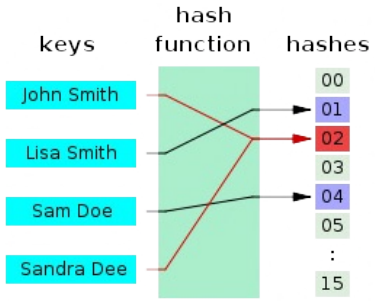
“해시 체이닝"

- 해시 테이블에서의 충돌을 피하기 위한 기법
- 연결리스트로 노드를 계속 추가해나가는 방식
  (제한 없이 계속 연결이 가능하나 메모리 문제가 생길 수 있음)

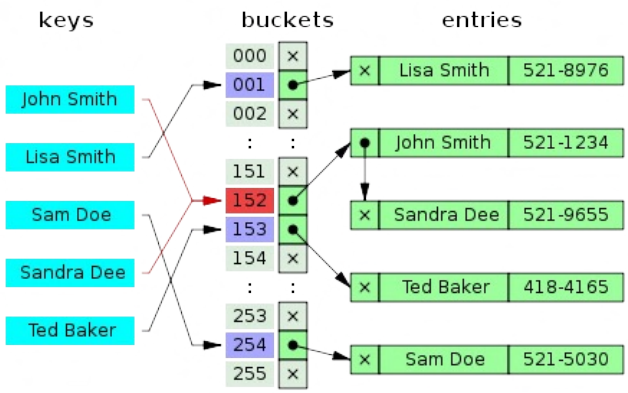
해시 함수



해시함수: 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수

key: 매핑 전 원래 데이터의 값
hash value(해시값): 매핑 후 데이터의 값
hashing: 매핑하는 과정

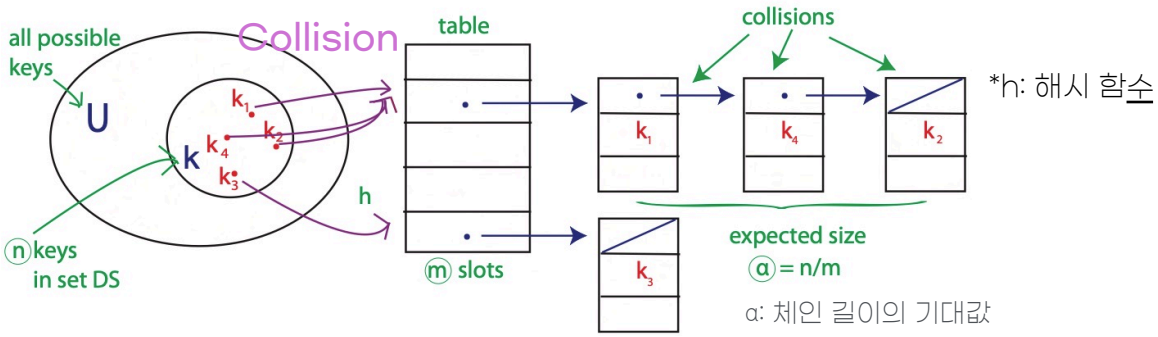
해시 체이닝 적용시



한 버킷당 들어갈 수 있는 엔트리의 수에 제한을 두지 않음으로써 모든 자료를 해시테이블에 담을 수 있음.

해당 버킷에 데이터가 있으면 체인처럼 노드를 추가하여 다음 노드를 가리키는 방식으로 구현한다.

체이닝을 이용한 핵심:



*h: 해시 함수

α: 체인 길이의 기대값

Figure 1: Hashing with Chaining

-> 모든 것이 균등할 때, 특정 키가 슬롯에 들어갈 확률은 1/m, 각각 1/m 확률로 n번의 독립적인 일이 일어나면 총 n/m이 된다.

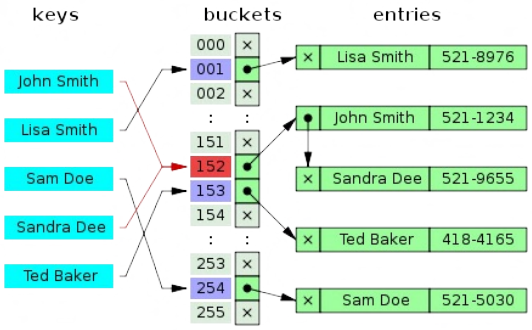
—> 해시테이블의 크기가 m, 실제 사용하는 키의 개수가 n, 해시함수가 키들을 모두 버킷에 균등하게 할당한다고 가정하면, 버킷 하나당 n/m = α개의 키들이 존재할 것이다.

n개의 key를 m크기의 table에 넣는다.
총 크기: n+m
  m: 해시 테이블의 크기
  n: 연결 리스트로 된 총 항목 수

삽입, 탐색, 삭제 연산

해시테이블의 크기가 m, 실제 사용하는 키 개수가 n, 해시함수가 키들을 모든 버킷에 균등하게 할당한다고 가정하면, 버킷 하나당 n/m = α 개의 키들이 존재

Ex) 탐색
“Sandra Dee의 전화번호를 해시테이블에서 탐색한다"



1. Sandra Dee를 해시값(152)로 바꾼다.
2. 버킷 요소들 가운데 Sandra Dee에 해당하는 데이터가 있는지 탐색한다.

-> 키를 해시값으로 바꾸고 해당 해시값에 해당하는 버킷의 요소들 α개를 모두 탐색한다. => O(1+ α)

“기억해야 할 해시 함수”

1. 나머지 함수

$h(k) = k \bmod m$

모든 정수를 0부터 m-1까지 원하는 크기의 집합으로 만들 수 있는 쉬운 함수.

