12장. 탐색 알고리즘

❑ 탐색의 효율

- 삽입, 삭제에 우선하여 탐색의 효율을 높이기 위한 알고리즘

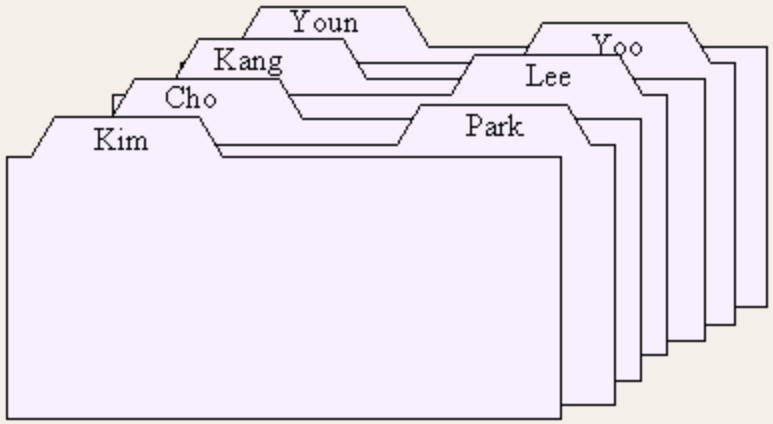
❑ 학습목표

- 이진탐색, 보간탐색, 이진 탐색트리 등 기본 탐색방법의 효율을 이해한다.
- 기수 탐색트리의 두 가지 구현방법과 효율을 이해한다.
- 선형탐사, 제곱탐사, 이중 해쉬의 방법과 효율 차이를 이해한다.
- 버켓, 별도 체인 등 닫힌 어드레싱 방법을 이해한다.
- 상황에 따른 자료구조 선택 방법을 이해한다.

탐색의 대상

- **레코드, 키**
  - 개인별 폴더를 하나의 레코드로 간주
  - 해당 폴더를 찾기 위한 이름은 일종의 키 필드
  - 레코드의 집합에서 주어진 키를 지닌 레코드를 찾는 작업을 탐색
  - 주어진 키 값을 목표 키(Target Key), 또는 탐색 키(Search Key)

- **외부탐색, 내부탐색**
  - 모든 레코드가 메인 메모리 내부에 들어와 있는 상태에서 이루어지는 탐색
  - 레코드가 보조 기억장치, 메인 메모리를 오가며 이루어지는 탐색



이진탐색

- 코드 12-1: 이진탐색

```
void BinarySearch(int A[ ], int First,  int Last,  int Key,  int& Index)
{ if (First > Last)                  탐색 키를 가진 레코드가 없을 때
     index = -1;                     에일리어스 변수 Index를 -1로 표시
  else
  {   int Mid = (First+Last)/2; 가운데 인덱스 계산
     if (Key = = A[Mid])             탐색 키가 가운데 키와 같으면
         Index = Mid;                에일리어스 변수 Index를 해당 인덱스로
     else if (Key < A[Mid])          탐색 키가 가운데 키보다 작으면
         BinarySearch(int A[], First, Mid-1, Value, Index);   왼쪽에서
   찾음
     else                            탐색 키가 가운데 키보다 크면
         BinarySearch(int A[], Mid+1, Last, Value, Index);   오른쪽에
   서 찾음
  }
}
```

보간탐색

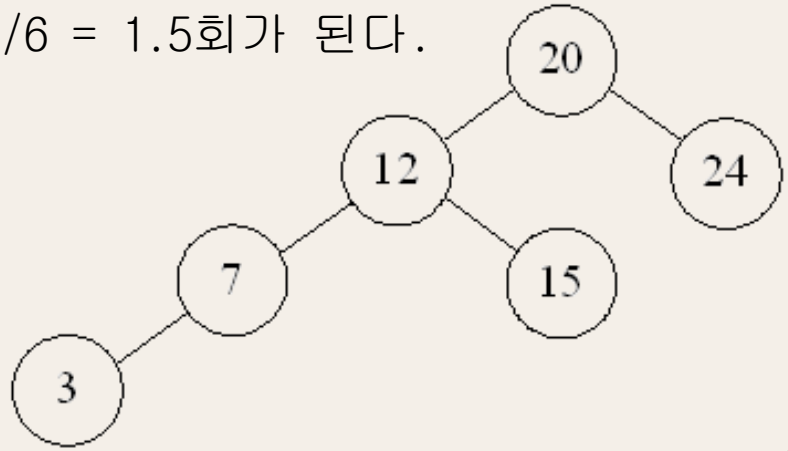
인덱스	1	2	3	4	5	6	7	8	9	10	11	12
키	20	42	55	62	78	92	112	132	140	145	150	170

- 키 값 150인 레코드를 찾기
- 이진탐색
  - 한가운데인 인덱스 (1+12)/2 = 6에서 탐색을 시작
- 보간탐색
  - 1 + (12-1)(150-20)/(170-20) = 10.5. 인덱스 10 또는 11에서 찾기 시작
  - 인덱스 공간 (12-1)을 1로 볼 때, 탐색 키가 (150-20)/(170-20) = 0.86 정도에
  - 크면 오른쪽, 작으면 왼쪽으로 범위를 축소하는 방식은 이진탐색과 동일
- 선형 보간
  - 인덱스 값의 증가에 따라 키 값도 이에 정비례하여 증가한다고 간주
  - 부동소수 나눗셈을 위한 시간
  - 효율 O(lg(LgN))으로서 이진탐색의 O(lgN)보다는 빠르지만 키 값이 선형으로 분포한다는 조건

이진탐색의 효율

- **트리모습에 좌우**
  - 비교적 균형 잡힌 트리의 높이는 lgN. 검색효율은 O(lgN)이다.
  - 최악의 경우 연결 리스트에 가까운 트리라면 검색효율은 O(N)
  - 확률적으로 분석한 이진 탐색트리의 평균 효율은 O(1.38 lgN)

- **평균효율**
  - 트리의 내부경로 길이에 의해 근사적으로 표시
  - 키 3인 노드의 내부경로 길이는 3. 이를 대략적인 비교의 횟수로 간주
  - (Level 0:0) + (Level 1: 1 + 1) + (Level 2: 2 + 2) + (Level 3: 3) = 9.
  - 평균적인 비교회수는 대략 9/6 = 1.5회가 된다.



게으른 삭제

❑ 이진 탐색트리 삭제

- 중위 후속자 또는 선행자를 삭제된 노드로 옮김으로 인해 트리 모습이 변함.
- 삭제가 빈번할수록 트리의 균형이 무너질 확률이 높아짐.

