해쉬 테이블의 이해와 구현 (Hashtable)

조대협 (http://bcho.tistory.com)

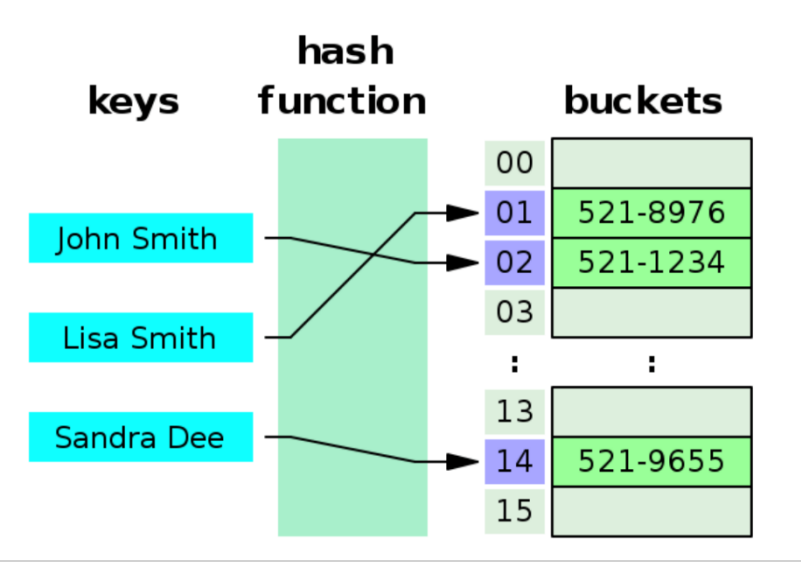
기본적인 해쉬 테이블에 대한 이해

해쉬 테이블은 Key 에 Value 를 저장하는 데이타 구조로, value := get(key)에 대한 기능이 매우매우 빠르게 작동한다. 개발자라면 자주 쓰는 데이타 구조지만, 실제로 어떻게 작동하는지에 대해서 정확하게 알고 있지는 모르는 경우가 많다. 이 글에서는 해쉬 테이블에 대한 기본적인 구조와, 구현 방식에 대해서 설명 하도록 한다.

해쉬 테이블의 기본적인 개념은 다음과 같다.

이름을 키로, 전화 번호를 저장하는 해쉬 테이블 구조를 만든다고 하자. 전체 데이타 양을 16 명이라고 가정하면

John Smith 의 데이타를 저장할때, index = hash_function("John Smith") % 16 를 통해서 index 값을 구해내고, array[16] = "John Smith 의 전화 번호 521-8976"을 저장한다.



(출처 :https://en.wikipedia.org/wiki/Hash_table)

이런 형식으로 데이타를 저장하면, key 에 대한 데이타를 찾을때, hash_function 을 한번만 수행하면 array 내에 저장된 index 위치를 찾아낼 수 있기 때문에, 데이타의 저장과 삭제가 매우매우 빠르다.

충돌 처리 방식에 따른 알고리즘 분류

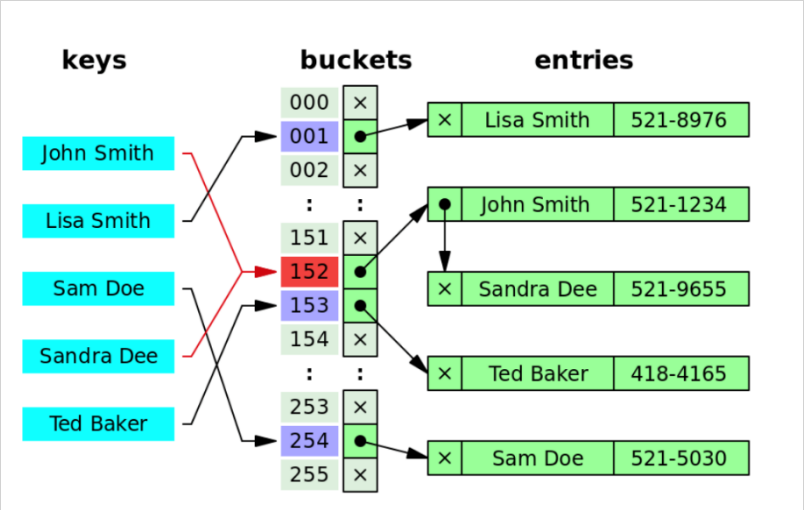
그런데, 이 해쉬 테이블 문제는 근본적인 문제가 따르는데, hash_function(key) / size_of_array 의 값이 중복이 될 수 가 있다.