2. 곱하는 방식

$$h(k) = [(a*k) \bmod 2w] >> (w-r)$$

“해시 테이블 크기 설정”

해시 테이블 m의 크기는 어떻게 설정할까??
m의 적정값은??

문제: m 크기의 해시 테이블을 만들고 키를 넣기 때문에 n의 값을 알기 전에 해시 테이블을 만들어야 함.

만약 m이 너무 작으면?
-> α(n/m)값이 커진다 => 1+ α는 더 이상 상수 시간이 아니게 됨 => 느려짐.

m이 너무 크면?
-> 메모리 낭비

우리는 **m이 충분히 커서 빠르게 기능하면서 m이 충분히 작아서 공간이 낭비되지 않기를 원함.**

Idea: 해시 테이블의 크기를 작은 상수로 시작해 필요에 따라 늘리거나 줄인다.

해시 테이블 늘리기: m -> m’
m: 현재 테이블의 크기, m’: 새롭게 늘려진 테이블의 크기

1. Make table of size m’
- 새로운 크기의 해시 테이블을 만들기 위해서는 메모리 할당을 하고 새롭게 해싱을 해줘야 함.
2. Build new hash function h’
3. Rehash:
- m크기의 테이블안에 매핑된 키들을 각 해시 테이블 T’에 삽입한다.

```
for item in T:
    T’ insert(item)
```

Θ(n+m+m’)
- 일반적으로는 해시 테이블에 있는 모든 항목을 돌아봐야하기 때문에 O(m)시간만큼 m개의 슬롯을 찾아가는 데에 써야하고,
- O(n) 시간만큼 각 리스트를 찾아가는데 소요된고
- 새로운 m’크기의 해시 테이블을 만들기 위해 m’ 시간이 소요된다.

늘리는데 소요되는 시간?
If m < n: grow table!
접근 1) m’ = m + 1:
cost of n inserts
=> 매번 새롭게 늘려야한다.
=> Θ(1+2+…+n) = Θ(n^2)

접근 2) m’ = 2m(테이블 더블링):
cost of n inserts
=> 2^i번 마다 새롭게 해시 테이블을 만들어준다.
=> Θ(1+2+4+8+…+n) = Θ(n)

Array Doubling?

상황)
연산 수행 중 배열이 꽉찼다. 배열의 크기를 늘리고자 한다.
어떤 상수 c만큼 늘릴까 아니면 두 배 만큼 늘릴까?

해결) 2배 만큼 늘린다.(훨씬 더 효율적이고 빠르다.)

Ex) 어떤 방이 존재. 방의 수용인원은 60명으로 제한됨.

60명의 사람들이 방에 들어간 상태에서 61번째 사람이 방에 들어가고자함.

-> 못들어감. 수용인원 제한때문에.

-> Array Doubling 전략 사용하여 수용 인원을 두배인 120명으로 늘린다.
120명을 수용할 수 있는 새로운 방을 마련한다.

Cost 계산
기존 방(A): 60명 수용 가능
새로운 방(B): 120명 수용 가능.
한 사람을 옮기는 비용을 t(transferring cost)라 한다.

60명을 옮기는데 드는 비용: 60t
만약 n명이라면 n*t
—> 이건 “현재” 지불해야 되는 transferring cost.

이전에도 더블링이 일어났을 것이므로 총 cost는 n*t이 아닐 것임.

30명일때의 transferring cost는 30t
…
-> 현재 크기 n까지 오기 직전에 transferring cost의 총 합 C는
t*(n/2)+t*(n/4)+t*(n/8)… = C (<=t*n)

따라서 transferrring cost = 2*t*n

분할 상환 분석(Amortized Analysis)

예시)
상황) 3벌의 옷을 산다.
알고 있는 정보) 옷 한벌당 최대 3만원

준비해야 하는 비용? 9만원(최악의 상황을 대비)

추가 정보) 옷 한개 구매당 평균 비용이 2만원
준비해야 하는 비용? 6만원

1. 평균 비용을 통해 각각의 옷마다 2만원이 들 것이라 예상하고 6만원을 준비한다.
2. 구매하는 옷이 2만원이라면 그대로 2만원을 사용한다.
3. 옷이 2만원보다 비싸다면, 부족한 금액을 2만원보다 싼 옷에서 보충한다.

분할상환 분석은 부족한 금액 혹은 남는 금액을 전체 연산에서 서로 보충해나가면서 평균 비용을 도출하는 구조를 기반으로 전체 연산의 평균 비용을 도출한다.