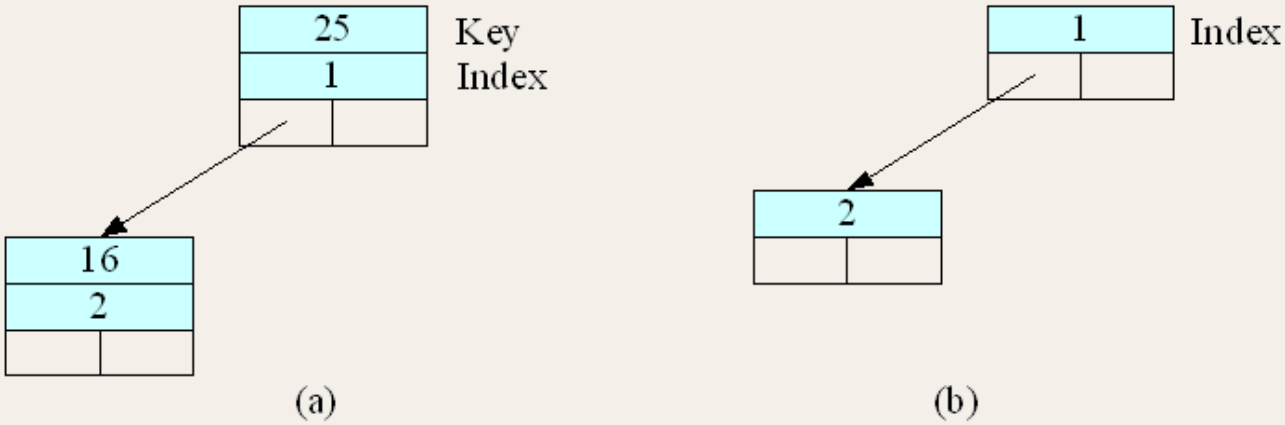
❑ 게으른 삭제(Lazy Deletion)

- 실제로 삭제하지 말고 해당 노드에 “삭제됨” 표시만 함.
- 나중에 몰아서 한꺼번에(Batchwise) 삭제
- 삭제 전까지는 이전의 트리 모습, 이전의 효율을 계속 유지

간접 이진트리

- 레코드는 배열에, 트리는 키 또는 인덱스 정보만 함유
- 삽입, 삭제시간이 감소

Index	1	2	3	...
Key	25	16	12	
Data	Kim	Cho	Park	



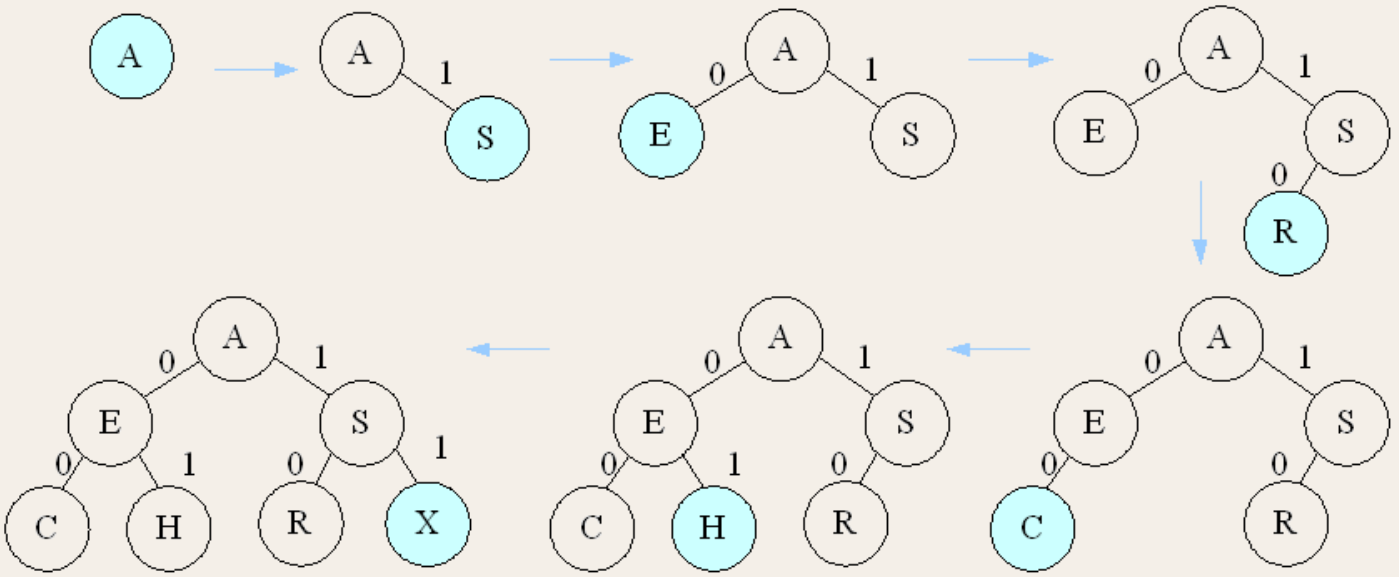
디지털 탐색트리

❑ 디지털 탐색트리(Digital Search Tree)

