기술면접 – 해시 테이블(Hash Table)

● 읽을 내용이 많으니 빨간색과, 1-2, 7페이지 그리고, 충돌, 충돌해결법 두 가지만 집중적으로 본다.

테이블(Table)

- 저장되는 **데이터가 key와 value가 하나의 쌍을 이루는 것**
- 테이블에 저장되는 모든 **데이터들을 이를 구분하는 키가 있어야 하고, 이 키는 데이터를 구분하는 기준이 되기 때문에 중복이 되어서는 안됨**

해싱(Hashing)

- 해싱(Hashing)은 **임의의 길이의 데이터를 고정된 더 짧은 길이의 값이나 키로 변환하는 것이다.**
- 그리고 해싱은 해시 테이블(Hash Table)과 해시 함수(Hash Function)로 구성된다.

해시 함수(Hash Function)

- 해시함수는 위에서 말한 해싱작업을 위한 함수로, **임의의 길이의 데이터를 고정된 길이의 데이터로 매핑(변환) 하는 함수**입니다.
- 해시 함수는 함수의 특성인 특정 값을 함수 f(x)에 대입했을 때 결과 값이 동일하게 나온다는 특성을 가지고 있습니다. 덕분에 임의의 데이터를 특정 데이터로 매핑 하는 것이 가능한 것입니다.
- **해시 함수에 들어가는 원 데이터(임의의 길이의 데이터) = key 값**
- **해시 함수를 지나쳐 나온 고정된 길이의 데이터 = 해시값**
- **데이터를 해시함수에 넣고 해시 값을 추출하는 과정 = 해싱**
- 해시함수는 **특정 값을 입력하면 항상 고정된 값을 전달해준다는 특징과, 반대로 출력된 결과 값을 토대로 입력값을 유추할 수 없는 특징,** 입력 값의 변화가 적어도 출력 값의 변화가 해시함수가 무엇이냐에 따라 크게 변할 수 있다는 특징이 존재합니다. 또한, 특정 키 값에 대해 항상 일정한 해시 값을 얻을 수 있으므로 데이터의 **인덱싱에 유리**합니다. **해시의 특성을 이용한 데이터 저장 방식인 해시 테이블을 이용하면 효율적인 데이터 처리 구조를 생성할 수 있습니다.**
- 이러한 해시 함수는 **해시 맵과 같은 빠른 데이터 처리 방식**과 암호화에 사용할 수 있습니다.
- 해시함수 종류는 MD, SHA 알고리즘 등이 있다.

해시 테이블(Hash Table)

- 위에서 말한 것을 합친 것으로 테이블에 해시개념을 추가해 테이블의 Key, Value가 쌍으로 이룰 때, 이 **Key를 해쉬 함수로 해싱(Hashing)해서 hash code로 변환하고, 코드가 고유의 index로 작용하여 그 index 위치에 Value를 접근, 저장, 검색하는 테이블 개념**이다.
- index를 이용하기 때문에 어떠한 값을 탐색, 삽입, 삭제에 평균적으로 O(1)의 시간복잡도로 수행이 가능합니다.

해시 테이블 시간복잡도, 효율성

Hash table

Type	Unordered associative array
Invented	1953

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

삽입(Insert), 삭제(Delection), 검색(Search)

- 저장(Insert) 단계의 시간복잡도는 O(1)이다. 키(key)는 고유하며 해시함수(hash function)의 결과로 나온 해시(hash)와 값(value)를 저장소에 넣으면 되기 때문이다. 이 때, 해시함수의 시간복잡도는 함께 고려하지 않는다.
- 삭제(Delete) 단계의 시간복잡도는 O(1)이다. 키(key)는 고유하며 해시함수(hash function)의 결과로 나온 해시(hash)에 매칭되는 값(value)를 삭제하면 되기 때문이다.
- 검색(Search) 단계의 시간보잡도는 O(1)이다. 키(key)로 값(value)를 찾아내는 과정은 위 두 과정과 비슷한다. 1) 키로 hash를 구한다. 2) hash로 값(value)를 찾는다.
- 하지만, 이 세 가지 경우 최악의 경우 O(n)이 될 수 있다. 해시 충돌로 인해 모든 bucket의 value들을 찾아봐야 하는 경우도 있기 때문이다. 이에 관해서도 해시충돌과 함께 후술하겠다.

