파일처리 기말 대비

6장

인덱스: 파일의 레코드들에 대한 효율적 접근을 위해 <키 값, 레코드주소(포인터)>쌍을 체계적으로 수집하여 관리하는 보조적인 데이터구조

키 종류에 따른 인덱스 분류: 기본 인덱스, 보조 인덱스

파일 조직에 따른 인덱스 분류: 집중 인덱스, 비집중 인덱스

파일 조직에 따른 인덱스 분류: 밀집 인덱스(역 인덱스), 희소 인덱스(역 인덱스)

균형이원탐색트리: 이원탐색트리에서 루트로부터 리프 노드(터미널 노드)까지의 경로 길이가  **이거나 -1** 일 때 원소의 탐색 시간이 인 트리

AVL(높이 균형 이원 탐색 트리)

삽입과 삭제 연산 시간도 짧으면서 검색 시간도

트리의 일부만 재균형시키면서 트리 전체가 균형을 계속 유지할 수 있도록 트리 형태를 제어

AVL 트리 T의 정의

공백이 아닌 이진 트리

모든 노드의 오른쪽 서브트리 Right(T)와 왼쪽 서브트리 Left(T)의 높이 차이가 1이거나 0

공백 서브트리의 높이는 -1

AVL 트리의 균형 인수(balance factor, BF)

왼쪽 서브트리의 높이 h(Left(T))에서 오른쪽 서브트리의 높이 h(Right(T))를 뺀 수

즉, BF = h(Left(T)) – h(Right(T))

모든 노드의 균형 인수에 따라 +- 1 범위 이내이면 이 노드는 AVL성질을 만족

AVL트리는 N개 노드의 트리에 대해 임의 접근이 시간에 수행

최악의 검색 연산 시간:

삽입과 삭제도 시간에 수행 가능

그러나 AVL 성질 유지를 위해 보통 연산보다 복잡한 연산을 수행해야 함

AVL 트리에서 non-AVL 트리로 변환되는 4가지 원인들(BF가 +-2범위 밖이 되는 원인)

1. LL: 노드의 x의 왼쪽 자식(L)의 왼쪽 서브트리(L)에 삽입
2. RR: 노드의 x의 오른쪽 자식®의 오른쪽 서브트리®에 삽입
3. LR: 노드의 x의 왼쪽 자식(L)의 오른쪽 서브트리®에 삽입
4. RL: 노드의 x의 오른쪽 자식®의 왼쪽 서브트리(L)에 삽입

B-트리

인덱스를 조직하는 트리 구조로 가장 많이 사용

m-원 균형 탐색 트리로서 효율적인 균형 알고리즘 제공

m차 B-트리의 특성

1. 공백이거나 높이가 1이상인 m-원 탐색 트리
2. 루트(root)와 리프(leaf)를 제외한 모든 노드는 최소 ⌈**m/2**⌉, 최대 m 개의 서브트리를 가짐. 따라서, 적어도 ⌈ **m/2**⌉ **-1**개의 키 값을 가짐
3. 루트는 리프가 아닌 이상 적어도 2개의 서브트리를 가짐
4. 모든 leaf는 같은 level에 있음

* **B\*-트리(Knuth가 제안)**
* **B-트리의 오버헤드를 줄이고 삽입과 삭제 연산의 성능을 개선하기 위해 고안된 B-트리의 변형**
* **기본 아이디어는 노드 분할의 빈도를 줄이는 것**
* **B-트리의 오버헤드(문제점)**
* **B-트리의 구조를 유지하기 위해 추가적인 연산이 필요**
* **Overflow 발생 노드는 분할 작업, 노드의 키 값들의 최소 수 유지를 위한 합병 혹은 재분배 작업**
* **m차 B\*-트리의 특성**

1. **공백이거나 높이가 1이상인 m-원 탐색 트리**
2. **루트는 리프가 아닌 이상 최소 2개, 최대 개의 서브트리를 가짐**
3. **루트(root)와 리프(leaf)를 제외한 모든 노드는 적어도 개의 서브트리를 가짐. 따라서 적어도 개의 키 값을 가짐**
4. **모든 leaf는 같은 level에 있음**

* **삽입으로 인한 노드 분할의 빈도를 축소함. 노드가 오버플로 되면 즉시 분할하는 대신**
* **오버플로가 된 키 값과 포인터를 바로 인접한 형제 노드의 여유공간을 이용해 재분배**