- 비트 값을 기준으로 자식 노드를 분기
- 5비트 키 필드를 가정하고 알파벳 순서를 비트 값으로 할당
- A는 첫 문자이므로 00001

❑ 입력순서

- A = 00001, S = 10011, E = 00101, R = 10010, C = 00011, H = 01000, X = 11000

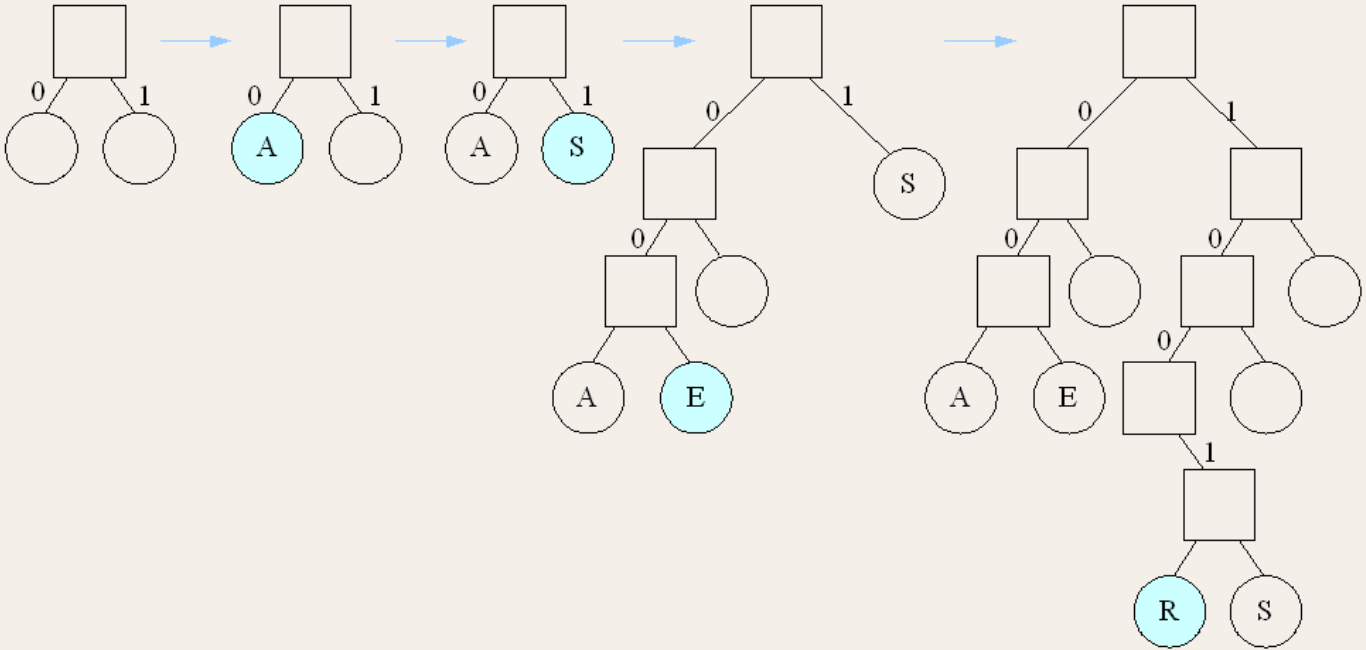


디지털 탐색트리

- **효율**
  - O(lgN)
  - 어떤 키 내부의 비트 값이 0이 될 확률이나 1이 될 확률이 1/2로 동일
  - 새로 삽입되는 노드가 왼쪽으로 삽입될 확률이나 오른쪽으로 삽입될 확률이 거의 동일하므로 균형 트리. 트리 높이는 lgN에 근접
  - 최악의 경우에는 비트 수만큼 O(N)

기수탐색 트리

- 일명 트라이(TRIE)
  - 탐색 또는 검색을 의미하는 ReTRIEval의 가운데 스펠
  - 디지털 탐색트리의 레코드가 내부노드 또는 외부노드 모두에 분포
  - 트라이의 레코드는 모두 외부노드에 분포
- 입력순서
  - A = 00001, S = 10011, E = 00101, R = 10010
  - E가 들어올 때, 첫 비트가 0이므로 루트에서 왼쪽으로. A와 E의 키 비트열을 비교. 두 번째 까지는 00, 세 번째 비트에서 서로가 달라짐. 두 번째 비트까지 경로를 공유해야 하므로 그 경로 상에 내부노드를 추가.



트라이

균형 잡힌 트라이

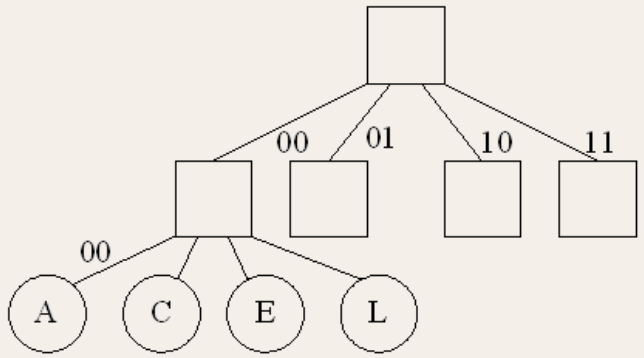
- 효율은 O(lgN)
- 이 경우의 효율은 키 비교 횟수가 아님.
- 키 자체의 비트 열 중 처음 lgN 개의 비트 값을 검사

트라이의 모습

- 레코드의 키 값의 입력 순서와 무관
- A, R, E, S 순으로 레코드가 들어와도 최종결과는 동일
- 키 내부의 비트 패턴 자체가 삽입위치를 결정
- S와 R처럼 여러 비트에 걸쳐 비트 패턴이 같아지면 트리가 한쪽으로만 분기(One-Way Branch) 되어 균형이 무너짐. 빈 외부노드가 다수 발생하여 공간적 효율이 저하됨.

다중링크 트라이

- 다중링크 트라이
  - 하나의 노드에서 뻗어나가는 링크의 숫자를 M으로 늘림으로써 높이를 줄임
  - 예를 들어, 링크를 4개로 놓고 비트 패턴 00이면 가장 왼쪽, 01이면 그 오른쪽
  - 해쉬보다 더 좋은 효율을 보일 수도 있음. 해쉬에서 키를 구성하는 모든 요소를 검사. 트라이는 키 일부만 검사하고도 원하는 레코드를 찾아갈 수도 있음.
- 공간낭비의 우려
  - 레벨 1에서 1개, 레벨 2에서 4개의 노드가 존재하면 레벨 3에서 $4^2 = 16$
  - 리프 노드 근처에 지나치게 많은 외부노드. 미사용으로 인한 공간 낭비
  - 루트 근처에서는 M을 크게 하고, 리프 근처에서 M을 작게 하여 효율을 개선



해쉬 함수의 필요성

❑ 탐색의 효율

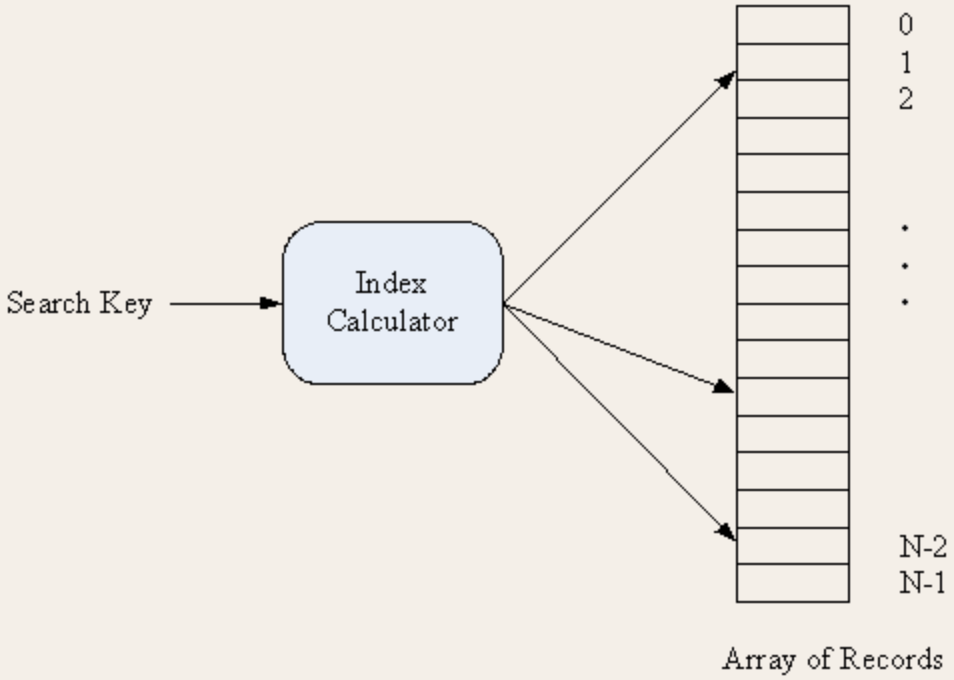
- O(lgN)의 탐색 효율이라면 100,000 개의 레코드 중 하나를 찾는데 필요한 비교의 횟수는 lg100,000 ≈ 17 .
- 매우 빠른 알고리즘. 그러나 이 속도도 느릴 수 있음.
- 119 데이터베이스라면 비상시 전화접속 즉시 주소확인. 예: 실시간 응용 프로그램에서는 검색속도가 결정적.