2) 삭제(Delection)

- 삭제 단계의 시간복잡도는 O(1)이다. 키(key)는 고유하며 해시함수(hash function)의 결과로 나온 해시(hash)에 매칭되는 값(value)를 삭제하면 되기 때문이다. 이 때, 해시함수의 시간복잡도는 함께 고려하지 않는다.
- 하지만, 최악의 경우 O(n)이 될 수 있다. 해시 충돌로 인해 모든 bucket의 value들을 찾아봐야 하는 경우도 있기 때문이다. 이에 관해서도 해시충돌과 함께 후술하겠다.

해시 충돌(Hash Collision)

- **만약 서로 다른 두 개의 키가, 해쉬 함수를 거쳐, 그 결과가 같다면(동일한 인덱스) 이를 충돌이라고 한다**
- 충돌이 많이 일어난다면, O(1)으로 값을 탐색할 수 있는 장점이 없어지고 성능이 악화된다. Worst Case에서는 O(n)으로 탐색이 수행될 것이다.
- 그래서 hash function을 설계할 때, collision을 최소화할 수 있도록 설계해야 합니다.

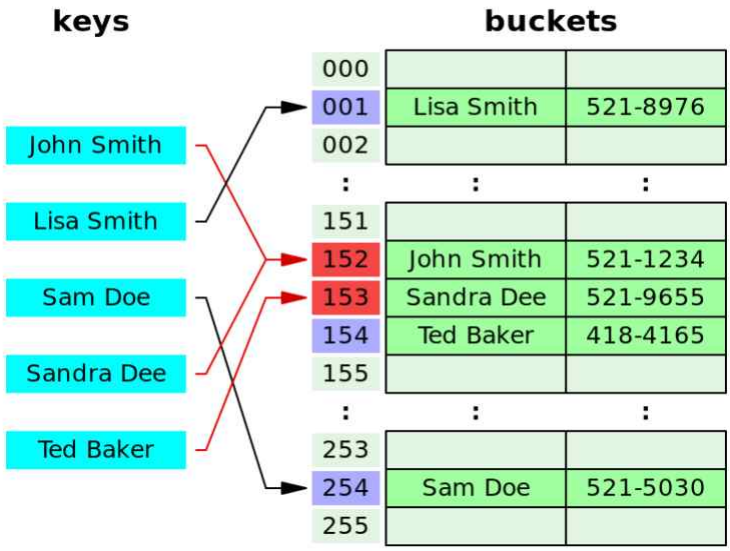
해시 충돌해결법

1. 개방주소법(Open Addressing)

- **해시 충돌이 발생하면, 비어 있는 다른 칸에 저장하는 방식**
- 바로 옆이 비었다면 거기에 넣을 수도 있고, 따로 함수를 만들어 다른 곳에 넣을 수도 있다
- C#에서는 Open Addressing 방식을 이용한다.

- 개방 주소법은 대표적으로 3가지가 있다.
- 선형 탐색(Linear Probing): 충돌 시 다음 버켓, 혹은 몇 개를 건너뛰어 데이터를 삽입한다.
- 제곱 탐색(Quadratic Probing): 충돌 시 제곱만큼 건너뛴 버켓에 데이터를 삽입(1,4,9,16..)
- 이중 해시(Double Hashing): 충돌 시 다른 해시함수를 한 번 더 적용한 결과를 이용함.

다음은 개방주소법의 선형탐색 (Linear Probing) 방식이다.