**한 노드가 만원이 되더라도 다른 인접 형제 노드가 만원이 될 때까지 분할이 지연됨**

* **인접 형제 노드가 모두 만원이면 두 인접 형제 노드의 키 값들을 3개의 노드로 분할**
* **각 내부 노드는 항상 2/3 이상 키 값으로 채워짐**

**같은 키 값의 수에 대해 B-트리 보다 적은 수의 노드가 필요**

* **7차 B\*-트리와 7차 B-트리와의 비교**



7장

* **B+-트리와 B-트리와의 차이점**
* **인덱스 세트**
* **인덱스 세트의 키 값은 리프 코드에 있는 키 값을 찾아가는 경로만 제공하기 위해 사용**
* **인덱스 세트의 키 값은 모두 순차 세트에 다시 나타나게 됨**
* **인덱스 세트의 노드의 키 값 대신 리프 노드는 <키 값, 그 키 값에 해당하는 레코드에 대한 포인터>가 저장**

**⇒ 인덱스 세트 노드와 순차 세트 노드의 내부 구조는 서로 다름**

**⇒ 인덱스 세트 노드와 순차 세트 노드가 저장할 수 있는 원소의 수가 다름**

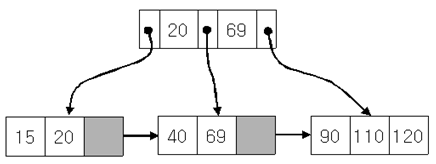
* **순차 세트**
* **순차 세트의 모든 노드가 순차적으로 연결된 연결 리스트로 구성(키 값에 따른 레코드들의 순차 접근 가능)**