❑ 해쉬

- 효율 면에서 근본적인 차이를 지향한다. O(logN)이 아니라 O(1)을 지향
- 데이터 개수 N과 무관하게 단번에 찾아내겠다는 것
- 주어진 키를 사용해서 실제 레코드의 주소를 직접적으로 계산해내는 것을 해쉬라고 함. 배열을 사용한다면 이 주소는 배열의 인덱스를 의미
- 해쉬함수를 가하여 채운 도표를 해쉬 테이블(Hash Table)이라 함.

해쉬함수

- 해쉬함수 = 인덱스 계산기



충돌

전화번호 키

- 445-3800이라는 전화번호를 키로 하는 레코드
- 키 자체를 인덱스로 하여 이 레코드를 T[4453800]에 저장
- 이 경우의 해쉬 함수는 자체 매핑(Identity Mapping)
- 메모리 크기는 제한적이므로 배열 인덱스의 범위를 줄일 필요가 있음. 같은 동네라면 국을 생략
- T[3800]에 저장. 이 경우 해쉬 함수는 4453800에서 3800으로 가는 매핑.

스트리핑(Stripping)

- 메모리 크기 인덱스 0...999로 매핑.
- 전화번호의 처음 네 자리를 떼어버리면 T[800]에
- 445-3800 이나 445-7800이나 모두 동일한 인덱스로 매핑
- 충돌(Collision)
  - 서로 다른 키에 해쉬 함수를 가한 결과 동일한 인덱스로 매핑 되는 현상
  - 메모리에 여유 공간이 있음에도 불구하고, 해쉬 함수 자체의 특성으로 인해 발생

해쉬함수

- 자릿수 선택(Digit Selection)
  - 일부 자릿수만 골라냄.
  - 13자리 주민등록번호 중 홀수 자릿수만 선택. h(8812152051218) = 8112528
  - 만약 처음 네 자리를 선택하면 출생 년월이 같은 사람들은 모조리 충돌
- 자릿수 접기(Digit Folding)
  - 각각의 자릿수를 더해 버리는 방식. h(8812152051218) = 8 + 8 + 1 + ... + 8 = 44
  - 메모리 인덱스의 범위는 0 ≤ h(KEY) 0 ≤ 9*13 = 117 의 사이에 존재
  - 메모리가 더 크면 h(8812152051218) = 88 + 12 + 15 + ... + 8 방식도 가능
- 모듈로 함수(Modulo Function)
  - h(KEY) = KEY mod TableSize. Key를 해쉬 테이블 크기로 나눈 나머지를 인덱스로. 해쉬 테이블 크기로 나눈 나머지는 항상 그보다 작으므로 반드시 인덱스 0..(TableSize-1) 안으로 매핑됨.
  - 1236 % 100이나, 3636 % 100이나 모두 동일한 인덱스로 매핑
  - 충돌을 최소화하기 위해서 TableSize 숫자의 선택에 유의. 통계에 의하면 이 숫자는 소수(Prime Number)로 하는 것이 유리함.

문자열 키

자릿수 접기

- 문자 값은 ASCII 코드 값을 할당
- EACH = 69 + 65 + 67 + 72 = 273, WALL = 87 + 65 + 76 + 76 = 304

- 코드 12-2: 문자열 폴딩 I

```
int HashByFold(stringClass Key)
{ int HashVal = 0;
  for (int i=0; i < Key.Length( ); i++)
     HashVal += Key[i];
  return HashVal
}
```

문자열 키

- 코드 12-3: 문자열 폴딩 II

```
int HashByFold(stringClass Key)
{ int HashVal = 0;
  for (int i=0; i < Key.Length( ); i++)
    HashVal = HashVal << 3 + Key[i];
  return HashVal
}
```

- <<
  - 비트 쉬프트 연산자(Bitwise Left Shift Operator)
  - 비트열을 왼쪽으로 3비트 이동
  - 이전 HashVal에다가 8을 곱한 다음에 그 다음 문자 값을 더함.
  - 문자마다 8진법 형태의 자릿수가 있다고 전제
  - ABC라는 문자열을 'A' *8^2 + 'B' * 8 + 'C'로 간주.
  - 이 방식으로 KIM이라는 문자열을 10진수로 나타내면 'K' * 10^2 + 'I' * 10+ 'M'이 됨.

호르너 규칙

- 긴 문자열 키
  - "HWANG" = $8 * 10^4 + 23*10^3 + 1*10^2 + 14*10 + 7$
  - 곱셈의 횟수는 총 10 번
- 호르너 규칙
  - ((((8 *10 + 23)*10 + 1)*10 + 14)*10) + 7 로 변형
  - 곱셈의 횟수는 총 4번. 연산속도 향상.
  - 부동 소수점 연산일 경우에는 계산 결과의 정밀도 향상
- 정수 오버플로우
  - 숫자가 너무 커질 경우.
  - 95 % 7 = ((7 + 2)*10 + 5) % 7 = (2 * 10 + 5) % 7 로 변환
  - 7이라는 인수에 10이나 다른 수가 몇 번 곱해지건 간에 7로 나누어 떨어지므로 모듈로 계산의 결과에 아무런 영향을 주지 못함.
- 긴 문자열의 모듈로(예: "HWANG")
  - 먼저 8을 7로 나눈 나머지 1만 취한다. 여기에 그 다음 숫자인 23을 붙여서 123을 만든다. 이를 7로 나눈 나머지인 4만 취한다. 여기에 그 다음 숫자인 14를 붙여서 414를 만든다. 이를 7로 나눈 나머지인 1만 취한다. 여기에 그 다음 숫자인 7을 붙여 17을 만든다. 이를 7로 나눈 나머지인 3이 전체 키에 모듈로 7을 가한 결과가 된다.

충돌 해결방법

❑ 충돌

- 어떤 해쉬 함수를 사용하든 피해갈 수 없는 문제
- 해결책(Collision Resolution)이 필요.

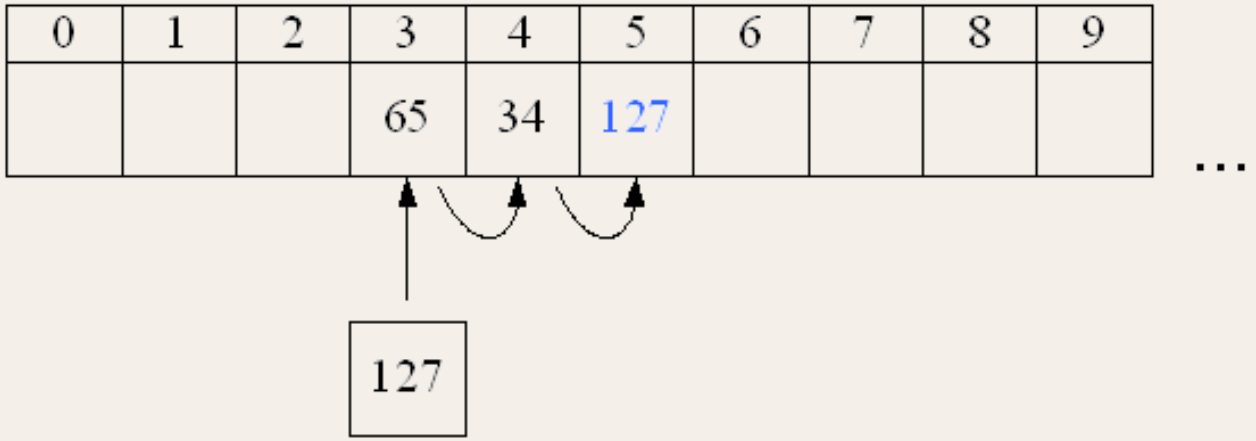
❑ 충돌 해결방법

- 열린 어드레싱(Open Addressing)
  - 충돌이 일어나면 자기 자리가 아닌 곳으로 들어갈 수 있도록 허용
  - 다른 키를 가진 레코드가 해쉬 되어 들어가야 할 자리까지도 오픈
  - 선형탐사, 제곱탐사, 이중해쉬
- 닫힌 어드레싱(Closed Addressing)
  - 자기자리가 아니면 절대로 못 들어가게 함
  - 버켓, 별도 체인