152번 충돌에 대해서 그 다음으로 비어있는 주소인 153번에 저장, 153번에 대한 충돌을 154번에 저장을 볼 수 있다. 이러한 원리로 탐색, 삽입, 삭제가 이루어지는데 다음과 같이 동작한다.

삽입: 계산한 해시 값에 대한 인덱스가 이미 차있는 경우 다음 인덱스로 이동하면서 비어있는 곳에 저장한다. 이렇게 비어있는 자리를 탐색하는 것을 탐사(Probing)라고 한다.
탐색: 계산한 해시 값에 대한 인덱스부터 검사하며 탐사를 해나가는데 이 때 "삭제" 표시가 있는 부분은 지나간다.
삭제: 탐색을 통해 해당 값을 찾고 삭제한 뒤 "삭제" 표시를 한다.

Open Addressing의 장단점

장점 :
- 또 다른 저장공간 없이 해시테이블 내에서 데이터 저장 및 처리가 가능하다.
- 또 다른 저장공간에서의 추가적인 작업이 없다.

단점 :
- 해시 함수(Hash Function)의 성능에 전체 해시테이블의 성능이 좌지우지된다.
- 데이터의 길이가 늘어나면 그에 해당하는 저장소를 마련해 두어야 한다.

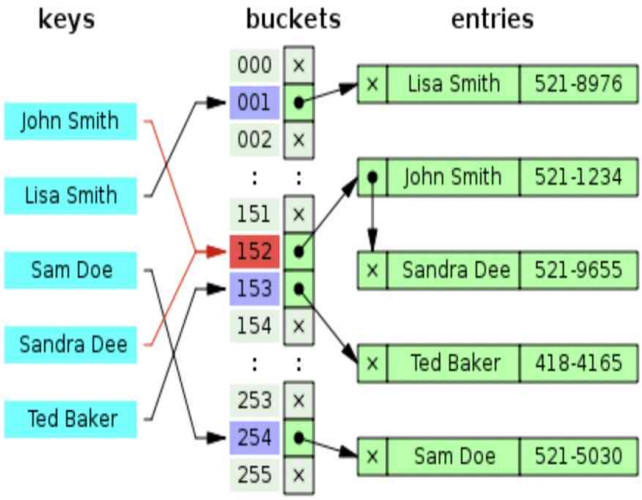
Open Addressing 시간 복잡도 (Big-O)

- 삽입, 삭제, 검색 모두 대상이 되는 Hash를 찾아가는 과정에 따라 시간복잡도가 계산이 된다. 해시함수를 통해 얻은 Hash가 비어있지 않으면 다음 버킷을 찾아가야 한다. 이 찾아가는 횟수가 많아지면 많아질수록 시간복잡도가 증가한다. 최상의 경우 O(1) ~ 최악의 경우 (O(n)).
- 따라서 Open Addressing에서는 비어있는 공간을 확보하는 것(= 저장소가 어느 정도 채워졌을 때 저장소의 사이즈를 늘려주는 것)이 필요하다.
- 최악의 경우 저장소를 모두 살펴보아야 하는 경우가 생길 수 있다.(O(n))

2. 체이닝(Seperate Chaining)
- 버켓 내에 연결리스트(Linked List)를 할당하여, 버켓에 데이터를 삽입하다가 **해시 충돌이 발생하면 연결리스트로 데이터들을 연결하는 방식이다.**
- Java에서는 Separate Chaining 방식을 사용하여 HashMap 을 구현하고 있다. Separate Chaining 방식으로는 두 가지 구현 방식이 존재한다.