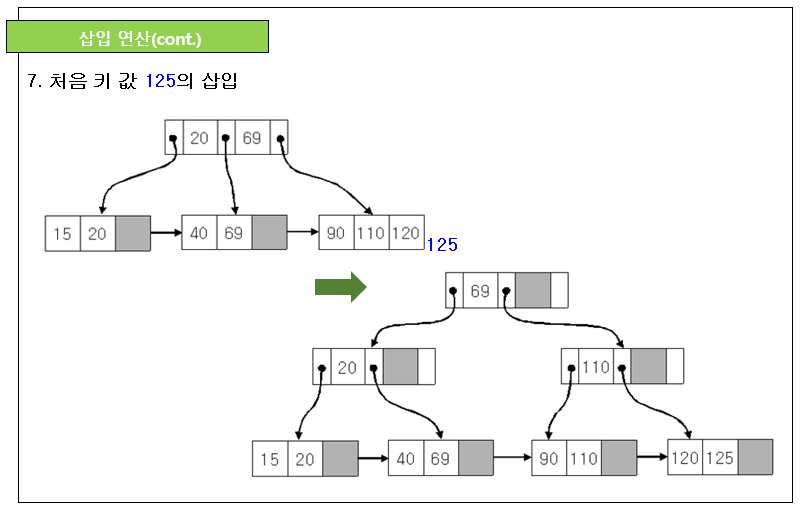
**B+-트리 삽입 dusts**

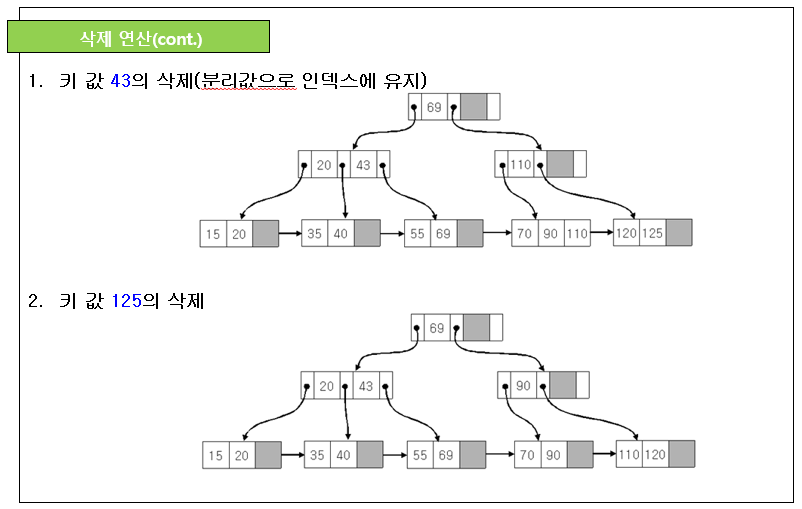
**5. 처음 키 값 20, 120의 삽입**

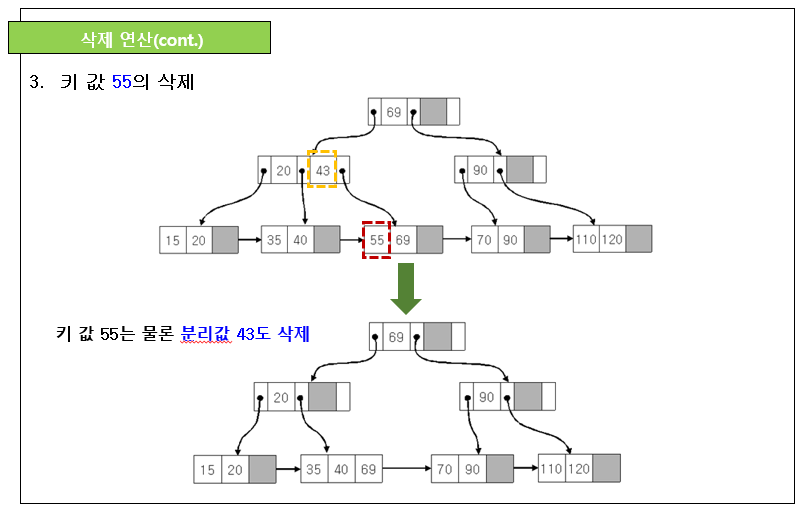