선형탐사

선형탐사(Linear Probing)

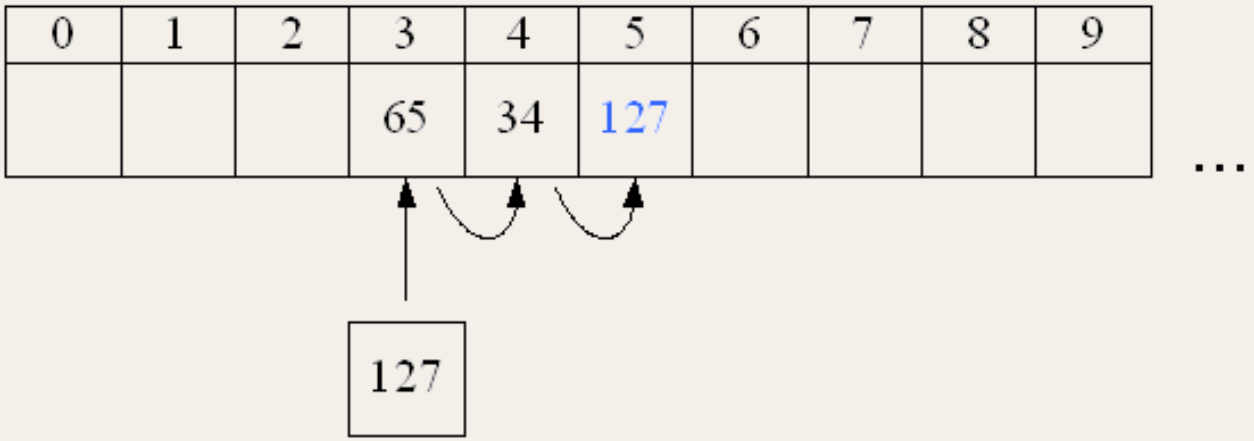
- 충돌이 일어나면 그 다음 빈 곳
- T[h(KEY)]가 점유되어 있을 때, T[h(KEY) + 1], T[h(KEY) + 2], ... 의 순서로 빈 곳이 있을 때까지 찾아감.
- 배열의 끝을 만나면 다시 처음으로 되돌아와서 거기서부터 빈자리를 찾음
- h(key) = h % 31



선형탐사의 태그

필요성

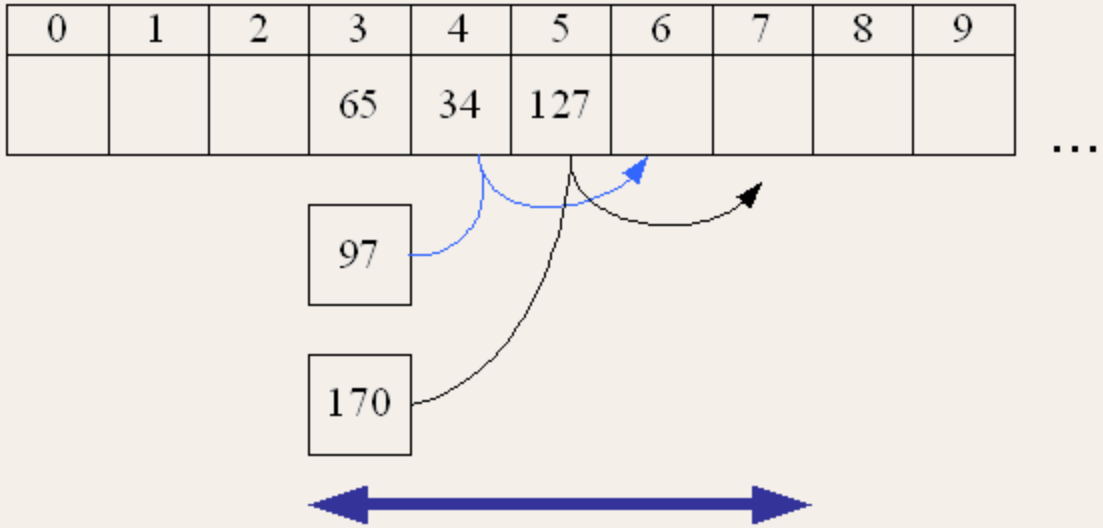
- 65, 34, 127까지 들어간 상태에서 키 34인 레코드를 삭제.
- 인덱스 4의 위치는 빈칸
- 이후, 키 127을 가진 레코드를 탐색시, 인덱스 4가 "빈칸이니 찾는 레코드가 없다"라고 결론지을 수는 없음. 인덱스 5에 찾는 레코드가 있기 때문.
- 배열 아이템을 세 가지 상태로 분류
  - 사용 중(Occupied), 삭제 됨(Deleted), 미 사용(Empty)
- 탐색도중 미 사용 칸을 만나면 탐색을 중지하고 찾는 레코드가 없다고 결론
- 탐색도중 삭제됨 칸을 만나면 탐색을 계속함.