- 연결 리스트를 사용하는 방식(Linked List) : 각각의 버킷(bucket)들을 연결리스트(Linked List)로 만들어 Collision 이 발생하면 해당 bucket 의 list 에 추가하는 방식이다. 연결 리스트의 특징을 그대로 이어받아 삭제 또는 삽입이 간단하다. 하지만 단점도 그대로 물려받아 작은 데이터들을 저장할 때 연결 리스트 자체의 오버헤드가 부담이 된다. 또 다른 특징으로는, 버킷을 계속해서 사용하는 Open Address 방식에 비해 테이블의 확장을 늦출 수 있다.
- Tree 를 사용하는 방식 (Red-Black Tree) : 연결 리스트 대신 트리를 사용하는 방식이다. 연결 리스트를 사용할 것인가와 트리를 사용할 것인가에 대한 기준은 하나의 해시 버킷에 할당된 key-value 쌍의 개수이다. 데이터의 개수가 적다면 링크드 리스트를 사용하는 것이 맞다. 트리는 기본적으로 메모리 사용량이 많기 때문이다. 데이터 개수가 적을 때 Worst Case 를 살펴보면 트리와 링크드 리스트의 성능 상 차이가 거의 없다. 따라서 메모리 측면을 봤을 때 데이터 개수가 적을 때는 링크드 리스트를 사용한다.

다음은 체이닝의 연결리스트 (Linked List) 방식이다.



152번 충돌에 대해서 연결리스트를 이용해 원래 있던 152번에 추가로 붙여주는 작업을 수행한다.

Chaining 장단점

장점 :
1) 한정된 저장소(Bucket)을 효율적으로 사용할 수 있다.
2) 해시 함수(Hash Function)을 선택하는 중요성이 상대적으로 적다.
3) 상대적으로 적은 메모리를 사용한다. 미리 공간을 잡아 놓을 필요가 없다.

단점 :
1) 한 Hash에 자료들이 계속 연결된다면(쏠림 현상) 검색 효율을 낮출 수 있다.
2) 외부 저장 공간을 사용한다.
3) 외부 저장 공간 작업을 추가로 해야 한다.

Chaining 시간 복잡도 (Big-O)

Insertion : 충돌이 일어났을 때, 해당 해시(Hash)가 가진 연결리스트의 Head에 자료를 저장할 경우, O(1)의 시간복잡도를 가진다. 해당 해시(Hash)를 산출하고 저장하면서 기존 값(value)를 연결하는 행위만 하면 되기 때문이다.
반면 Tail에 자료를 저장할 경우, O(α)의 시간 복잡도를 가진다. 해당 해시(Hash)를 저장할 때 모든 연결리스트를 지나서 Tail에 접근해야 하기 때문이다. 최악의 경우, O(n)의 시간 복잡도를 가진다. 한 개의 해시(Hash)에 모든 자료가 연결되어 있을 수 있기 때문이다.

Deletion & Search : 삭제와 검색은 시간 복잡도 측면에서 비슷한 개념을 공유한다. 산출된 Hash의 연결리스트를 차례로 살펴보아야 하므로 O(α)의 시간 복잡도를 가진다. 최악의 경우 O(n)의 시간복잡도를 가진다. 한 개의 해시(Hash)에 모든 자료가 연결되어 있을 수 있기 때문이다. 이 경우 모든 자료를 다 살펴보아야 한다.

개방 주소법(Open Addressing) vs 체이닝(Separate Chaining)

- 일반적으로 Open Addressing 은 Separate Chaining 보다 느리다. Open Addressing 의 경우 해시 버킷을 채운 밀도가 높아질수록 Worst Case 발생 빈도가 더 높아지기 때문이다. 반면 Separate Chaining 방식의 경우 해시 충돌이 잘 발생하지 않도록 보조 해시 함수를 통해 조정할 수 있다면 Worst Case 에 가까워지는 빈도를 줄일 수 있다
- 두 방식 모두 worst case에서 O(N)의 성능을 가진다. 하지만 Open Addressing 방식은 연속된 공간에 데이터를 저장하기 때문에 상대적으로 캐시의 hit rate가 높아 데이터의 양이 적을 때는 더 효율적이다. 하지만 separate chaining 방식에 비해, open addressing 은 버킷을 계속해서 사용하기 때문에 데이터가 많은 경우 hash table의 확장이 잦고 캐싱 효율이 떨어진다. 데이터가 많은 경우 separate chaining이 더 효율적이다.