****

**6. 키 값 40의 삽입**

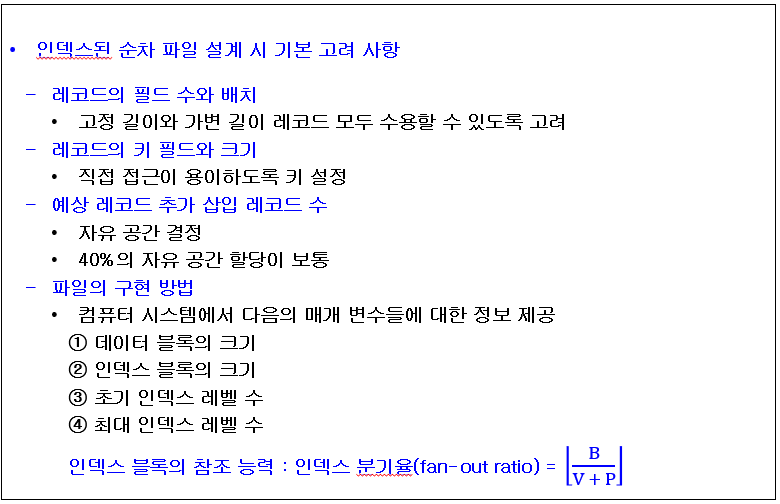


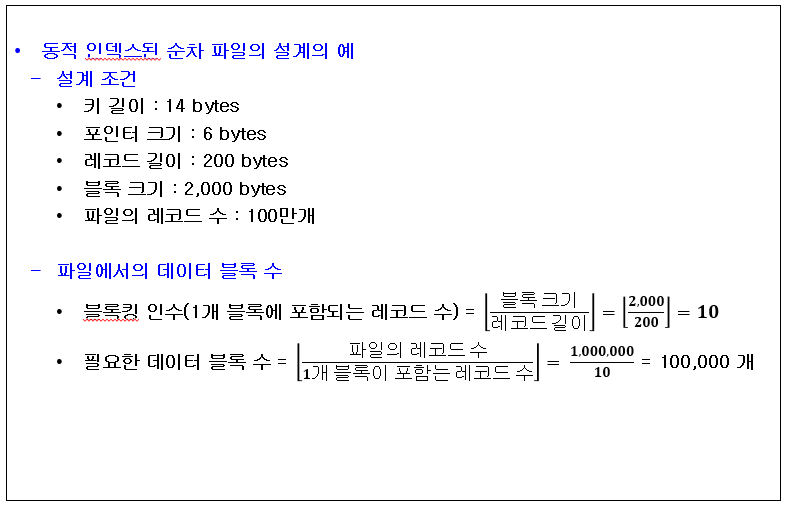


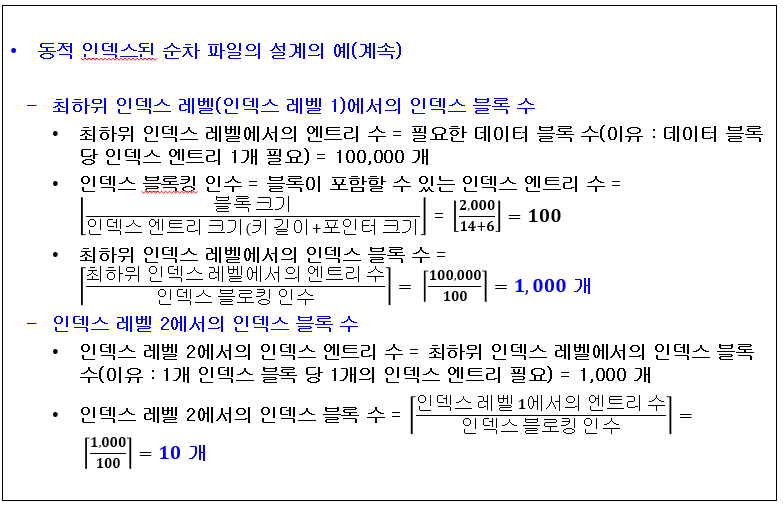


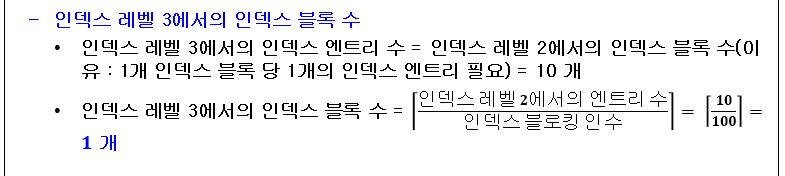


인덱스된 순차 파일의 설계