클러스터 현상

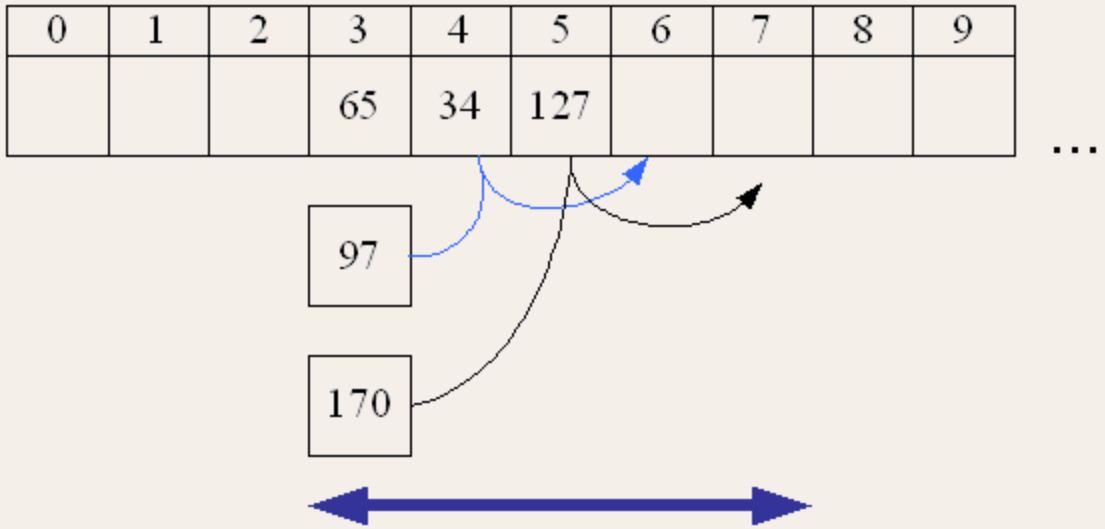
선형탐사의 최대 단점

- 클러스터(Cluster, 군집, 群集) 현상
- 선형탐사의 클러스터는 1차 클러스터(Primary Cluster)
- 클러스터 = 레코드가 분산되지 않고 떼 지어 몰려다님
- 키 34인 레코드는 사실상 34 % 31 = 3에 들어가야 할 레코드가 오른쪽에 밀린 것이고, 키 127인 레코드도 사실은 127 % 31 = 3에 들어가야 하는 레코드.



클러스터 현상

- 97, 170 삽입: 인덱스 3, 4, 5, 6, 7로 이어지는 클러스터가 형성
- 어떤 레코드의 해쉬 결과가 이 인덱스 범위 안으로 들어오기만 하면 그 레코드는 클러스터 끝에 가서 붙고, 클러스터의 크기는 하나 증가
- 클러스터가 커짐으로 인해 이제 어떤 레코드가 클러스터 안으로 해쉬될 확률이 더 높아짐.
- 이런 식으로 클러스터는 급속도로 팽창한다.



부하율

❑ 해쉬 테이블의 부담

- 부하율(負荷率, Load Factor) λ(Lambda)
- 테이블 크기 M, 레코드 개수 N일 때 부하율 λ = N / M
- 테이블 크기가 클수록, 또 레코드 수가 적을수록 부하율은 낮아짐.
- 부하율이 커질수록 클러스터가 형성될 확률이 높아짐

❑ 크누스(Knuth, 1962)

- 선형탐사의 삽입 시 비교회수는 (1 + 1/(1-λ)2)/2에, 탐색 시 비교회수는 (1 + 1/(1-λ))/2에 접근
- 따라서 부하율이 1에 가까울수록 비교 회수는 급증
- 테이블 크기 M을 늘리면 배열의 빈 공간이 많아져 메모리가 낭비
- M이 너무 작으면 작은 클러스터끼리 서로 붙어 큰 클러스터가 형성
- 선형 탐사에서 M ≈ 2N 정도로 할 때 대략 상수시간(Constant Time)에 삽입과 탐색이 가능

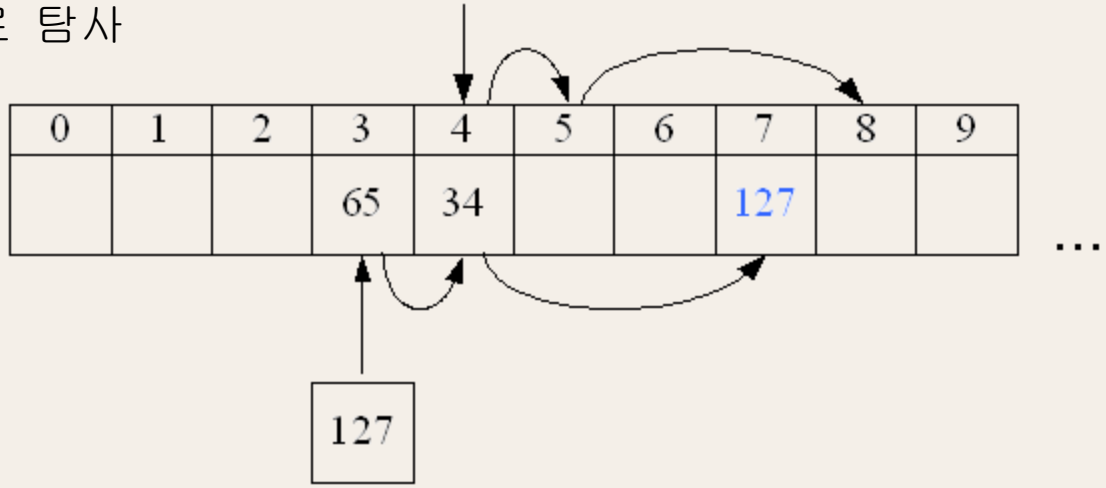
제곱탐사

❑ 제곱 탐사(Quadratic Probing)

- 충돌이 일어날 때 바로 그 뒤에 넣지 않고 조금 간격을 두고 삽입
- T[h(KEY)]가 점유되어 있을 때, T[h(KEY) + 1], T[h(KEY) + 2], T[h(KEY) + 3], ... 의 순서로 제곱 간격을 두고 빈 곳을 찾아감.
- 정확하게는 T[h(KEY) + 1] % M이 다음 찾을 인덱스 위치

❑ 65, 34, 127의 삽입

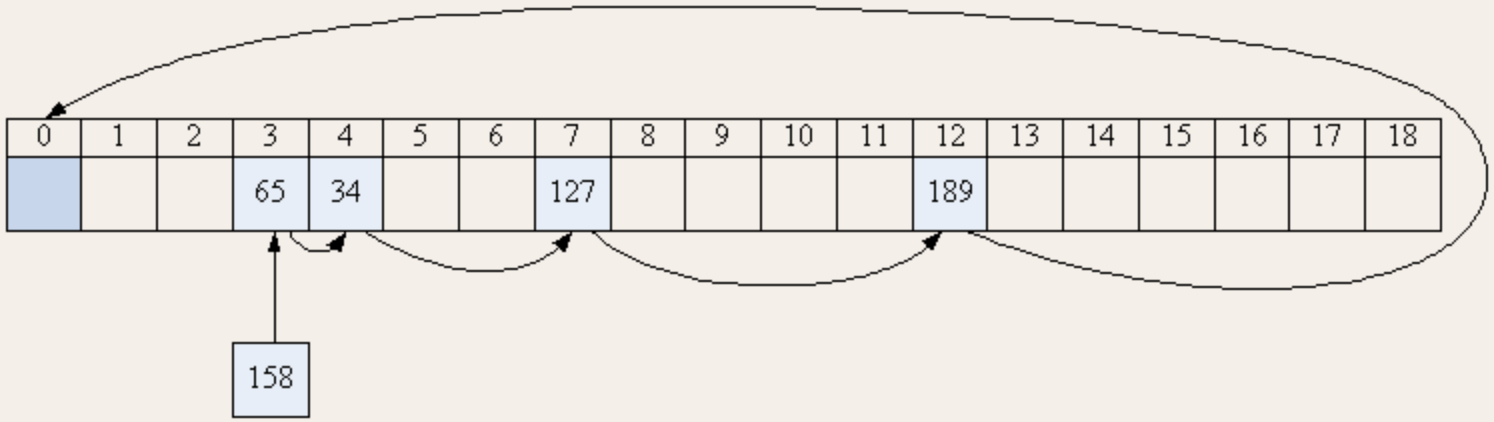
- h(key) = h % 31
- 65인 레코드가 인덱스 3으로 들어감. 키 34인 레코드가 인덱스 3으로 해쉬 어 충돌. 1의 제곱은 1이므로 이 레코드는 인덱스 4로
- 키 127인 레코드가 역시 인덱스 3으로 해쉬됨. 이후 4, 7, 2 순으로 탐사