예를 들어 저장하고자 하는 key 가 정수라고 하고, hash_function 이 key%10 이라고 하자. 그리고 size_of_array 가 10 일때, key 1,11,21,31 은 같은 index 값을 가지게 된다. 이를 collision (충돌)이라고 하는데, 이 충돌을 해결하는 방법에 따라서 여러가지 구현 방식이 존재한다.

1. Separate chaining 방식

JDK 내부에서도 사용하고 있는 충돌 처리 방식인데, Linked List 를 이용하는 방식이다.

각 index 에 데이타를 저장하는 Linked list 에 대한 포인터를 가지는 방식이다.



(출처 :https://en.wikipedia.org/wiki/Hash_table)

만약에 동일 index 로 인해서 충돌이 발생하면 그 index 가 가리키고 있는 Linked list 에 노드를 추가하여 값을 추가한다. 이렇게 함으로써 충돌이 발생하더라도 데이타를 삽입하는 데 문제가 없다.

데이타를 추출을 하고자할때는 key 에 대한 index 를 구한후, index 가 가리키고 있는 Linked list 를 선형 검색하여, 해당 key 에 대한 데이타가 있는지를 검색하여 리턴하면 된다.

삭제 처리

key 를 삭제하는 것 역시 간단한데, key 에 대한 index 가 가리키고 있는 linked list 에서, 그 노드를 삭제하면 된다.

Separate changing 방식은 Linked List 구조를 사용하고 있기 때문에, 추가할 수 있는 데이타 수의 제약이 작다.

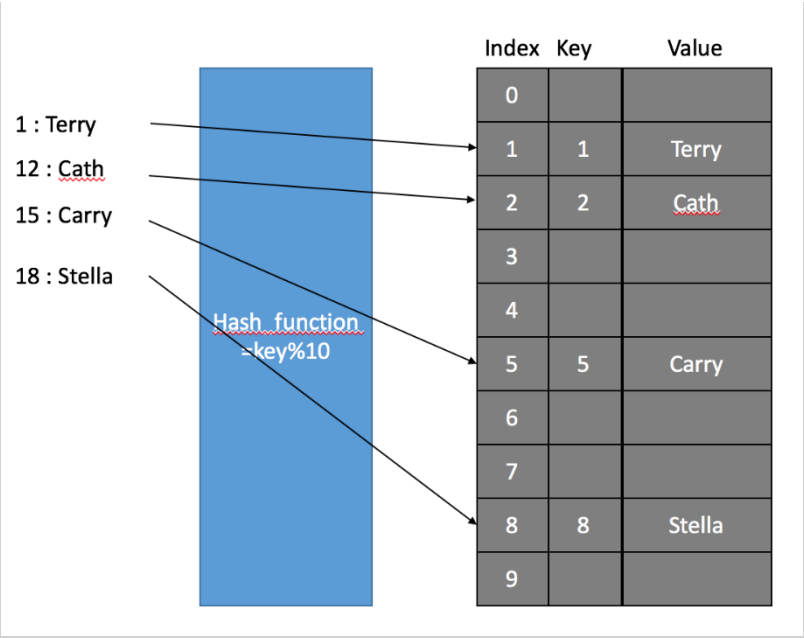
참고 : 동일 index 에 대해서 데이타를 저장하는 자료 구조는 Linked List 뿐 아니라, Tree 를 이용하여 저장함으로써, select 의 성능을 높일 수 있다. 실제로, JDK 1.8 의 경우에는 index 에 노드가 8 개 이하일 경우에는 Linked List 를 사용하며, 8 개 이상으로 늘어날때는 Tree 구조로 데이타 저장 구조를 바꾸도록 되어 있다.