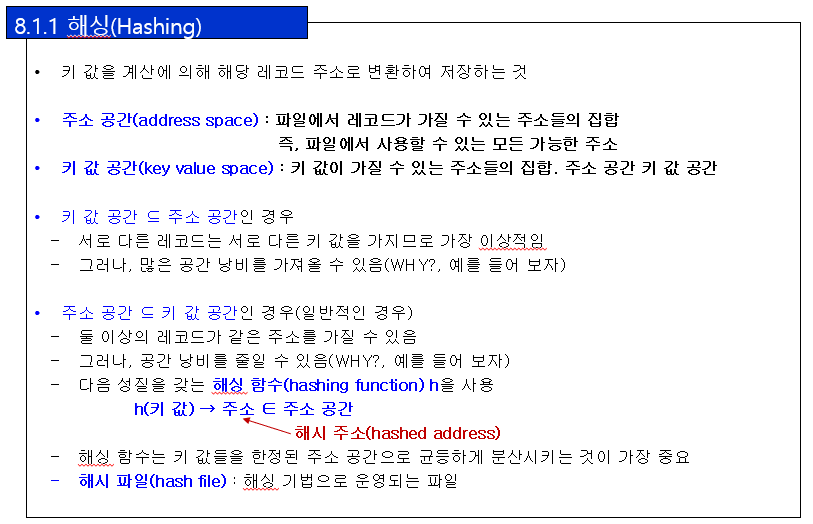


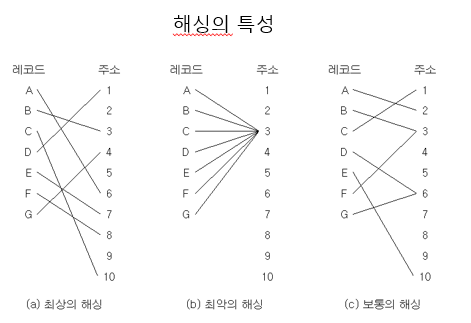


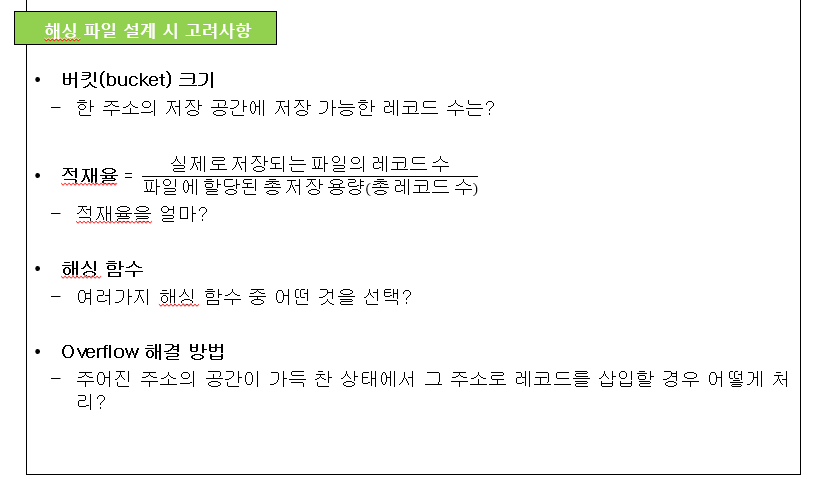




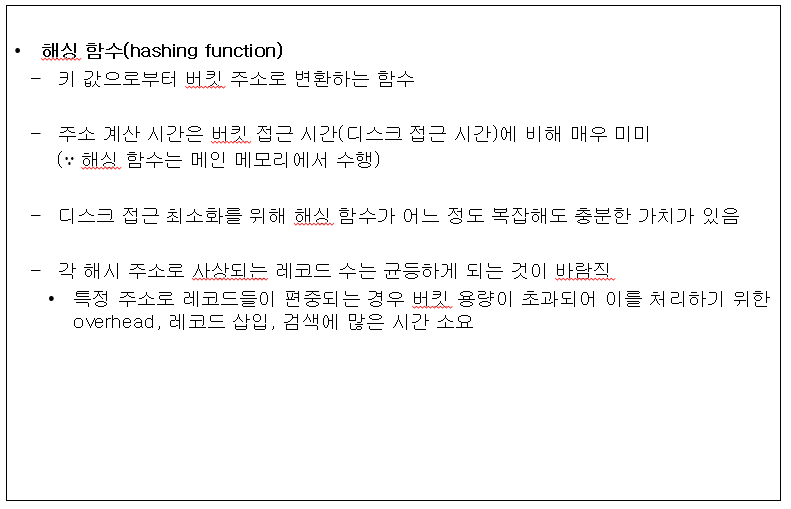
8장



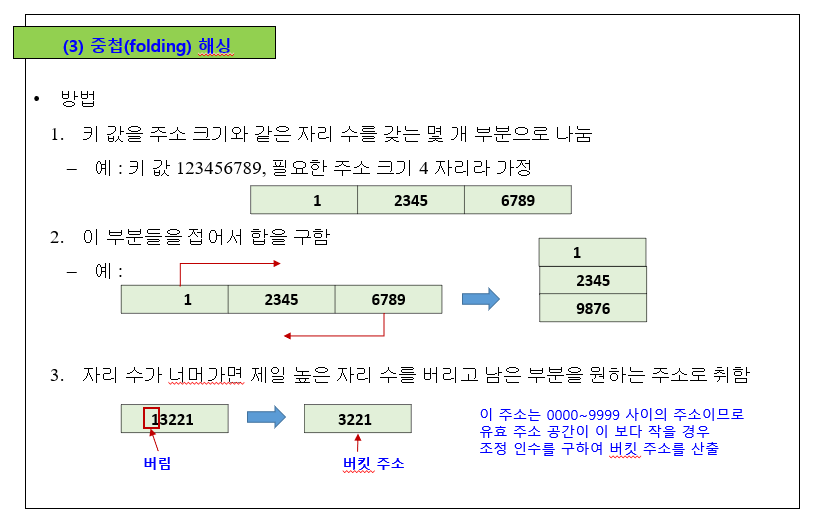




