● Hash Table은 어떤 자료 구조인가요?

▶ Hash Table은 효율적인 탐색(즉, 빠든 탐색)을 위한 자료구조로써 키(key)와 값(value)을 묶어서 하나의 데이터로 저장하는 특징을 갖습니다. 배열과 링크드 리스트의 조합으로 되어 있고, 해싱(hashing)을 사용하기 때문에 많은 양의 데이터를 검색하는데 있어서 뛰어난 성능을 보입니다. hash function(해시 함수) h에 저장할 데이터의 key값을 입력하면 해시 코드(hash code), 혹은 해시값을 얻게 되고, 이것에 해당되는 인덱스을 가진 배열의 요소에 연결돼 있는 링크드 리스트에 키와 값을 저장하게 됩니다. 저장, 삭제, 검색의 시간복잡도는 모두 O(1)입니다.

▶ HashTable의 새로운 버전은 HashMap입니다.

▶ 헤시테이블/헤시멥에서 원하는 데이터를 검색하는 과정은 다음과 같습니다. 일단 검색하고자 하는 값의 키로 해시 함수를 호출하여, 계산결과인 해시코드로 해당 값의 인덱스에 연결된 링크드 리스트 중에 검색한 키와 일치하는 데이터를 찾습니다.

● 좋은 hash funtion의 조건

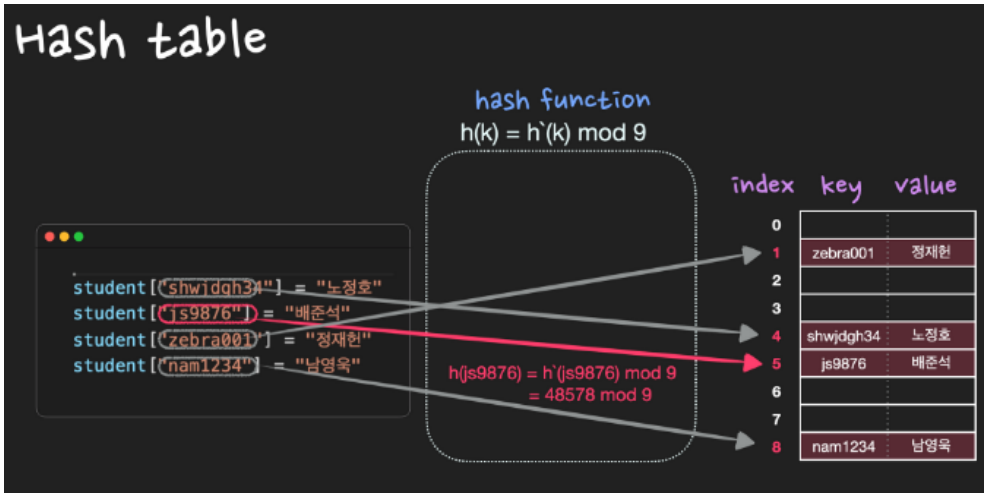
▶ 링크드 리스트의 크기가 커질수록 검색속도가 떨어지는 반면(O(n)) 배열은 크기가 커져도 원하는 요소가 몇 번째에 있는지만 알면 빠르게 원하는 값을 찾을 수 있기 때문에(O(1)), 하나의 링크드 리스트에 최소한의 데이터를 저장해 주는 것이 좋고, 그럴려면 저장될 데이터의 크기를 고려해서 HashTable의 크기를 적절하게 지정해주고 좋은 해시함수를 설정해 주어야 합니다. 좋은 해시 함수의 hash function의 핵심적인 조건은 연산 속도가 빨라야 하고, 해시값이 고르게 분포되게 하여 서로 다든 키에 대해서 중복된 해스코드를 반환하는 것을 최소화하는 것입니다. 그래야 충돌을 최소화하여 hashTable에서에서 빠든 검색시간을 얻을 수 있습니다.

● Direct-address Table (직접 주소화 테이블)

▶ Direct-address Table(직접 주소화 테이블)이란, key 값으로 k를 갖는 원소는 index k에 저장하는 방식입니다. 만약 키가 출석 번호이고 값이 이름일 때, 3번 홍길동은 배열의 인덱스3에 저장하는 식입니다. 하지만 직접 주소화 방법으로 통해 key-value 쌍의 데이터를 저장하고자 하면 많은 문제가 발생합니다. 예를 들면 key의 숫자가 크면, 불필요한 공간 낭비가 발생할 수 있습니다. 또한 만약 key가 숫자가 아니라면 이에 상응하는 인덱스 넘버를 알지 못하게 되므로, key가 다양한 자료형을 담을 수 없게 됩니다. (key, value) 데이터 쌍을 저장하기 위한 방법으로 직접 주소화 방법이 잘 맞지않습니다. 때문에 해시 함수를 사용하는 hash table을 사용합니다.

