
- 공백(empty)이 아닌 이원 탐색 트리의 성질
  - 모든 노드는 상이한 키 값을 갖는다.
  - 루트 노드 N_i의 왼쪽 서브 트리(Left(N_i))에 있는 모든 노드의 키 값은 루트 노드의 키 값보다 작다. (maximum key value of the left sub-tree is less than key value of the root node.)
  - 루트 노드 N_i의 오른쪽 서브 트리(Right(N_i))에 있는 모든 노드의 키 값은 루트 노드의 키 값보다 크다. (minimum key value of the right sub-tree is greater than key value of the root node.)
  - 왼쪽 서브트리와 오른쪽 서브트리는 모두 이원 탐색 트리이다. (Both left sub-tree and right sub-tree are also binary search trees.)

Binary Search Tree (BST) (2/2)

- Binary Tree와 Binary Search Tree 예제



BST에서의 검색

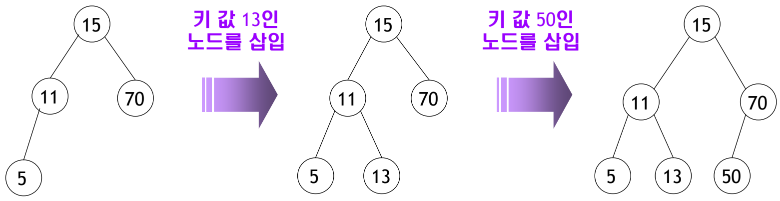
- 루트 N_i일 때, 주어진 키 값 K인 노드 검색 과정
  1. 트리가 공백 : 검색은 노드를 찾지 못하고 실패
  2. $K = K_i$: 노드 N_i가 원하는(찾고자 하는) 노드 → 검색 종료
  3. $K < K_i$: N_i의 왼쪽 서브 트리를 검색 → left sub-tree를 대상으로 recursive하게 검색
  4. $K > K_i$: N_i의 오른쪽 서브 트리를 검색 → right sub-tree를 대상으로 recursive하게 검색

```
searchBST(B, s_key)     // B = binary search tree, s_key = search key value
    p ← B;
    if (p = null) then     // 검색 실패 (해당 키 값을 가진 노드가 없음)
        return null;
    if (p.key = s_key) then        // 검색 성공
        return p;
    if (p.key < s_key) then        //  오른쪽 서브 트리 검색
        return searchBST(p.right, s_key);
    else return searchBST(p.left, s_key);     // 왼쪽 서브 트리 검색
end searchBST()
```

BST에서의 삽입

- 루트 N_i인 BST에, 키 값 K인 새로운 노드를 삽입
  1. 트리가 공백(empty): K를 루트 노드로 삽입
  2. $K = K_i$: 트리에 값이 같은 키 값이 이미 존재하므로 삽입을 거부 (unique 보장)
  3. $K < K_i$: N_i의 왼쪽 서브 트리로 이동하여 계속 탐색
  4. $K > K_i$: N_i의 오른쪽 서브 트리로 이동하여 계속 탐색

- BST에 삽입 예: 키 값 13, 50인 노드를 삽입

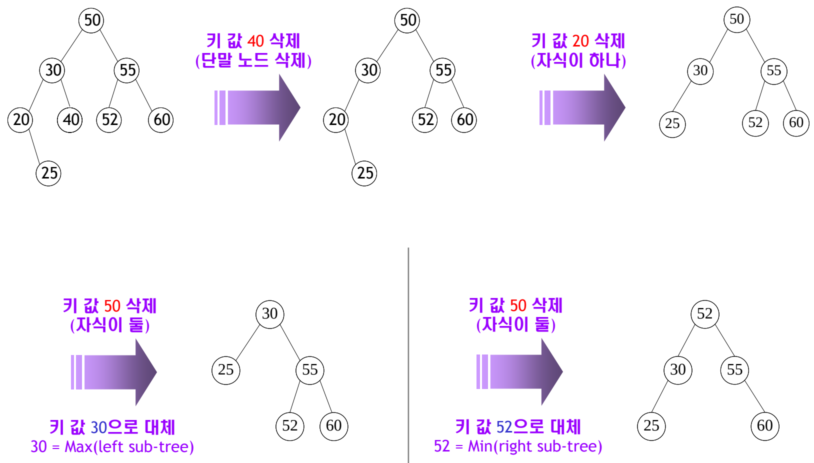


BST에서의 삭제 (1/2)

- 삭제될 노드가 가진 자식 수에 따라 다른 연산을 수행
  1. 자식이 없는 단말 노드(leaf node)의 삭제
     → 단순히 해당 단말 노드를 삭제함
  2. 자식이 하나인 노드의 삭제 (left or right child 하나만 존재)
     → 삭제되는 노드 자리에 자식 노드를 위치시킴
  3. 자식이 둘인 노드의 삭제
     – 삭제되는 노드 자리에 왼쪽 서브 트리에서 제일 큰 키 값 또는 오른쪽 서브 트리에서 제일 작은 키 값으로 대체 선택
       · Max(left sub-tree) = key value of the right most node in the left sub-tree
       · Min(right sub-tree) = key value of the left most node in the right sub-tree
     – 해당 서브트리에서 대체 노드를 삭제

BST에서의 삭제 (2/2)

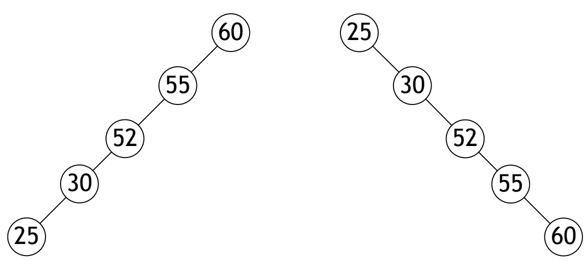
- BST에 삭제 예: 키 값 40, 20, 50인 노드를 삭제



BST의 성능 (1/2)

- **편향된**(Skewed) Binary Search Tree
  - 편향되어 있는 경우, BST의 탐색 시간은 최악이 됨
  - N개의 노드인 BST에서 최악의 탐색 시간: N번의 노드 탐색

- Skewed BST의 예



BST의 성능 (2/2)

성능

- 트리 형태와 (개별) 노드에 대한 접근 빈도에 의존적임
- 가장 자주 접근되는 노드는 루트에 가장 가깝게 유지하는 것이 유리함
- 균형 트리(balanced tree) 유지: 모든 노드에 대해 양쪽 서브 트리의 노드 수가 가능한 동일하게 만들어 트리의 최대 경로 길이를 최소화

BST의 단점

- 잦은 삽입과 삭제가 발생하는 경우, 균형 트리를 유지하기 어려움 (→ AVL-tree)
- 작은 분기율(branching factor)에 따른 긴 탐색 경로와 검색 시간 (→ B-tree)
  - 분기율이 2이므로, 각 노드는 많아야 두 개의 서브 트리를 가짐
  - N개의 노드를 갖는 트리의 최소 높이: $\lfloor log_2 N \rfloor + 1$

AVL 트리 (1/2)

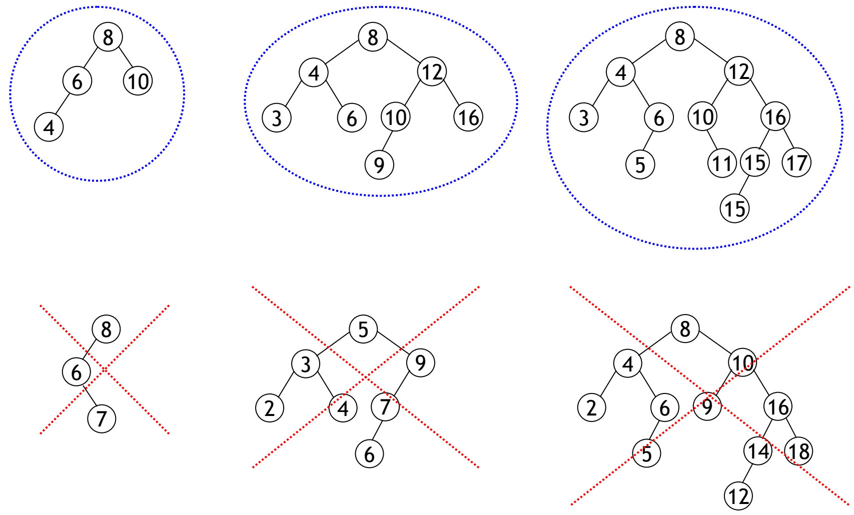
높이 균형(height-balanced) BST

- 삽입, 삭제 연산 시간이 짧음
- 검색 시간: O(logN)
  (→ Skew가 발생치 않으므로, worst case = average case가 됨)
- AVL 트리: 고안자 Adelson-Velskii와 Landis 이름의 Initial에서 유래
- 트리 전체를 재균형 시키지 않고도(국부적으로 재균형 시키면서) 트리의 균형 유지