제곱탐사와 클러스터

❑ 제곱 탐사의 빈 칸

- 중간에 빈 칸을 두는 이유는 그 곳으로 해쉬 되는 레코드들이 들어갈 공간을 비워 둠으로서 오른쪽 끝에 가서 붙는 클러스터를 방지
- 인덱스 5나 6으로 해쉬되는 레코드는 끝으로 밀려나지 않고 제자리를 찾아 먹음. 빈 공간을 찾는 순서가 2, 3, 4의 제곱으로 점프하므로 사이에 넓은 공간이 확보됨.
- 만약 두 개의 레코드가 같은 키로 해쉬 되면 비록 띄엄띄엄 하기는 하지만 같은 위치를 반복해서 찾아가게 됨. 158을 키로 하는 레코드를 삽입하려면 인덱스 3, 4, 7, 12, 0 등 정해진 순서를 따라감.
- 이들 인덱스에 있는 레코드들만으로 볼 때는 여전히 클러스터된 상태이며 이를 2차 클러스터(Secondary Cluster)라 함.



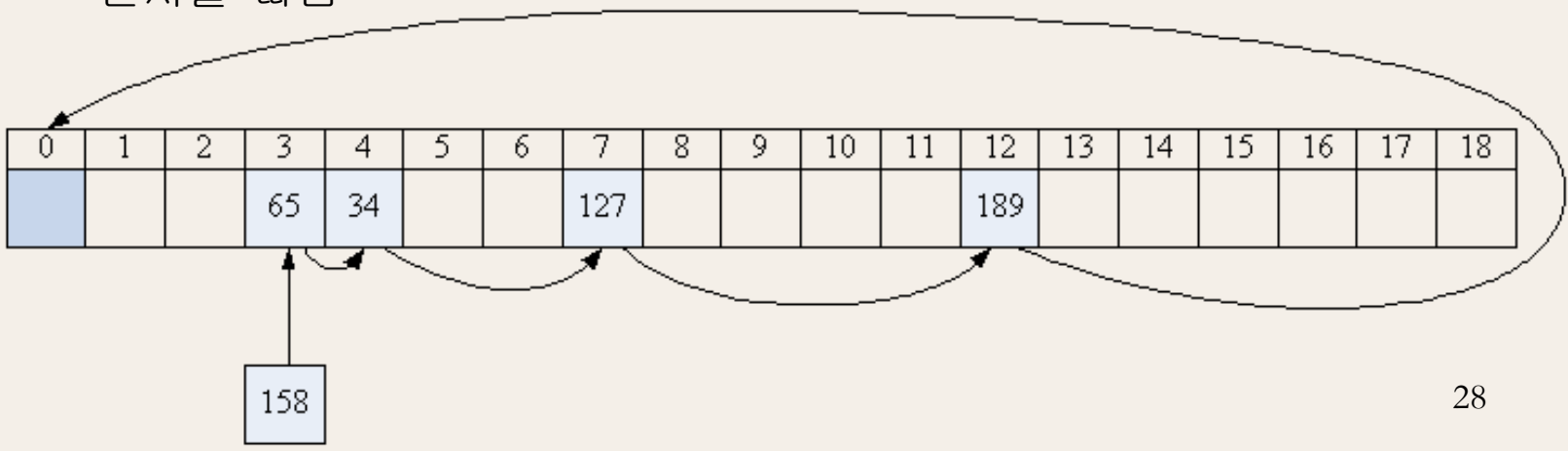
1차 클러스터, 2차 클러스터

❑ 개념상 차이

- 인덱스 X로 해쉬 될 레코드가 충돌에 의해 Y로 밀려나면 X, Y 가 클러스터를 형성
- X, Y 둘 중 하나로 해쉬 되는 모든 것이 클러스터 되는 것이 1차 클러스터
- X 로 해쉬 되는 레코드는 다음 빈 곳을 찾기 위해 동일한 곳을 찾아나가기 때문에 그 순서를 따라서 형성되는 클러스터가 2차 클러스터

❑ 2차 클러스터

- 1차 클러스터보다는 완화된 모습
- 인덱스 4로 매핑 되는 레코드는 4, 5, 8, 13의 순으로 다른 인덱스 순서를 따름



이중해쉬

❑ 이중 해쉬(Double Hash)

- 제곱탐사의 단점인 2차 클러스터를 방지
- 선형 탐색과 제곱탐색에서의 탐사 순서가 키 값과는 무관하게 규칙적
- 키 값에 따라서 탐사순서를 달리함으로써 탐사순서의 규칙성을 제거
- 두 개의 해쉬 함수 h1, h2를 사용
- h1은 주어진 키로부터 인덱스를 계산하는 해쉬 함수
- h2는 충돌 시 탐색할 인덱스의 간격(Step Size)을 의미

이중해쉬

- h1 = KEY % 13, h2 = 1 + KEY % 11
  - 키 14인 레코드의 삽입
    - 14 % 13 = 1에서 충돌. (1 + (14 % 11)) = 4를 간격으로 탐사 순서는 1, 5, 9, 0으로 진행. 인덱스 0에서 빈 곳을 찾았으므로 거기에 삽입
  - 키 27인 레코드의 삽입
    - 27 % 13 = 1. (1 + (27 % 11)) = 6. 탐사 순서는 1, 7, 0, 6 인덱스 6에 삽입
  - 키 18인 레코드의 검색
    - 18 % 13 = 5. (1 + (18%11)) = 8. 탐사 순서 5, 0, 인덱스 0에서 빈 곳이므로 그런 레코드 없다고 결론.
- h2 함수의 선택
  - 어떤 키에 대해서도 h2의 결과는 0이 되어서는 안 됨. 0이면 계속 같은 자리에서 찾는 꼴
  - h1과 동일한 함수여서는 안됨. 모든 키에 대해서 간격이 같아지게 때문.

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72		15		50	

  - 2 러스터를 나 의 해 수를 해 내는 더 른 시간적 제로

선형탐사, 이중해쉬

❑ 선형탐사, 이중해쉬

		부하율			
		50%	66%	75%	90%
선형탐사	탐색	1.5	2.0	3.0	5.5
	삽입	2.5	5.0	8.5	55.5
이중 해쉬	탐색	1.4	1.6	1.8	2.6
	삽입	1.5	2.0	3.0	5.5

버켓

버켓(Bucket)

- 배열 요소가 다시 여러 개의 요소로 이루어짐
- 이 경우 자료구조는 배열의 배열(Array of Array)로서 2차원 배열
- 충돌되는 레코드를 하나의 인덱스 안에 둘 수는 있음
- 해당 인덱스로 가서 다시 키가 일치하는 레코드를 찾아야 하는 시간적 부담
- 사용되지 않는 배열 공간이 낭비됨.