해싱 함수



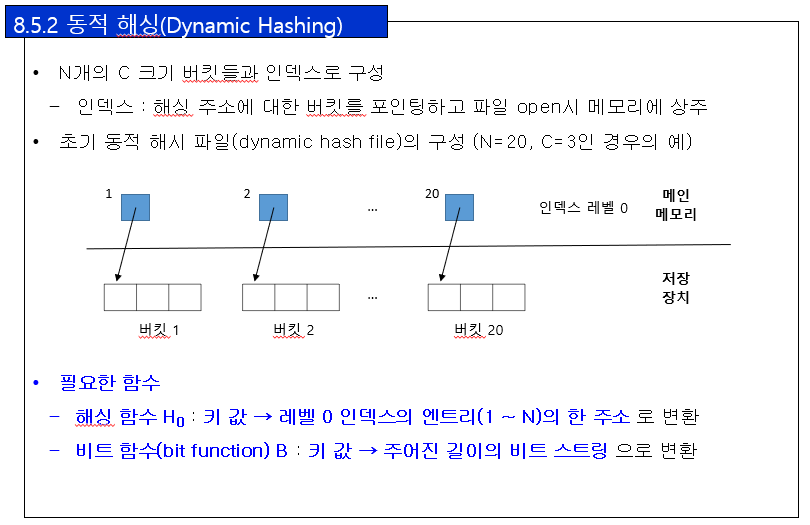


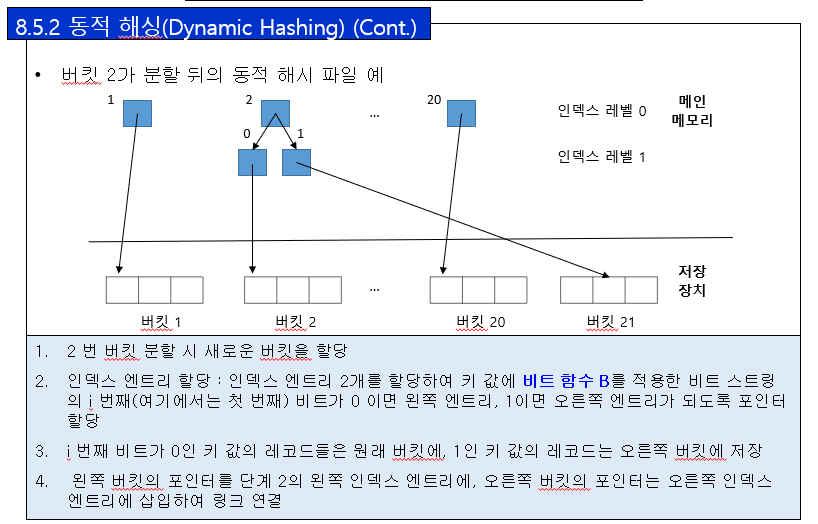


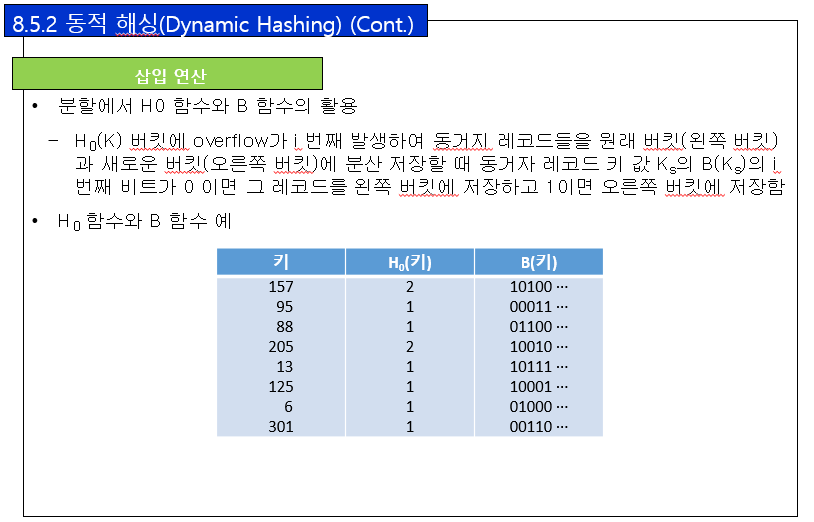
3211\*조정인수 = 레코드가 삽입 될 주소