정의

- AVL 트리 T: 공백이 아닌 Binary (Search) Tree
- B_f(= balance factor) = $|h(Right(N_i)) - h(Left(N_i))| \leq 1$, $N_i \in T$
  (→ 모든 노드의 왼쪽 서브 트리와 오른쪽 서브 트리의 높이는 그 차가 1 이하임)
  - 오른쪽 서브 트리의 높이가 큰 경우: $B_f(N_i) = 1$
  - 왼쪽 서브 트리의 높이가 큰 경우: $B_f(N_i) = -1$
  - 두 서브 트리의 높이가 같은 경우: $B_f(N_i) = 0$
- 공백 서브 트리의 높이(B_f(empty tree)): -1로 정의

AVL 트리 (2/2)



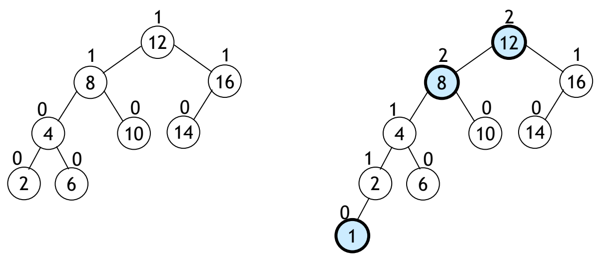
AVL 트리에서의 검색과 삽입 (1/2)

검색

- 일반 BST에서의 검색 연산과 동일
- 시간 복잡도: $O(\log_2 N)$

삽입

- 삽입되는 위치에서 루트로의 경로에 있는 조상 노드들의 B_f에 영향을 줄 수 있음
- 즉, 불균형이 발생할 수 있으며, 따라서, 불균형이 탐지된 가장 가까운 조상 노드의 B_f를 ±1 이하로 재균형 시켜야 함



신규 노드 삽입으로 인한 AVL 파괴 예

AVL 트리에서의 검색과 삽입 (2/2)

삽입 (계속)

- 불균형으로 판명된 노드를 x라 할 때, x의 두 서브 트리 높이의 차, 즉 B_f가 2가 됨
- 다음 네 가지 경우 중 하나로 인해 발생함

LL : x의 왼쪽 자식(L)의 왼쪽 서브 트리 (L) 에 삽입
RR : x의 오른쪽 자식(R)의 오른쪽 서브 트리 (R) 에 삽입
LR : x의 왼쪽 자식 (L) 의 오른쪽 서브 트리 (R) 에 삽입
RL : x의 오른쪽 자식의 왼쪽 서브 트리 (L) 에 삽입

AVL 트리에서 회전(rotation) (1/5)

회전이 필요한 이유

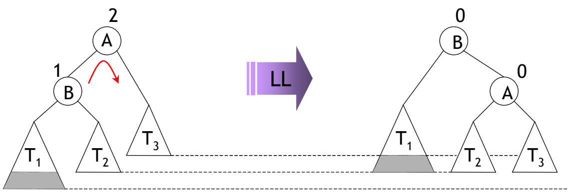
- 노드의 삽입 (혹은 삭제)로 인하여 불균형이 발생한 경우($|B_f|>1$), 트리의 재균형을 위하여 회전이 필요함
- 즉, 불균형이 발생한 노드 x를 중심으로 회전을 수행하여 균형을 맞춤

회전의 종류

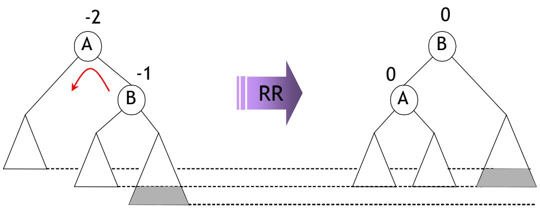
- 단일 회전 (single rotation)
  - LL, RR 등이 이에 해당하며, 한 번의 회전만 필요함
  - 탐색 순서를 유지 하면서 부모와 자식 원소의 위치를 교환함으로써 재균형이 이루어짐
- 이중 회전 (double rotation)
  - LR, RL에서 발생하며, 두 번의 회전을 필요로 함

AVL 트리에서 회전(rotation) (2/5)

- LL Rotation

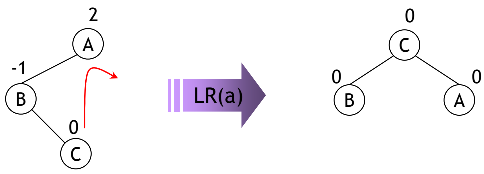


- RR Rotation

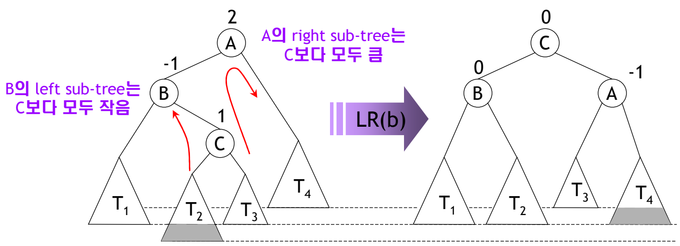


AVL 트리에서 회전(rotation) (3/5)

- LR Rotation: case (a)

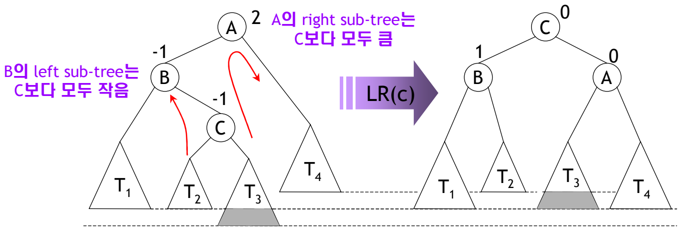


- LR Rotation: case (b)

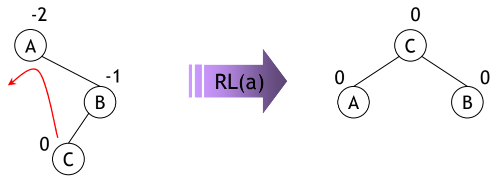


AVL 트리에서 회전(rotation) (4/5)

- LR Rotation: case (c)

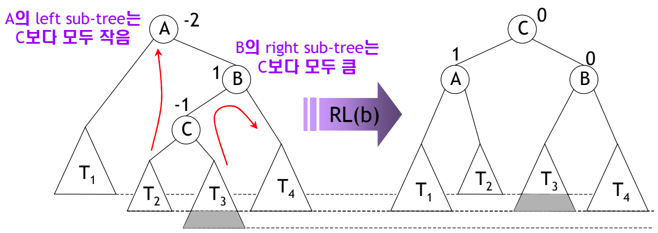


- RL Rotation: case (a)

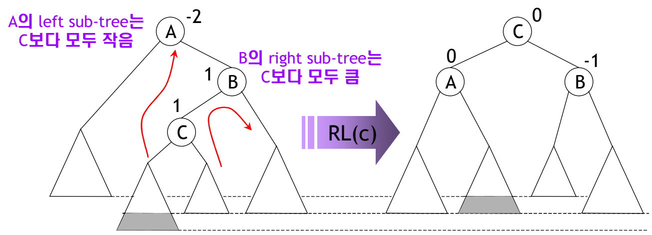


AVL 트리에서 회전(rotation) (5/5)

- RL Rotation: case (b)

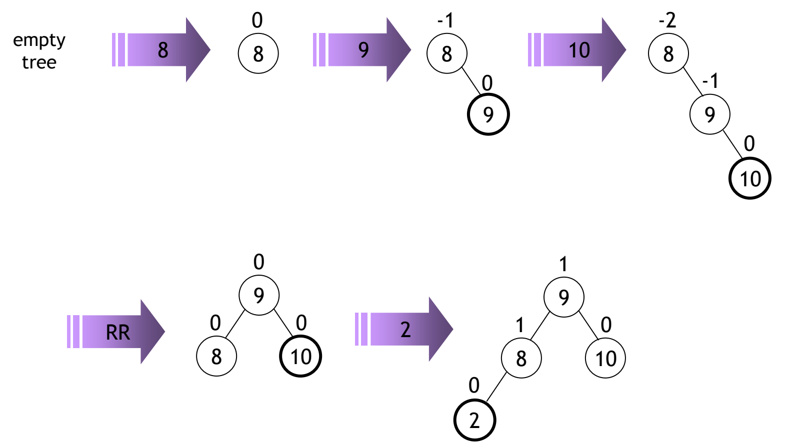


- RL Rotation: case (c)