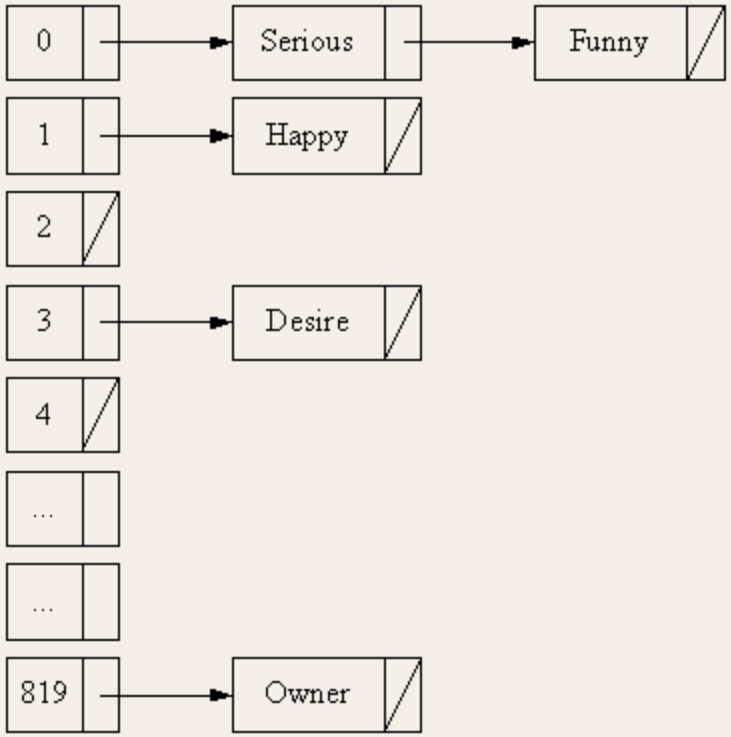
3					4				
65	34	127	189						

...

별도 체인

별도 체인(Separate Chaining)

- 배열 요소가 연결 리스트를 가리키는 헤드
- 일명 분산 테이블(Scatter Table)
- 충돌이 일어날 때마다 해당 레코드를 연결 리스트의 첫 위치에 삽입
- 동적 구조(Dynamic Structure)



Key	Hash Value
Serious	0
Happy	1
Funny	0
Owner	819
Desire	3
...	...

별도 체인

❑ 별도 체인의 효율

- 체인의 길이에 비례
- 평균적으로 체인 길이는 부하율(N/M)과 동일
- 부하율 λ일 때 별도 체인에서의 평균 키 비교회수는 (1 + λ/2)
- 부하율 1 이하의 별도 체인에서는 상수시간 삽입 및 검색이 가능
- 노드 공간을 할당하는 함수를 불러야 하는 시간, 포인터를 따라가는 시간, 포인터 변수 자체가 차지하는 공간을 요함

❑ 버켓과 별도 체인

- 닫힌 어드레싱
- 해쉬 된 인덱스 안에서 다시 배열이나 연결 리스트 등의 또 다른 자료구조를 사용함으로써 충돌을 해결

유니버설 해쉬

❑ 좋은 해쉬 함수

- 레코드를 메모리 공간에 골고루 분포시킴으로써 충돌을 최소화
- 입력 데이터 패턴과 무관하게 이러한 특성을 보여야 함.
- 그러나 실제로 이런 함수를 만들기가 어려움

❑ 유니버설 해쉬

- 다양한 해쉬 함수를 준비해 놓고 선택적으로 사용
- 해쉬 함수의 선택은 랜덤(Random)하게 이루어 짐.
- 응용 프로그램이 시작할 때 하나를 골라서 끝날 때까지 사용
- 유니버설(Universal, 보편적)
  - 테이블 크기 M일 때, 서로 다른 키가 같은 인덱스로 매핑될 확률이 1/M 이하인 해쉬함수

재 해쉬

❑ 재 해쉬(再 해쉬, Rehash)

- 삽입이 계속되면 부하율이 일정 한계를 넘음
- 부하율을 낮추는 테이블 크기 M을 키우는 것.
- 힙 공간에 동적 배열 생성
- 한꺼번에 기존 테이블의 2 배 크기의 배열을 생성
- 이전 데이터를 복사

❑ 재 해쉬를 가하는 시기

- 테이블이 반 정도 찰 때
- 부하율이 미리 정해진 어떤 값을 넘을 때
- 삽입이 실패할 때

자료구조의 선택

❑ 판매촉진 아이디어

- 아이디어가 제기되는 순서대로 리스트에 삽입
- 정렬된 순서로 탐색할 필요가 없음. 정렬 안 된 배열이나 연결 리스트가 유리
- 배열이라면 배열의 마지막에, 연결 리스트라면 연결리스트의 처음에 삽입
- 배열이라면 대략적으로 최대 몇 개의 아이디어가 나올 것인가를 예측

❑ 문서 편집기의 사전기능

- 삽입이나 삭제는 거의 없음, 탐색작업이 위주
- 정렬된 배열의 이진탐색에 의해 O(lgN)의 효율
- 균형 잡힌 이진 탐색트리를 구성해 놓으면 효율은 O(lgN)
- 삽입, 삭제가 거의 없으므로 한번 균형 잡힌 트리의 모습이 변형되지 않음
- 정렬된 연결 리스트: 이진탐색이 어려움.

자료구조의 선택

❑ 도서관의 문헌 카탈로그

- 탐색 위주. 사서에 의한 삽입, 삭제도 적지 않음
- 정렬된 배열은 이진탐색에 의해 O(lgN)의 효율
- 정렬된 배열은 삽입, 삭제에 있어서 크게 불리. O(N)의 이동 때문.
- 정렬된 연결 리스트에서 평균적으로 N/2의 위치에서 찾으므로 대략 O(N).
- 정렬된 연결리스트에서 삽입 삭제는 앞뒤 포인터만 조정하므로 O(1).

❑ 정렬된 배열이 유리한가 정렬된 연결 리스트가 유리한가.

- 이진 탐색트리가 O(lgN)으로서 최적의 효율을 보임.
- 균형 트리에 가까울수록 탐색에 O(lgN)에 가까워짐.
- 삽입 삭제에 걸리는 시간은 인근 노드의 포인터만 조정하면 되므로 O(1)

자료구조의 선택

선택기준 (예시)

- 활용할 수 있는 컴퓨터의 시간적, 공간적 리소스가 충분한가.
- 입력 데이터의 크기와 특성 및 분포는 어떠한가.
- 삽입 삭제 검색 중 어떤 것이 가장 많이 이루어지며 그 작업이 얼마나 빨리 이루어져야 하는가.
- 결과 데이터가 정렬될 필요성이 있는가 없는가.
- 데이터를 순차적으로 접근하는가 아니면 랜덤한 순서로 접근 되는가