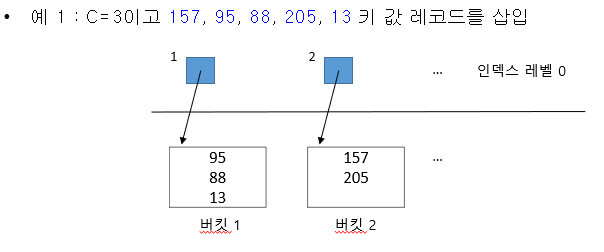
# 조정인수 = 버킷수/10,000

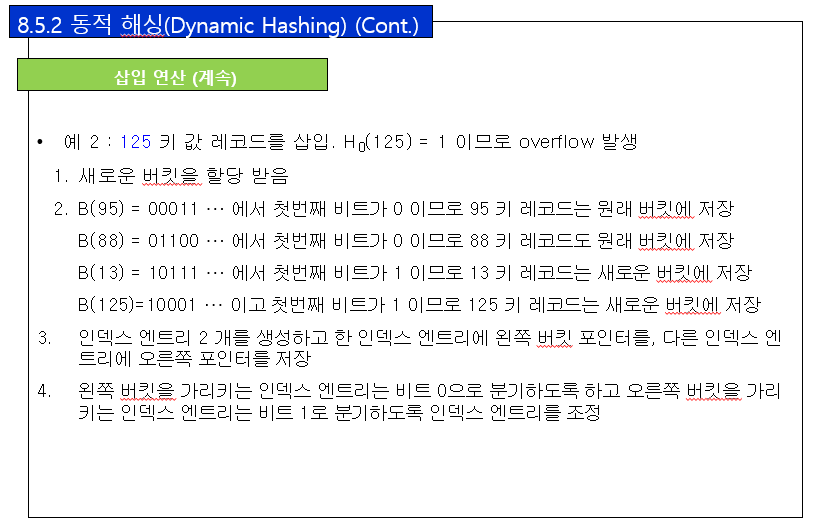
확장성 직접 파일

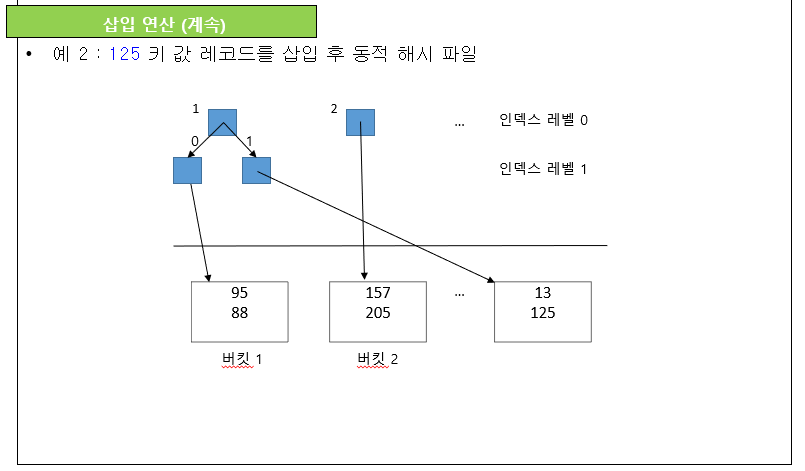


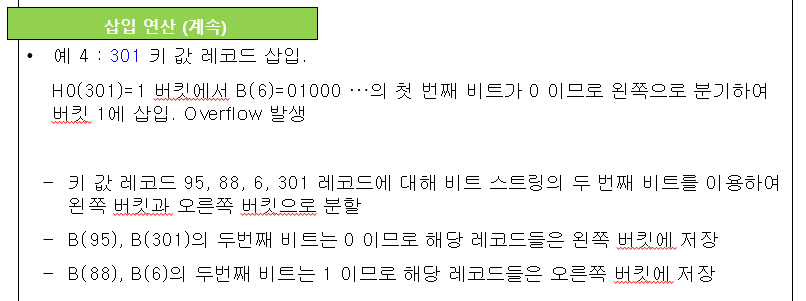