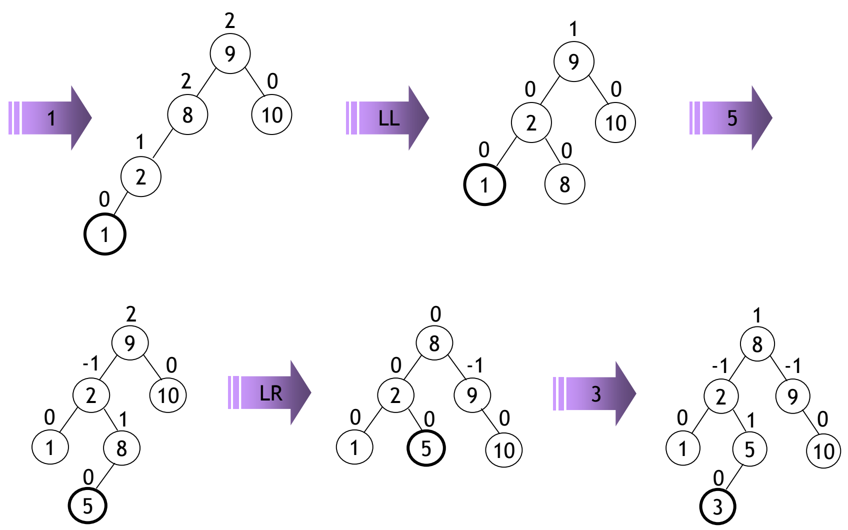


AVL 트리 구성 예제 (1/4)

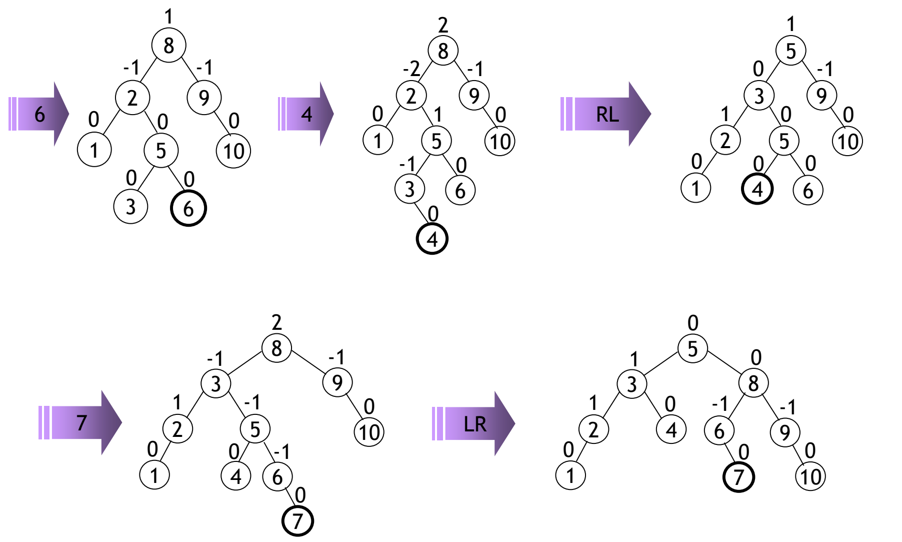
- 키 값(8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12)을 차례대로 삽입하면서 AVL 트리를 구성하는 예



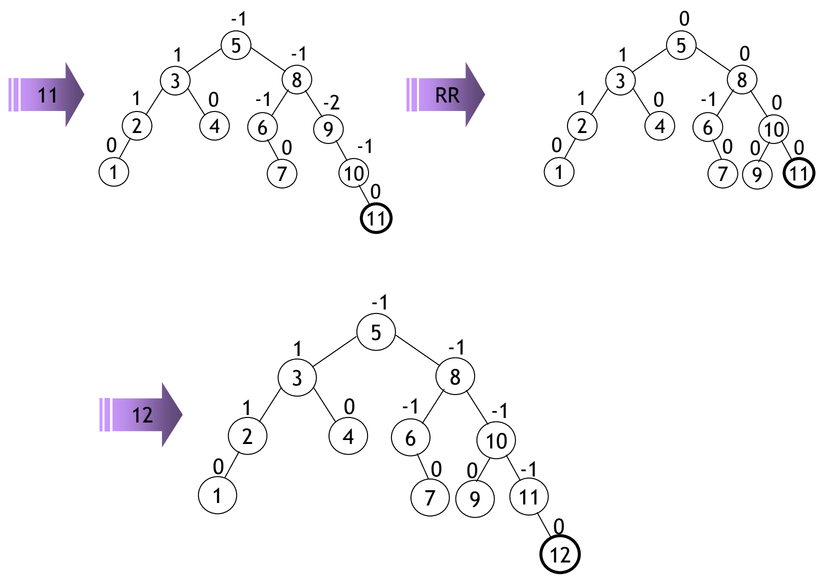
AVL 트리 구성 예제 (2/4)



AVL 트리 구성 예제 (3/4)



AVL 트리 구성 예제 (4/4)



AVL 트리의 높이

AVL 트리에서 높이의 범위

- N개의 노드를 가진 높이 AVL 트리는 완전 균형 이진 트리[1)]보다 45% 이상은 높아지지 않음이 증명되었음
- $\log_2(N+1) \leq h \leq 1.4404 \log_2(N+2) - 0.328$

AVL 트리 : 완전 균형 이진 트리[1)]

- O(1.4 log N) : O(log N)
- AVL 트리는 부분적인 재구성만을 수행하기 때문에, 탐색 시간 측면에서는 AVL 트리가 더 길다. (그러나, 훨씬 실용적이다.)

1) 완전 균형 이진 트리(complete balanced binary tree): 주어진 키 값에 대해서, (검색 측면에서) 가장 이상적으로 좌우 균형이 잡힌 트리

Index Structures 강의

강의 내용

• Binary Trees	(Ch. 6 in "화일 구조")
• B-tree, B*-tree, Trie	(Ch. 6 in "화일 구조")
• B^+-tree	(Ch. 7 in "화일 구조")
• Indexed Sequential File	(Ch. 7 in "화일 구조")

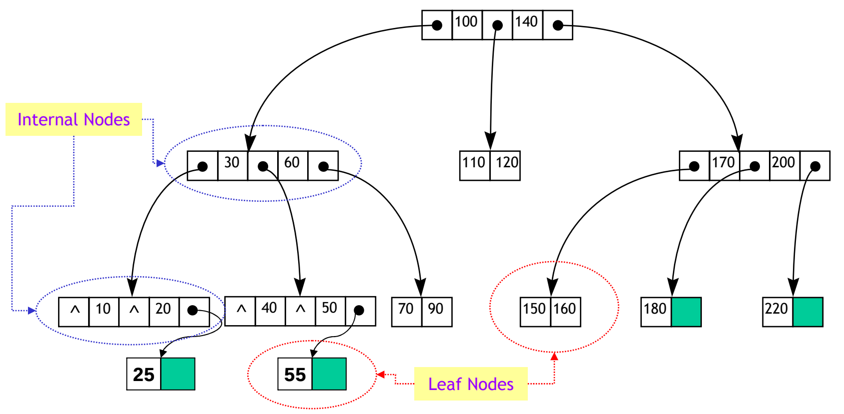
m-way Search Tree (1/3)

- BST보다 분기율(branch factor)을 높이면?
  즉, m개 서브 트리를 형성하면?
  - 장점: 트리의 높이가 감소 (특정 노드의 탐색 시간 감소)
  - 단점: 삽입, 삭제 시 트리의 균형을 유지하기 위해 복잡한 연산이 필요

- m-way search tree의 성질
  - 노드 구조 = <n, P_0, K_1, P_1, K_2, P_2, … , P_{n-1}, K_n, P_n>
    (n: 키 값의 수, $1 \le n \le m$, P_i: 서브 트리에 대한 포인터, K_i: 키 값)
  - 키는 오름차순으로 정렬: $K_i < K_{i+1}$, $1 \le i \le n-1$
  - P_i $(0 \le i \le n-1)$가 가리키는 서브 트리 내의 키 값 < K_{i+1}
  - P_n이 가리키는 서브 트리 노드들의 키 값 > K_n
  - P_i 가 가리키는 서브 트리: m-way search tree

m-way Search Tree (2/3)

- 예제 (3-way search tree)



실제로, 키 값 K_i는 (K_i, A_i)를 의미하며, 여기서 A_i는 데이타 레코드의 주소를 의미함

m-way Search Tree (3/3)

- m-way search tree의 탐색 시간: 탐색 경로 길이(높이)에 비례
  - 각 레벨에서는 한 개의 노드만 탐색 (높이가 h이면 h개 노드 탐색)
  - 분기율(m)을 크게 하면 할 수록 트리의 높이가 낮아짐
- 한 노드에 m-1개 키 값을 저장하는 m-way search tree의 경우,
  - 높이 h이면 (m^h-1)개의 키 값을 저장
    ((m-1) + m(m-1) + m(m(m -1)) + … = m - 1 + m^2 - m + m^3 - m^2 + … = m^h - 1)
  - 예: 4-way search tree에서 높이 3이면, 4^3-1=63개 키 값 저장
- n개의 키를 가진 m-way search tree
  - 최소 높이 h = $\lceil \log_m(N+1) \rceil$
  - 최대 탐색 시간: O(h) = O($\log_m$(N+1))
  - 예: m=2이면, BST의 탐색 시간 (= O($\log_2$(N+1))