2. Open addressing 방식

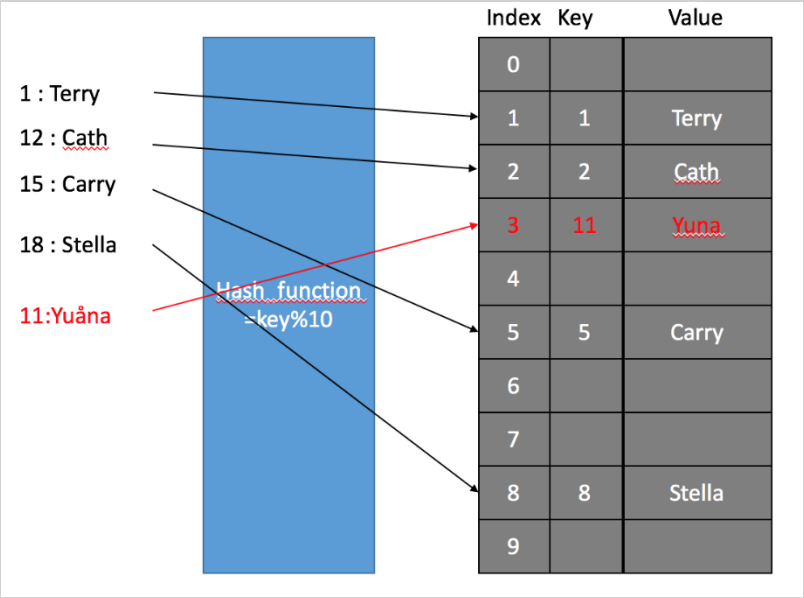
Open addressing 방식은 index 에 대한 충돌 처리에 대해서 Linked List 와 같은 추가적인 메모리 공간을 사용하지 않고, hash table array 의 빈공간을 사용하는 방법으로, Separate chaining 방식에 비해서 메모리를 덜 사용한다. Open addressing 방식도 여러가지 구현 방식이 있는데, 가장 간단한 Linear probing 방식을 살펴보자

Linear Probing

Linear Probing 방식은 index 에 대해서 충돌이 발생했을 때, index 뒤에 있는 버킷중에 빈 버킷을 찾아서 데이타를 넣는 방식이다. 그림에서 key % 10 해쉬 함수를 사용하는 Hashtable 이 있을때, 그림에서는 충돌이 발생하지 않았다.



아래 그림을 보자, 그런데, 여기에 11 을 키로 하는 데이타를 그림과 같이 넣으면 1 이 키인 데이타와 충돌이 발생한다. (이미 index 가 1 인 버킷에는 데이타가 들어가 있다.) Linear Probing 에서는 아래 그림과 같이 충돌이 발생한 index (1) 뒤의 버킷에 빈 버킷이 있는지를 검색한다. 2 번 버킷은 이미 index 가 2 인 값이 들어가 있고, 3 번 버킷이 비어있기 3 번에 값을 넣으면 된다.



검색을 할때, key 11 에 대해서 검색을 하면, index 가 1 이기 때문에, array[1]에서 검색을 하는데, key 가 일치하지 않기 때문에 뒤의 index 를 검색해서 같은 키가 나오거나 또는 Key 가 없을때까지 검색을 진행한다.