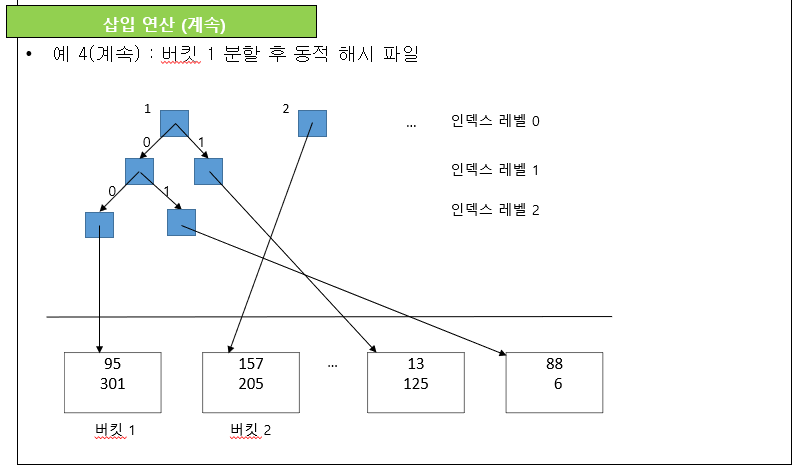


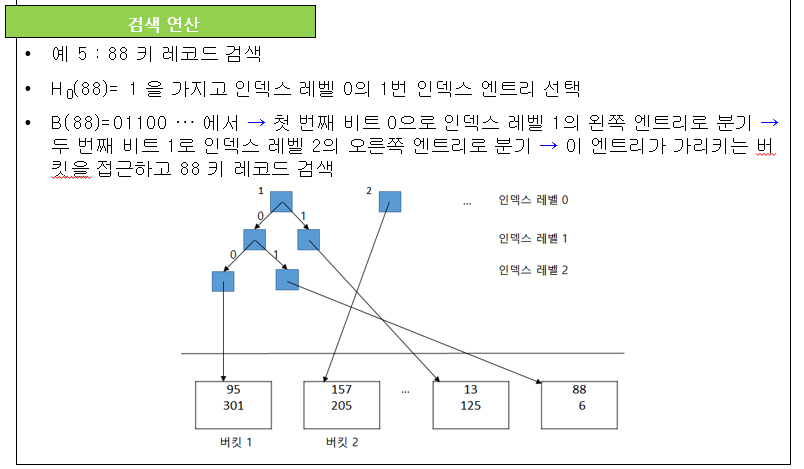




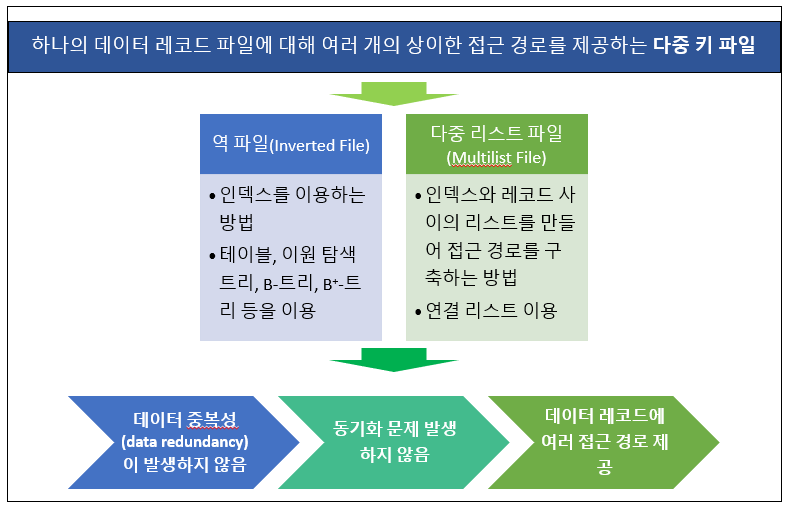




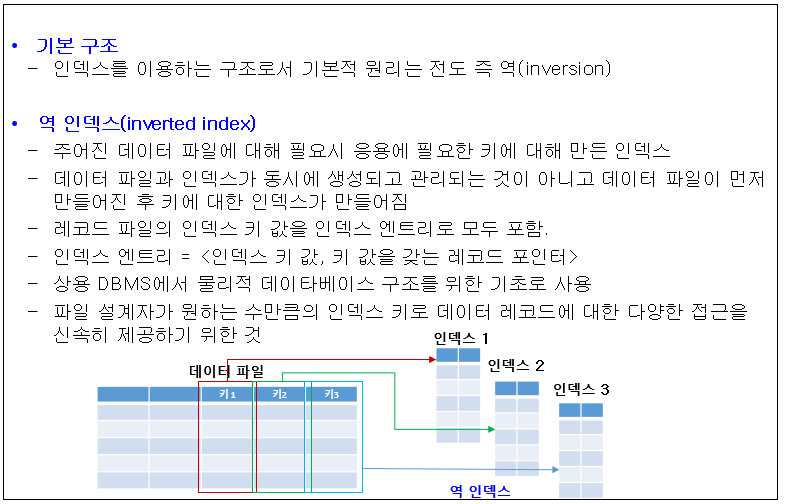




9장



역 파일 구조



다중 리스트 파일 구조