B-트리

- Bayer & McCreight가 제안
  R. Bayer and C. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica 1*, pp. 173-189, 1972.
- B-tree?: balanced m-way search tree
  - m-way search tree의 균형을 유지하기 위하여 효율적인 알고리즘을 제공
  - B^+-tree와 함께 가장 많이 사용되는 인덱스 방법
- 차수 m인 B-트리의 특성
  - B-트리는 공백(empty)이거나 높이가 1 이상인 m-way search tree임
  - 루트와 단말(leaf)을 제외한 내부 노드(internal node)는 다음 특성을 가짐
    - 최소 $\lceil m/2 \rceil$, 최대 m개의 서브 트리를 가짐 (→ 절반 이상의 Utilization을 보장)
    - 적어도 $\lceil m/2 \rceil$ - 1개의 키 값 (→ 노드의 반 이상 채워짐)
  - 루트 노드: 단말이 아니면 적어도 두 개의 서브 트리를 가짐
  - 모든 단말 노드는 같은 레벨임

B-트리의 노드 구조 (1/2)

노드 구조 (m-way)

$<n, p_0, <K_1, A_1>, P_1, <K_2, A_2>, P_2, \ldots, P_{n-1}, <K_n, A_n>, P_n>$

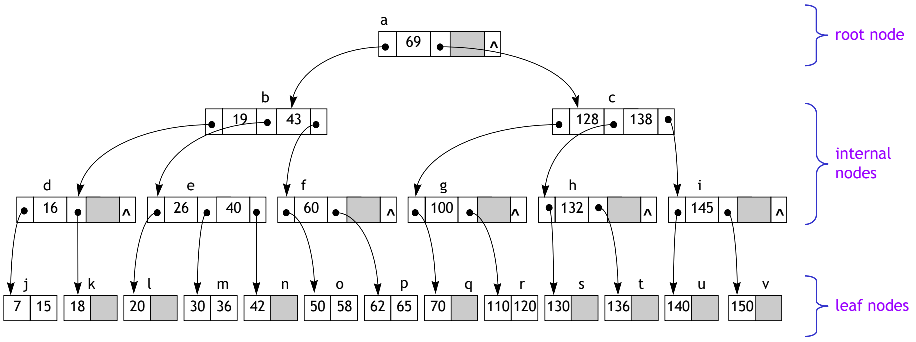
- n = 키 값의 수($1 \leq n < m$), $P_0, \ldots, P_n$ = 서브 트리에 대한 포인터,
  K_n = 키 값, A_i = 키 값 K_i를 가진 레코드에 대한 포인터
- 한 노드 안의 키 값은 오름차순으로 정렬($1 \leq i \leq n-1 \rightarrow K_i < K_{i+1}$)
- P_i가 가리키는 서브 트리의 모든 키 값 $< K_{i+1}$
- P_n이 가리키는 서브 트리의 모든 키 값 $> K_n$
- $P_i(0 \leq i \leq n)$가 가리키는 서브 트리 : m-way B-트리 구조를 만족함

B-트리의 장점

- 삽입과 삭제가 발생한 후에도 균형 상태를 유지
- 저장 장치의 효율성 제공
  - 각 노드(루트 제외)는 반 이상의 Storage Utilization을 제공
  - 최악의 경우 $O(\log_n(N+1))$의 높이 및 탐색 시간 ($n = m/2$)

B-트리의 노드 구조 (2/2)

B-트리의 예 (3-way)



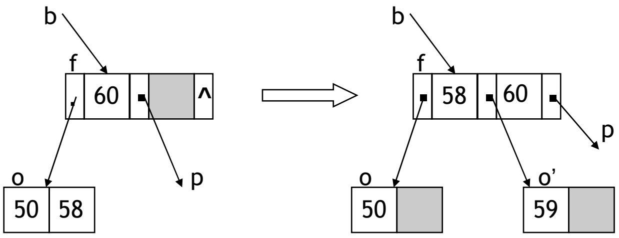
B-트리에서의 연산

- **검색**: m-way search tree의 검색과 같은 과정
  - 직접 탐색: 주어진 키 값을 사용하여 tree traverse (중간에 검색이 종료되기도 함)
  - 순차 탐색: inorder traversal을 수행해야 하므로 성능이 좋지 않음 (→ B^+-트리)
- **삽입**: 항상 단말 노드에 삽입
  - 단말 노드에 여유 공간이 있는 경우: 단순히 순서에 맞도록 단말 노드 내에 삽입
  - 여유 공간이 없는 경우: overflow, split 발생
    - 해당 노드를 두 개의 노드로 분할해야 함
    - 새로운 키 값을 포함하여 키 값들을 정렬한 후, 중간 키 값을 중심으로 큰 키들은 새로운 노드에 저장하고, 나머지는 기존 노드에 저장
    - 중간 키 값: 분할된 노드의 부모 노드로 올라가 삽입 (recursion)
    - 그 결과, 다시 overflow 발생하면, 위와 같은 분할(split) 작업을 반복

B-트리에서 삽입의 예 (1/4)

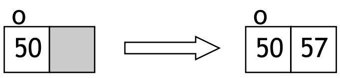
- 앞서 예를 든 B-트리에 새로운 키 값 22, 41, 59, 57, 54, 44, 75, 124, 122, 123을 차례로 삽입

(a) 단말 노드 l에 키 22 삽입　　(b) 단말 노드 n에 키 41 삽입

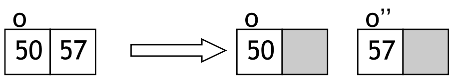


(c) 노드 o에 키 59 삽입 → Split 발생

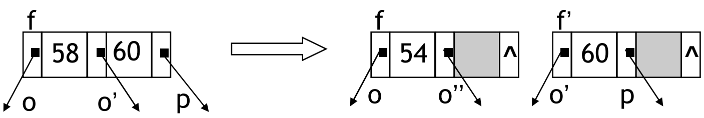
B-트리에서 삽입의 예 (2/4)



(d) 노드 o에 키 57 삽입

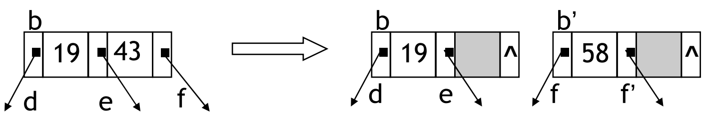


(d) 키 54 삽입으로, 노드 o의 분할
(54는 부모 노드 f로 이동)

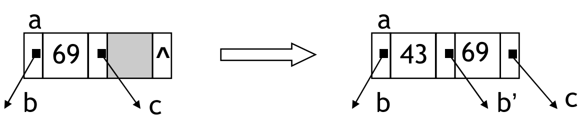


(e) 노드 f에 키 54의 삽입 (58은 부모 노드 b에 삽입)

B-트리에서 삽입의 예 (3/4)



(g) 노드 b에 키 58 삽입 (43은 부모 노드 a에 삽입)

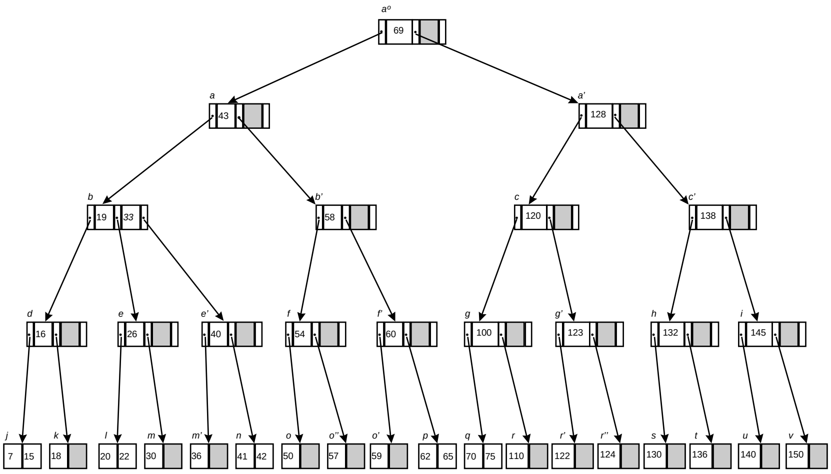


(h) 노드 a에 키 43 삽입

- 나머지 키 값인 33, 75, 124, 122를 차례로 삽입: 문제(overflow)가 발생하지 않음
- 마지막 키 값인 123을 삽입: B-트리는 한 레벨 증가됨 (simulation해 볼 것)

B-트리에서 삽입의 예 (4/4)