분할상환 분석

해시 테이블에서 데이터 삽입
k insert takes $\Theta(k)$ time
=> $\Theta(1)$ amortized / insert

“문자열 매칭”

Simple Algorithm:

$$\text{any}(s == t[i : i + \text{len(s)}] \text{ for } i \text{ in range}(\text{len}(t) - \text{len}(s)))$$
$$\text{— } O(|s|) \text{ time for each substring comparison}$$
$$\implies O(|s| \cdot (|t| - |s|)) \text{ time}$$
$$= O(|s| \cdot |t|) \qquad \text{potentially quadratic}$$

Figure 3: Illustration of Simple Algorithm for the String Matching Problem

Karp - Rabin 알고리즘:O(S+T)

Ex) 주어진 문자열 ababcdeabcdefg에서 abcdefg라는 패턴을 찾아내자.

찾는 문자열 S "abcdefg"의 해시값이 55일 때, 이제 T의 앞에서부터 7칸씩의 해시값을 가능한 자리마다 모두 계산해보며 55와 비교합니다.

ababcdeabcdefg
H(x) = 37

abcdefg
H(x) = 55

예를 들면 첫 번째 위치인 T[0:6]의 해시값은 37이라고 합시다. 이때는 해시값이 55와 다르므로 이 자리는 아니라고 패스합니다.

ababcdeabcdefg
H(x) = 55

abcdefg
H(x) = 55

도중에 T[2:8]의 해시값이 55라고 합시다. 이때는 충돌 발생 여부를 고려하여, 실제로 T[2:8]과 S를 단순 비교합니다.
비교해 봤더니 "abcdefg"가 아니라 "abcdeab"였습니다. 찾지 못한 겁니다.

ababcdeabcdefg
H(x) = 55

abcdefg
H(x) = 55

해시값도 같고 단순비교를 해서도 같다면 찾은 겁니다.

문자열을 해싱하는데 O(S)의 시간이 소요됨.
-> 각각의 위치에서 해싱을 하다보면 시간 복잡도는 마찬가지로 O(TS)가 된다.

해시값을 빨리 계산하기위한 적절한 해시 함수 필요!!
-> Rabin fingerprint(rolling hash function)

Rolling hash는 sliding window와 같이 문자열을 훑으면서 Hash 값을 구하는데 유용한 것으로, 주어진 문자열을 처음부터 끝까지 패턴과 비교하면서 해시값을 구하는데, 중복되는 부분의 해시값은 그대로 두고 업데이트되는 부분만 해시값을 계산해주어서 중복되는 연산을 줄여줍니다.

$$f(x) = m_0 + m_1 x + \underbrace{m_k}_{\text{k번째 문자의 아스키 코드}} x^k + \ldots + m_{n-1} x^{n-1}$$

$$H[i] = S[i] * x^{M-1} + S[i+1] * x^{M-2} + \cdots + S[i+M-1] * x^0$$

예를 들어 x=10을 이용해 패턴 abcdea를 변환하면
1*10^5 + 2*10^4 + 3*10^3 + 4*10^2 + 5*10^1 + 1*10^0 = 1234510이 된다.

i+1번째 문자열의 해시값은 이전 위치의 해시값을 이용해 O(1)에 계산할 수 있는 것이다!

$$\begin{aligned} H[i+1] &= S[i+1] * x^{M-1} + S[i+2] * x^{M-2} + \cdots S[i+M-1] * x^1 + S[i+M] * x^0 \\ &= x * (\underbrace{S[i+1] * x^{M-2} + S[i+2] * x^{M-3} + \cdots + S[i+M-1] * x^0}_{= H[i] - S[i] * x^{M-1} \quad \because H[i] = S[i] * x^{M-1} + S[i+1] * x^{M-2} + \cdots + S[i+M-1] * x^0}) + S[i+M] * x^0 \\ &= x * (H[i] - S[i] * x^{M-1}) + S[i+M] * x^0 \end{aligned}$$

새로운해시 = 2 x (기존해시 - 가장 앞문자해시) + 가장 뒷문자해시
예를 들어 ABCDEFG라는 전체 문자열에서 3글자의 부분을 매칭하는 중이라면 가장 처음 매칭되는지
테스트할 전체부분은 ABC임. 때문에 $A \times 2^2 + B \times 2^1 + C \times 2^0$과 같은 해시값이 만들어짐. 이 때
만약 ABC가 매치되지 않았다면 한칸 옆으로 움직여서 BCD의 해시값을 구해야하는데, 가장 먼저 전체
해시값에서 A의 해시값을 빼면 BC의 해시값($B \times 2^1 + C \times 2^0$)이 된다. 해당 해시값에 2를 곱하면
$B \times 2^2 + C \times 2^1$이 되고 여기에 D의 해시값인 $D \times 1$을 넣으면 전체 해시는
$B \times 2^2 + C \times 2^1 + D \times 2^0$이 되어 새로운 해시값이 잘 계산된다.

한 해시값에서 다음 해시값을 구하는데 O(1)의 시간이 들기 때문에 모든 부분 문자열의 해시값을 O(T)의 시간복잡도로 계산 가능.
해시 값이 같으면 문자별로 확인하기 때문에 각 하위 문자열을 확인하는데 O(S) 시간이 걸림.