● hash table

▶ hash table은 hash function h를 이용해서 (key, value) 데이터를 index: h(k)에 저장합니다. 즉, 해시 값에 해당되는 인덱스에 저장을 하는 것입니다. 이 때, “키 k값을 갖는 원소가 위치 h(k)에 hash된다,” 또는 “h(k)는 키 k의 해시값이다”라고 표현합니다. key는 무조건 존재해야 하며, 중복되는 key가 있어서는 안되는 반면, 값은 데이터의 중복이 허용됩니다. 한편, hash table을 구성하고 있는, (key, value) 데이터를 저장할 수 있는 각각의 공간을 slot 또는 bucket이라고 합니다.



● Collision

▶ collision이란 서로 다든 key의 해시값이 똑같을 때를 말합니다. 즉, 중복되는 key는 없지만 해시값은 중복될 수 있는데 이 때 collision이 발생했다고 합니다. 따라서 collision이 최대한 적게 나도록 hash function을 잘 설계해야하고, 어쩔 수 없이 collision이 발생하는 경우 separate chaining 또는 open addressing등의 방법을 사용하여 해결합니다.

▶ 좋은 해시 함수 hash function의 조건은 각 상황마다 달라질 수 있지만, 대략적인 조건은 연산 속도가 빨라야 하고(계산이 빨리 되어야 하고), 해시값이 최대한 겹치지 않아야 합니다.

→ separate chaining: 추가적인 메모리를 사용해 동일한 버킷에 값이 있으면 링크드 리스트로 해당 value를 뒤에 저장하는 방법입니다. 충돌된 키-값 쌍의 데이터가 8개 미만이면 링크드 리스트로 연결하고, 8개 이상이면 레드 블랙 트리로 연결하여 해결합니다(6개가 되면 다시 링크드 리스트로 변환). 링크드 리스트는 충돌하는 데이터가 많아질수록 탐색 시간이 O(n)으로 늘어나기 때문에, 효율적이게 탐색하기 위해 레드 블랙 트리로(O(logn)) 변환하는 것입니다.

· 보조 해시 함수: 보조 해시 함수(supplement hash function)의 목적은 key의 해시 값을 변형하여 해시 충돌 가능성을 줄이는 것입니다. Separate Chaining 방식을 사용할 때 함께 사용되며 보조 해시 함수로 Worst Case 에 가까워지는 경우를 줄일 수 있습니다.

→ open addressing: 원래라면 해시함수로 얻은 해시값에 따라서 데이터와 키값을 저장하지만 동일한 주소에 다든 데이터가 있을 경우 다음 인덱스로 이동하면서 비어 있는 주소에 저장하는 기법입니다. 이러한 원리로 탐색, 삽입, 삭제가 이두어지는데 다음과 같이 동작합니다.

- 삽입: 계산한 해시 값에 대한 인덱스가 이미 차있는 경우 다음 인덱스로 이동하면서 비어있는 곳에 저장합니다. 이렇게 비어있는 자리를 탐색하는 것을 탐사(Probing)라고 합니다.

- 탐색: 계산한 해시 값에 대한 인덱스부터 검사하며 탐사를 해나가는데, 이 때 “삭제” 표시가 있는 부분은 지나갑니다.

- 삭제: 탐색을 통해 해당 값을 찾고 삭제한 뒤 “삭제” 표시를 합니다.

→ 일반적으로 Open Addressing 은 Separate Chaining 보다 느립니다. Open Addressing 의 경우 해시 버킷을 채운 밀도가 높아질수록 Worst Case 발생 빈도가 더 높아지기 때문입니다. 반면 Separate Chaining 방식의 경우 해시 충돌이 잘 발생하지 않도록 보조 해시 함수를 통해 조정할 수 있다면 Worst Case 에 가까워 지는 빈도를 줄일 수 있습니다. Java 에서는 Separate Chaining 방식을 사용하여 HashMap 을 구현하고 있습니다

→ 해시 버킷 동적 확장(Resize): 해시 버킷의 개수가 적다면 메모리 사용을 아낄 수 있지만 해시 충돌로 인해 성능 상 손실이 발생합니다. 그래서 HashMap 은 key-value 쌍 데이터 개수가 일정 개수 이상이 되면 해시 버킷의 개수를 두 배로 늘립니다. 이렇게 늘리면 해시 충돌로 인한 성능 손실 문제를 어느 정도 해결할 수 있습니다. 해시 버킷 크기를 두 배로 확장하는 임계점은 현재 데이터 개수가 해시 버킷의 개수의 75%가 될 때입니다. 0.75라는 숫자는 load factor 라고 불립니다 (https://cupeanimus.tistory.com/91)

● 시간복잡도와 공간효율

▶ 시간복잡도는 저장, 삭제, 검색 모두 기본적으로 O(1)이지만, collision으로 인하여 최악의 경우 O(n)이 될 수 있습니다.

▶ 공간효율성은 떨어집니다. 데이터가 저장되기 전에 미리 저장공간(slot, bucket)을 확보해야 하기 때문입니다. 따라서 저장공간이 부족하거나 채워지지 않은 부분이 많은 경우가 생길 수 있습니다.

※ Linked list, Array, Tree

Resources: inflearn 개발남 노씨, cupeanimus