- 반복된 삽입에 따라서 한 레벨이 증가된 B-트리

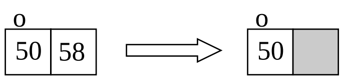


B-트리에서의 삭제

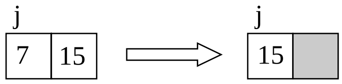
- 삭제될 키 값이 내부 노드에 있는 경우
  - 이 키 값의 후행(successor) 키 값과 교환 후 단말 노드에서 삭제
  - 단말 노드에서의 삭제 연산이 더 간단
  - 후행(successor) 키 값 대신 선행(predecessor) 키 값을 사용할 수도 있음
- 최소 키 값 수($\lceil m/2 \rceil$ - 1)보다 작은 경우: Underflow
  - 재분배(redistribution)
    - 최소 키 값보다 많은 키를 가진 형제 노드(sibling node)를 선택
    - 해당 노드와 선택한 노드의 키 값을 재분배하고, 중간 키 값을 부노 노드의 키 값으로 이동
    - 트리 구조를 변경시키지 않음
  - 합병(merge)
    - 재분배가 불가능한 경우에 적용 (재분배해도 조건인 "$\lceil m/2 \rceil$ - 1"를 만족하지 못하는 경우)
    - 형제 노드와 합병을 수행하여, 합병 결과 빈 노드는 제거
    - 트리 구조가 변경됨 (부모 노드로 삭제가 전이되며, recursive하게 적용됨)

B-트리에서 삭제의 예 (1/3)

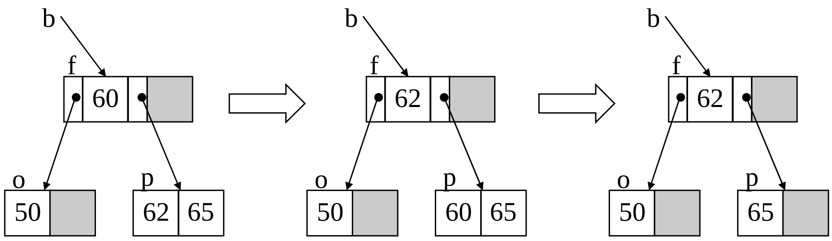
- 앞서 예를 든 B-트리에 키 값 58, 7, 60, 20, 15를 차례로 삭제



노드 o에서 키 값 58의 삭제

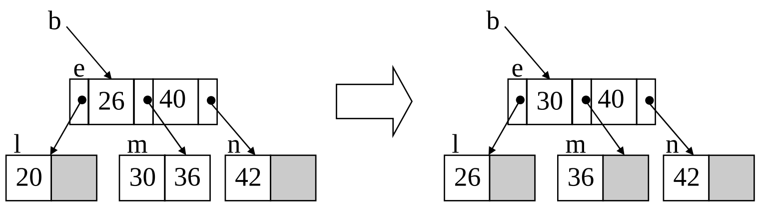


노드 j에서 키 값 7의 삭제



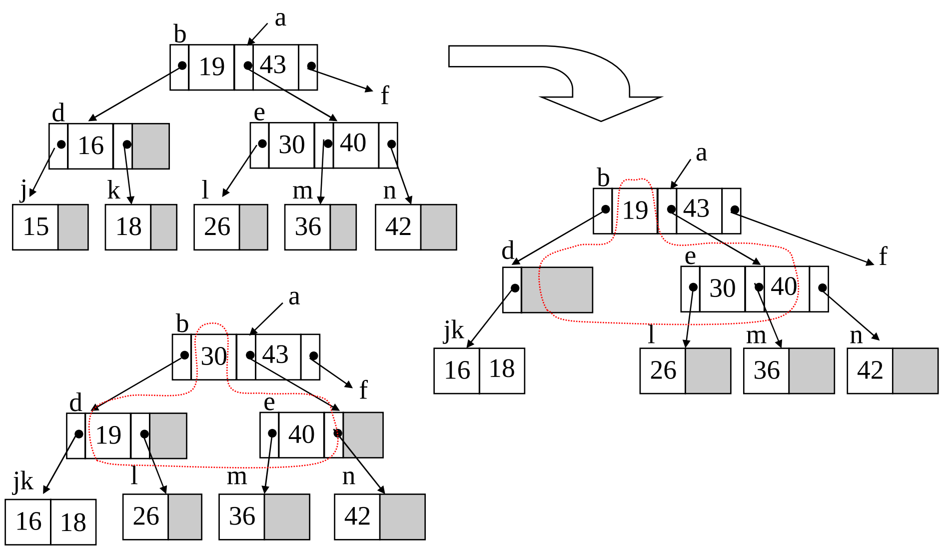
노드 f에서 키 값 60의 삭제

B-트리에서 삭제의 예 (2/3)



노드 l에서 키 값 20의 삭제

B-트리에서 삭제의 예 (3/3)



노드 j에서 키 값 15의 삭제

B*-트리 (1/2)

B-트리의 문제점

- 구조 유지를 위해 추가적인 연산 필요 (잦은 분할, 재분배, 합병이 발생)
- 삽입 → 노드의 분할, 삭제 → 노드의 합병 또는 재분배

B*-트리 : Knuth가 제안한 B-트리의 변형

D. Knuth, *The Art of Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley Publishing Co. Inc., 1973.

- B*-트리 : 공백(empty)이거나 높이가 1 이상인 m-way search tree
- 루트: 단말이 아닌 이상 최소 2개, 최대 2⌊(2m-2)/3⌋+1 개의 서브 트리를 갖는다.
- 루트, 단말 제외한 (내부) 노드: 적어도 ⌊(2m-2)/3⌋+1 개의 서브 트리, 즉, 최소 ⌊(2m-2)/3⌋ 개의 키 값을 갖는다.
- 모든 단말 노드는 같은 레벨에 있다.

→ 노드의 밀도를 높여(branching factor를 크게 하여), 노드의 분할 빈도를 줄임

→ 각 내부 노드는 2/3 이상 키 값으로 채워짐

→ B-트리보다 적은 수의 노드 필요

B*-트리 (2/2)

B*-트리에서 연산의 특징

- 삽입으로 인한 Overflow 발생 시,
  - 즉시 분할하는 대신, 인접한 형제 노드(sibling node)와의 재분배를 실시
  - 재분배가 불가한 경우(형제 노드도 full인 경우), 두 노드와 새로운 노드의 세 개 노드를 사용하여 재분배를 실시

→ 분할을 지연시키고, 노드가 항시 2/3 이상 채워지도록 보장

- 삭제로 인한 Underflow 발생 시,
  - 기본적으로는 B-트리와 유사함 (일단 재분배 이후 필요 시 합병)
  - 합병 시, B-트리와 다른 점은 세 개의 노드를 두 개의 노드로 합병하는 점임

트라이 (Trie)

reTRIEval의 약자

키를 구성하는 문자나 숫자의 순서로 키 값을 표현한 구조

m-ary trie

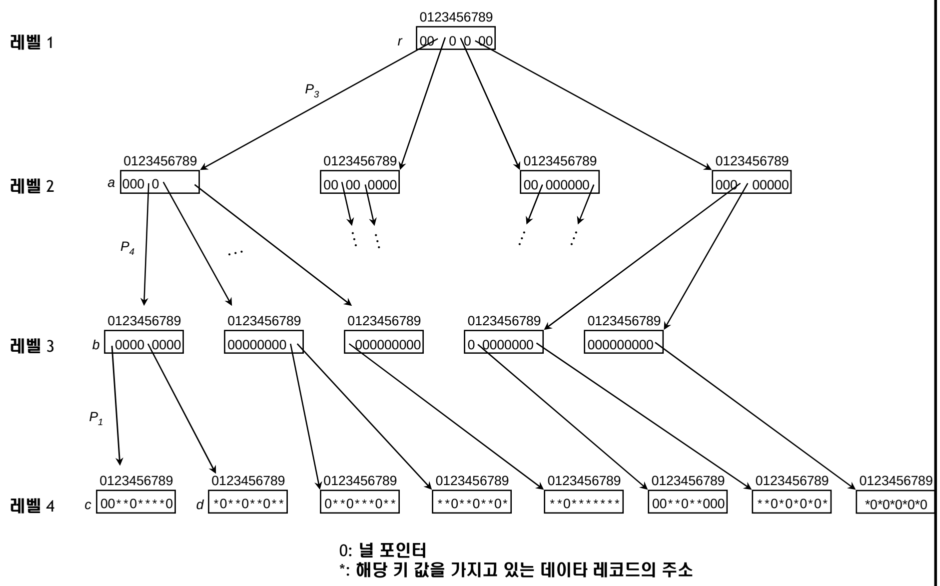
- 차수 m: 키 값을 표현하기 위해 사용하는 문자의 수, 즉 기수(radix)

  예) 숫자: 기수(0~9)가 10이므로 m=10, 영문자: 기수(a~z)가 26이므로 m = 26
- m-ary trie: m개의 포인터를 표현하는 1차원 배열
- 트라이의 높이 = 키 필드의 길이 (예: 키 = 1234 이면, 높이 4)