Separate Chaining	Open Addressing
hash table 내부와 외부에 모두 key가 저장됨	모든 key는 오직 hash table에만 저장
hash table에 추가적인 메모리 공간을 사용하므로, key의 개수가 hash table의 크기를 넘을 수 있음	추가적인 메모리 공간을 사용하지 않기 때문에, hash table의 크기를 넘을 수 없음
삭제가 쉬움	삭제가 어려움
상대적으로 hash function과 load factor에 덜 민감함	구조상 hash function과 load factor에 민감함
주로 얼마나 많은 데이터가 얼마나 자주 삽입/삭제될지 예측할 수 없을 때 사용	특정 key들의 주기와 개수를 알고 있을 때 주로 사용
cache 효율이 비교적 좋지 않음	cache 효율이 좋음
전혀 사용되지 않는 bucket이 존재할 수 있는 점에서 공간의 낭비	직접적으로 해당 bucket에 mapping되지 않아도 그 bucket을 사용하게 됨
추가적인 메모리 공간이 필요	추가적인 메모리 공간을 사용하지 않음

Open Addressing에서의 deletion

- 단순히 원하는 데이터를 삭제하기만 할 경우, 선형 탐사를 위한 부분이 유실되어 open addressing의 선형 탐사가 작동하지 않을 수 있다. 이를 방지하기 위해, 삭제된 bucket은 deleted 라는 임의의 값을 저장해두어야 하는 추가적인 연산이 필요해진다. 단순히 linked list의 한 Node를 삭제하기만 하면 되는 separate chaining보다 삭제 과정이 더 복잡하다.

해시 버킷 동적 확장(Resize)

- 해시 버킷의 개수가 적다면 메모리 사용을 아낄 수 있지만 해시 충돌로 인해 성능 상 손실이 발생한다. 그래서 HashMap 은 key-value 쌍 데이터 개수가 일정 개수 이상이 되면 해시 버킷의 개수를 두 배로 늘린다.
- 이렇게 늘리면 해시 충돌로 인한 성능 손실 문제를 어느 정도 해결할 수 있다. 해시 버킷 크기를 두 배로 확장하는 임계점은 현재 데이터 개수가 해시 버킷의 개수의 75%가 될 때이다. 0.75라는 숫자는 load factor 라고 불린다.
- Load Factor는 hash table의 size 대비 저장된 key의 개수를 의미한다. Load Factor의 값이 클 수록 collision이 발생할 확률이 높아지고, Load Factor가 1이 되는 순간부터는 무조건 collision이 발생한다. 일반적으로는 Load Factor를 0.75이하로 유지하는 것이 이상적이다. 그래서 Load Factor가 일정 수준 이상으로 높아지면, Resize 과정을 통해 hash table size를 2배로 늘린다. 일반적으로 그 시점이 0.75가 되는 시점이다.

해시 테이블의 장단점

[장점]

- **순서가 있는 배열에는 어울리지 않는다** : 상하관계가 있거나, 순서가 중요한 데이터의 경우 Hash Table은 어울리지 않다. 순서와 상관없이 key만을 가지고 hash를 찾아 저장하기 때문이다.
- **공간 효율성이 떨어진다** : 데이터가 저장되기 전에 미리 저장공간을 확보해 놓아야 한다. 공간이 부족하거나 아예 채워지지 않은 경우가 생길 가능성이 있다.
- **Hash Function의 의존도가 높다** : 평균 데이터 처리의 시간복잡도는 O(1)이지만, 이는 해시 함수의 연산을 고려하지 않는 결과이다. 해시함수가 매우 복잡하다면 해시테이블의 모든 연산의 시간 효율성은 증가할 것이다.

[단점]

- **해시 테이블은 키를 가지고 빠르게 value에 접근하고 조작할 수 있는 장점이 있어서 많이 사용된다.** 예를 들어 주소록 저장형태의 경우 이름 — 전화번호의 매칭을 이용하여 데이터를 처리한다.