삭제 처리

이러한 Open Addressing 의 단점은 삭제가 어렵다는 것인데, 삭제를 했을 경우 충돌에 의해서 뒤로 저장된 데이타는 검색이 안될 수 있다. 아래에서 좌측 그림을 보자, 2 번 index 를 삭제했을때, key 11 에 대해서 검색하면, index 가 1 이기 때문에 1 부터 검색을 시작하지만 앞에서 2 번 index 가 삭제되었기 때문에, 2 번 index 까지만 검색이 진행되고 정작 데이타가 들어있는 3 번 index 까지 검색이 진행되지 않는다.

그래서 이런 문제를 방지하기 위해서 우측과 같이 데이타를 삭제한 후에, Dummy node 를 삽입한다. 이 Dummy node 는 실제 값을 가지지 않지만, 검색할때 다음 Index 까지 검색을 연결해주는 역할을 한다.

Index	Key	Value
0		
1	1	Terry
2	2	Cathy
3	11	Yuna
4		
5	5	Carry
6		
7		
8	8	Stella
9		

Index	Key	Value
0		
1	1	Terry
2	11	Yuna
3		
4		
5	5	Carry
6		
7		
8	8	Stella
9		

Linear probing 에 대한 샘플 코드는

http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html

http://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm

를 참고하기 바란다.

Dummy Node 를 이용해서 삭제를 할때, 삭제가 빈번하게 발생을 하면, 실제 데이타가 없더라도 검색을 할때, Dummy Node 에 의해서, 많은 Bucket 을 연속적으로 검색을 해야 하기 때문에, 이 Dummy Node 의 개수가 일정 수를 넘었을때는 새로운 array 를 만들어서, 다시 hash 를 리빌딩 함으로써, Dummy Node 를 주기적으로 없애 줘야 성능을 유지할 수 있다.

Resizing

Open addressing 의 경우, 고정 크기 배열을 사용하기 때문에 데이타를 더 넣기 위해서는 배열을 확장해야 한다. 또한 Separate changing 에 경우에도 버킷이 일정 수준으로 차 버리면 각 버킷에 연결되어 있는 List 의 길이가 늘어나기 때문에, 검색 성능이 떨어지기 때문에 버킷의 개수를 늘려줘야 한다. 이를 Resizing 이라고 하는데, Resizing 은 별다른 기법이 없다. 더 큰 버킷을 가지는 array 를 새로 만든 다음에, 다시 새로운 array 에 hash 를 다시 계산해서 복사해줘야 한다.

해쉬 함수

해쉬 테이블 데이타 구조에서 중요한 것중 하나가 해쉬 함수(hash function)인데, 좋은 해쉬 함수란, 데이타를 되도록이면 고르게 분포하여, 충돌을 최소화할 수 있는 함수이다. 수학적으로 증명된 여러가지 함수가 있겠지만, 간단하게 사용할 수 있는 함수를 하나 소개하고자 한다. 배경에 대해서는 http://d2.naver.com/helloworld/831311 에 잘 설명되어 있으니 참고하기 바란다. (필요하면 그냥 외워서 쓰자)

```
String key

char[ ] ch = key.toChar();

int hash = 0;

for(int i=0;i<key.length;i++)

 hash = hash*31 + char[i]
```

Cache 를 이용한 성능 향상

해쉬테이블에 대한 성능 향상 방법은 여러가지가 있지만, 눈에 띄는 쉬운 방식이 하나 있어서 마지막으로 간략하게 짚고 넘어가자. 해쉬테이블의 get(key)과 put(key)에 간단하게, 캐쉬 로직을 추가하게 되면, 자주 hit 하는 데이타에 대해서 바로 데이타를 찾게 함으로써, 성능을 간단하게 향상 시킬 수 있다.

다음은 실제로 간단한 HashTable 클래스를 구현하는 방법에 대해서 알아보도록 하겠다.

출처: https://bcho.tistory.com/1072 [조대협의 블로그]

함께 읽어 보세요 : https://mangkyu.tistory.com/102