10진 트라이의 노드 구조

0	1	2	3	4	5	6	7	8	9
P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}

높이가 4인 10-ary Trie



m-ary trie 연산

탐색

- 탐색 종료: 단말 노드에서 성공적으로 종료하거나, 중간에 키 값이 없을 때 실패로 종료
- 탐색 속도 ≈ 키 필드의 길이 = 트라이의 높이 (최대 탐색 비용 ≤ 키 필드의 길이)
- 장점: 균일한 탐색시간(단점 : 저장 공간이 크게 필요)
- 선호하는 이유: 없는 키에 대한 빠른 탐색 때문 (Sparse한 경우에 유리)

삽입

- 단말 노드에 새 레코드의 주소나 마크를 삽입
- 단말 노드가 없는 경우, 새 단말 노드를 생성하고 내부 노드에 연결
- 노드의 첨가나 삭제는 있으나 분할이나 병합은 없음

삭제

- 노드와 원소들을 찾아서 널 값으로 변경
- 노드의 원소 값들이 모두 널(공백 노드): 노드 삭제

강의 내용

- Binary Trees (Ch. 6 in "화일 구조")
- B-tree, B*-tree, Trie (Ch. 6 in "화일 구조")
- B^+-tree (Ch. 7 in "화일 구조")
- Indexed Sequential File (Ch. 7 in "화일 구조")

Basic Concepts

- **Why we use indexing (or hashing)?**
  → We need to speed up access to desired data.
- **Terminology**
  - **(Search) key**: attribute or set of attributes used to look up records in a file.
  - An **index file** consists of records, called index entries, of the form

search key	pointer to the records

  - Index files are typically much smaller than the original file.

Which one is the best?

- **No one technique is the best.**
- **Evaluation criteria**
  - **Access time** (simple query, range query)
  - **Insertion time**
  - **Deletion time**
  - **Space overhead**

Ordered Indexes

- In an **ordered index**, index entries are stored sorted on the search key value.
  - **Primary index**: the index whose search key specifies the placement of the records.
  - **Secondary index**: the index that does not determine the structure of the file or placement of the records.
- **Index-sequential file**: ordered sequential file with a primary index. → good performance for sequential access and random access

Dense Index Files

- In a dense index, index record appears for every search-key value. (i.e., an index entry for every search key)

search key	pointer
Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

branch-name	account number	customer name	balance
Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

To deletion To insertion

Sparse Index Files

- In a sparse index, index record appears for some search-key values. (i.e., an index entry for multiple search keys)

search key	pointer
Brighton	
Mianus	
Redwood	

branch-name	account number	customer name	balance
Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

To deletion To insertion

Dense Index vs. Sparse Index

Dense index	Sparse index
faster access	**slower** access
more space	**less** space
more maintenance overhead for insertions and deletions	**less** maintenance overhead for insertions and deletions

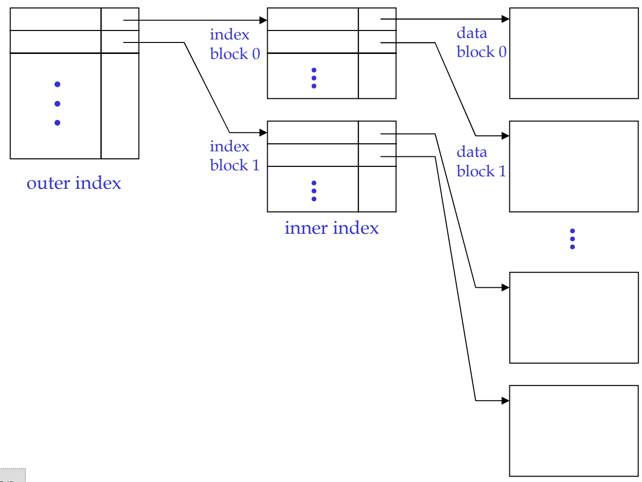
- A good compromise → multilevel index
  - Dense index for index entries to data records
  - Sparse index for index entries to index blocks

Multilevel Index (1/2)

- Index itself can be large → Index does not fit in memory → Access becomes expensive

- Build an index on top of an index
  - Treat primary index kept on disk as a sequential file and construct a sparse index on it.
    - Outer index – a sparse index of primary index
    - Inner index – the primary index file
  - If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

Multilevel Index (2/2)



To insertion

Index Update: Deletion

- If deleted record was the only record in the file with its particular search key value, the search key is deleted from the index also.
- Single-level index deletion:
  - Dense index —deletion of search key is similar to file record deletion.
  - Sparse index
    - If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search key value in the file (in search key order).
    - If the next search key value already has an index entry, the entry is deleted instead of being replaced.
- Multilevel deletion algorithms are simple extensions of the single-level algorithms.

To dense index files　To sparse index files

Index Update: Insertion

- Single-level index insertion:
  - Perform a lookup using the search key value appearing in the record to be inserted.
  - Dense index —if the search key value does not appear in the index, insert it.
  - Sparse index
    - If index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search key value appearing in the block is inserted into the index.
- Multilevel insertion algorithms are simple extensions of the single-level algorithms.

To dense index files | To multilevel indexes

Secondary Indexes (1/2)

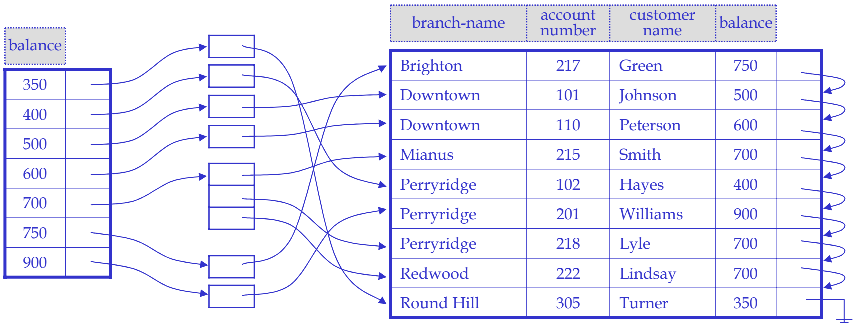
- In general, only one primary index can be created for one data file.
- However, there are needs to find all the records whose values in a certain field, which is not the search key of the primary index, satisfy some condition.
  - **select * from *account* where *customer-name* = "Johnson"**
  - **select * from *account* where *balance* > 700**

search key	pointer
Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

branch-name	account number	customer name	balance	
Brighton	217	Green	750	
Downtown	101	Johnson	500	
Downtown	110	Peterson	600	
Mianus	215	Smith	700	
Perryridge	102	Hayes	400	
Perryridge	201	Williams	900	
Perryridge	218	Lyle	700	
Redwood	222	Lindsay	700	
Round Hill	305	Turner	350	

Secondary Indexes (2/2)

- In a secondary index, an index record points to a bucket that contains pointers to all the actual records with that particular search key value.



- Secondary indexes improve the performance. However, they impose a series overhead on modification of the database.

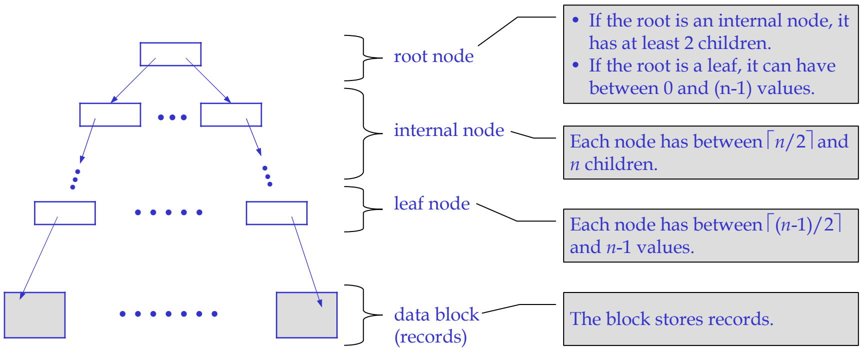
Why B⁺-Tree?

- Disadvantage of index-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.

- As an alternative to index-sequential files, B^+-tree
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions and
  - does not require of reorganization of entire file to maintain performance.
- B^+-tree index files are used extensively in practical areas.

B$^+$-Tree Index Files

- A B+-tree is a rooted tree that takes the form of a *balanced* tree in which every path from the root to a leaf is of the same length.



B$^+$-Tree Node Structure

- Typical node of a B$^+$-tree

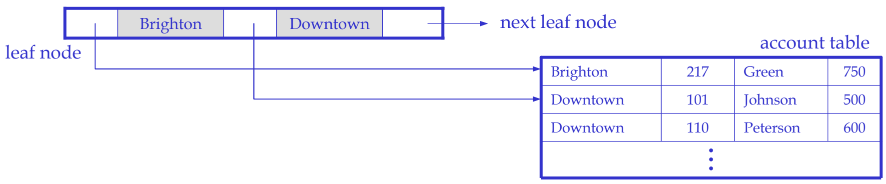
P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - K_i are the search key values.
  - P_i are pointers.
    - In case of internal nodes (non-leaf nodes), they point children nodes.
    - In case of leaf nodes, they point records.
- The search keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots . < K_{n-1}$$

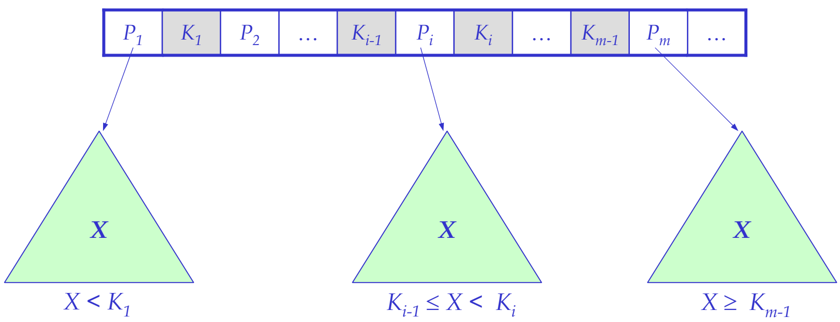
Leaf Nodes in B$^+$-Trees

- Properties of a leaf node:
  - For $i = 1, 2, \dots, n\text{-}1$, P_i points to a file record with search key value K_i.
  - If L_i, L_j are leaf nodes and $i < j$, L_i's search key values are less than L_j's search key values.
  - P_n points to next leaf node in search key order.



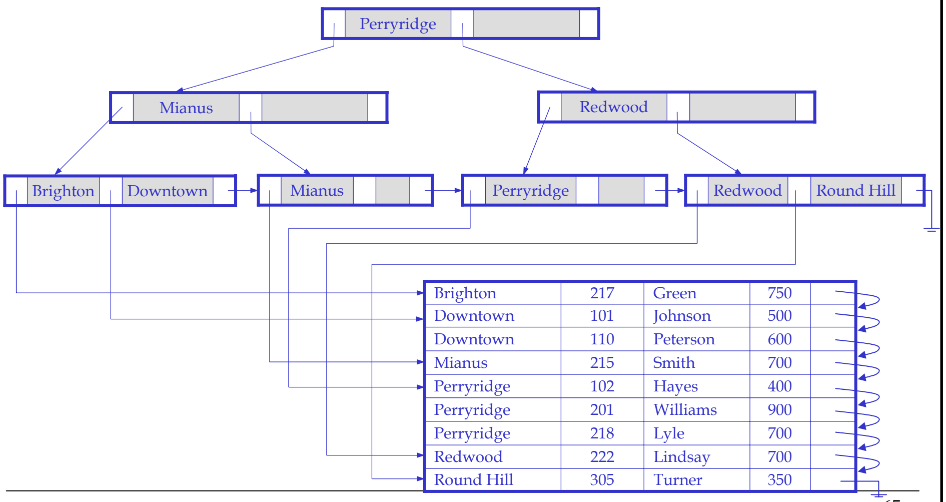
Internal Nodes in B$^+$-Trees

- Internal nodes form a multi-level sparse index on the leaf nodes.
- For an internal node with m pointers:



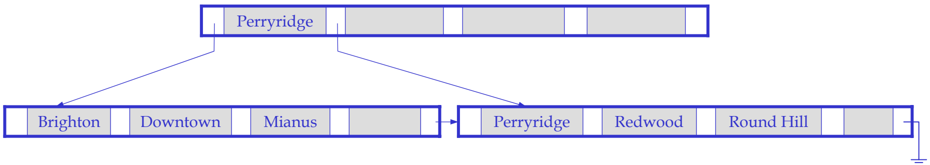
Example of a B$^+$-Tree (1/2)

- B$^+$-tree for *account* file (n = 3)



Example of a B$^+$-Tree (2/2)

- B$^+$-tree for *account* file (n = 5)



- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and n-1, with n = 5).
- Internal nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n, with n = 5).

Observations about B⁺-Trees

- Searches can be conducted efficiently since the B⁺-tree contains a relatively small number of levels.
- Set of leaf nodes forms a simple relation having the search key as an attribute.

Queries on B⁺-Trees (1/2)

- Find all records with a search key value of *k*.

Algorithm Search_B⁺-tree

- Start with the root node
  - Examine the node for the smallest search key value > *k*.
  - If such a value, K_i, exists, then follow P_i to the child node.
  - Otherwise $k \geq K_{m-1}$, then follow P_m to the child node.
- If the child node is an internal node, repeat the above procedure on the node.
- Else, i.e., if the child node is a leaf node:
  - If key $K_i = k$, follow pointer P_i to the desired record or bucket.
  - Else, no record with search key value *k* exists.

Queries on B$^+$-Trees (2/2)

- A path is traversed from the root to some leaf node.
- Searches can be conducted efficiently.
  - For K search key values, the path will be about $log_{\lceil n/2 \rceil}(K)$.
  - The size of a node ≈ 4 KB, $n \approx 100$ (40 bytes per index entry)

    For 1 million search key values, at most 4 (= $log_{50}1{,}000{,}000$) nodes are access in a lookup.
  - In case of a binary tree with 1 million search key values, around 20(= $log_2 1{,}000{,}000$) nodes are accessed in a lookup.

  → The difference is significant since every node access may need a disk I/O.

Updates on B$^+$-Trees: Insertion (1/3)

- Insert a record with a search key value of k.

Algorithm Insert_B$^+$-tree

- Find the leaf node in which k would appear.
- If k is already there in the leaf node, the record is added to file.
- If k is not there, then add the record to the file. Then:
  - if there is room in the leaf node, insert (k, pointer) pair into leaf node at appropriate position.
  - if there is no room in the leaf node, split it and insert (k, pointer) pair.

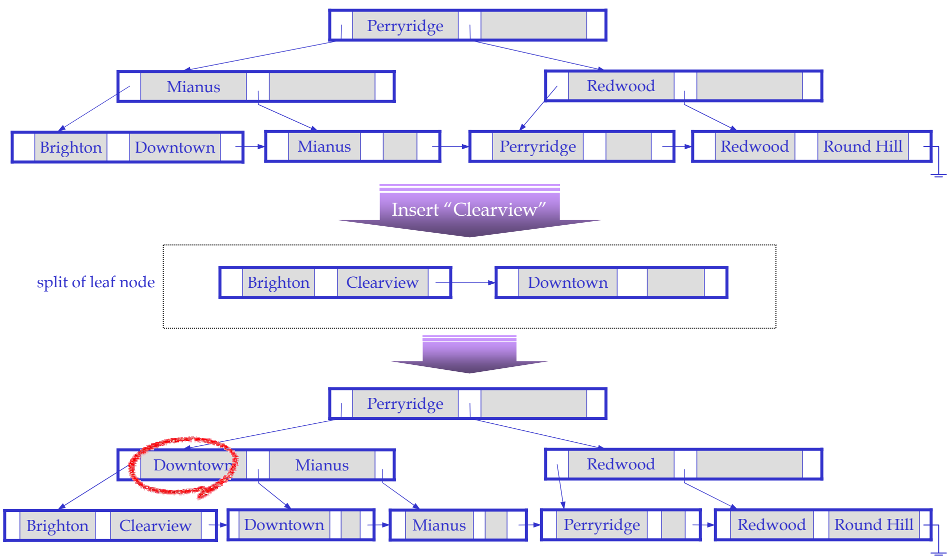
Updates on B$^+$-Trees: Insertion (2/3)

Algorithm Splitting_Node

- Take the *n* (key, pointer) pairs in sorted order.
- Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
- Make an index entry (*k, p*) for the new node.
- Insert the entry (*k*, *p*) in the parent of the node being split.

- If the root node is split, then the height of the tree is increased by 1.

Updates on B$^+$-Trees: Insertion (3/3)



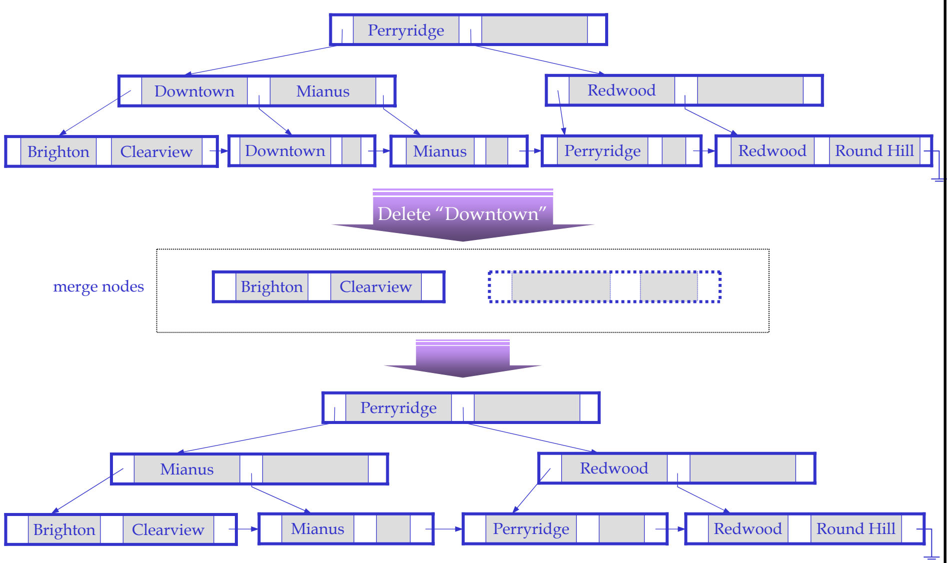
Updates on B$^+$-Trees: Deletion (1/2)

- Delete a record with a search key value of *k*.

Algorithm Delete_B$^+$-tree

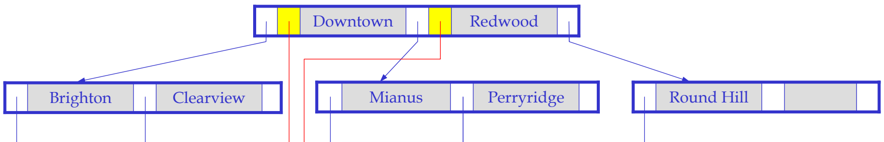
- Find the record to be deleted, and remove it from the file.
- Remove (*k*, pointer) from the leaf node.
- If the node has too few entries($< \lceil (n\text{-}1)/2 \rceil$) due to the removal, and
  - if the entries in the node and a sibling fit into a single node, then merge these nodes into one.
  - If the entries in the node and a sibling do not fit into a single node, then redistribute index entries into these nodes.

Updates on B$^+$-Trees: Deletion (2/2)



B-tree Index Files (1/2)

- Similar to B$^+$-tree, but B-tree allows search key values to appear only once; eliminates redundant storage of search keys.
- Search keys in internal nodes appear nowhere else in the B-tree; an internal node includes an additional pointer field for each search key.



Brighton	217	Green	750
Clearview	225	George	850
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

B-tree Index Files (2/2)

Advantages of B-tree indexes

- May use less tree nodes than a corresponding B$^+$-tree.
- Sometimes possible to find search key value before reaching leaf node.

Disadvantages of B-tree indexes

- Only small fraction of all search key values are found early.
- Since fan-out of internal nodes is reduced, depth is higher than B$^+$-tree..
- Insertion and deletion more complicated than in B$^+$-trees.
- Implementation is harder than B$^+$-trees.

- The advantages of B-tress are marginal for large indexes.
- The structural simplicity of a B$^+$-tree is practically preferred.

Index Structures 강의

강의 내용

- Binary Trees (Ch. 6 in "화일 구조")
- B-tree, B*-tree, Trie (Ch. 6 in "화일 구조")
- B^+-tree (Ch. 7 in "화일 구조")
- Indexed Sequential File (Ch. 7 in "화일 구조")

VSAM 파일 (1/3)

VSAM: Virtual Storage Access Method

B^+-트리 인덱스 구조 기법을 이용하는 대표적인 Index-Sequential File 구성 방법

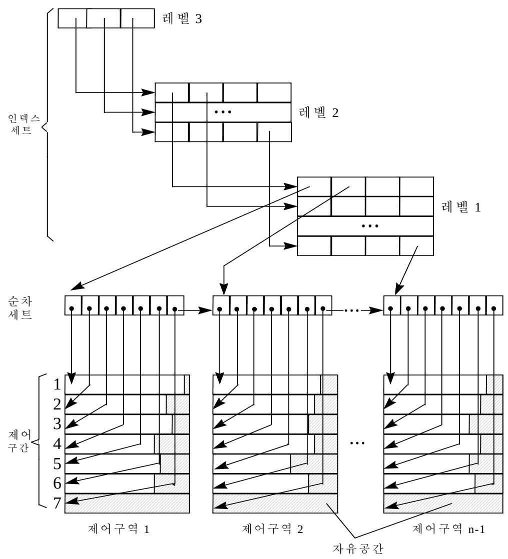
VSAM 파일의 구조

- 인덱스 세트(index set): 순차 세트의 상위 인덱스로서, 여러 단계로 구성
  → Internal Nodes in B^+-tree
- 순차 세트(sequence set): 제어 구역에 대한 인덱스 저장
  → Leaf Nodes in B^+-tree
- 제어 구간(control interval): 데이타 레코드 저장
  → Data Pages(or Blocks) in B^+-tree

c.f.) 다음 슬라이드의 예 참조

VSAM 파일 (2/3)

VSAM 파일의 예



VSAM 파일 (3/3)

제어 구간의 정보 및 구조

- 데이타 블록: 하나 이상 레코드의 저장 공간
- 추가적인 데이타 레코드의 삽입에 대비하여, 자유 공간(free space)을 유지함
- 데이타 블록 내에서는, 키 값에 따라 물리적 순서로 저장
- 제어 정보(control information): 데이타 레코드와 자유공간에 대한 정보

- 제어 구간의 구조

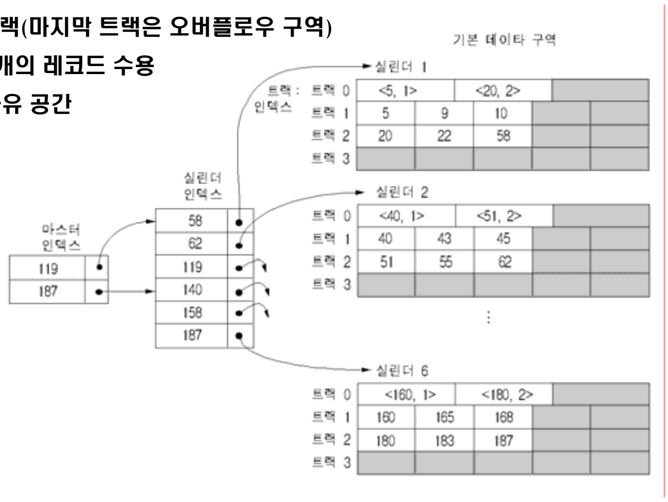
레코드 1	레코드 2	레코드 3	레코드 4
레코드 5	레코드 6	자유 공간	
자유 공간		레코드 제어정보	자유공간 제어정보

ISAM 파일 (1/4)

- ISAM: Indexed Sequential Access Method
- 인덱스, 기본 데이타 구역(prime data area)과 오버플로우 구역(overflow area)으로 구성
  - 인덱스: 마스터 인덱스(master index), 실린더 인덱스(cylinder index), 트랙 인덱스(track index)로 구성
  - 효율적인 삽입, 삭제 처리를 위하여 오버플로우 구역을 별도로 유지하는 점이 특이한 점
  - 많은 UNIX 시스템에서, ISAM 파일을 기본적으로 제공하며, 특히 C-ISAM이란 이름으로, C 언어에서 사용할 수 있는 ISAM Interface를 제공함

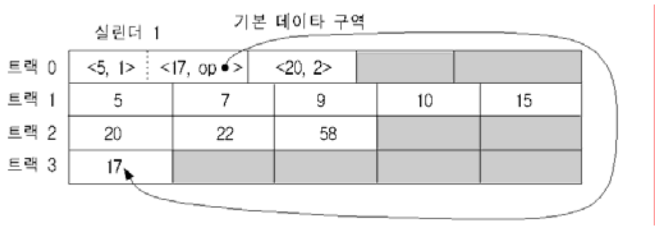
ISAM 파일 (2/4)

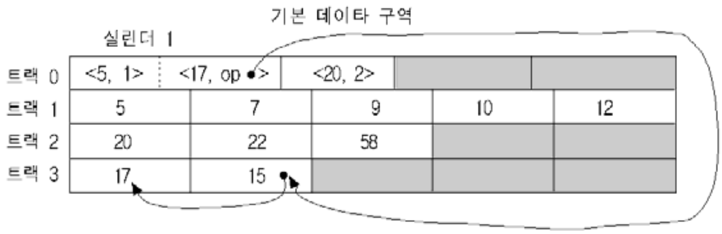
- ISAM 파일의 예
  - 기본 데이터 구역은 6개 실린더로 구성
  - 실린더당 4개 트랙(마지막 트랙은 오버플로우 구역)
  - 트랙당 (최대) 5개의 레코드 수용
  - 트랙당 40%의 자유 공간



ISAM 파일 (3/4)

- 오버플로우가 발생한 예





Index Structures

ISAM 파일 (4/4)

- X/Open 규격(X/Open Company, Ltd., *X/Open Portability Guide*, Prentice-Hall Inc., Dec. 1988.)에 정의된 ISAM 인터페이스
  - isopen, isstart — open a scan
  - isclose — close a scan
  - isread — fetch a tuple (read a tuple)
  - iswrcurr, iswrite — insert a tuple
  - isdelcurr, isdelete — delete a tuple
  - isrewcur, isrewrite — update a tuple
  - isbuild — create a relation
  - isaddindex — create a B^+-tree index
  - iserase — destroy a relation
  - isdelindex — drop a B^+-tree index
  - isindexinfo, isrename, isunlock, and etc.