HashMap

`Map`, `HashMap`, `TreeMap`이 구현되어 있는 JAVA를 예시로 설명함.

HashMap의 구조와 특징

Map과는 뭐가 다른 걸까?

엄연히 말해, HashMap은 Map의 일종이다.

우리가 Map이라고 하면 보통 Key, Value를 이용하는 방식만 생각한다.

다만, 그 자료를 관리하고 이용하는 방법은 여러 가지가 있다.

clear()
Removes all of the mappings from this map (optional operation).

containsKey(Object key)
Returns `true` if this map contains a mapping for the specified key.

containsValue(Object value)
Returns `true` if this map maps one or more keys to the specified value.

entrySet()
Returns a **Set** view of the mappings contained in this map.

equals(Object o)
Compares the specified object with this map for equality.

get(Object key)
Returns the value to which the specified key is mapped, or `null` if this n

hashCode()
Returns the hash code value for this map.

isEmpty()
Returns `true` if this map contains no key-value mappings.

keySet()
Returns a **Set** view of the keys contained in this map.

put(K key, V value)
Associates the specified value with the specified key in this map (option

putAll(Map<? extends K,? extends V> m)
Copies all of the mappings from the specified map to this map (optional

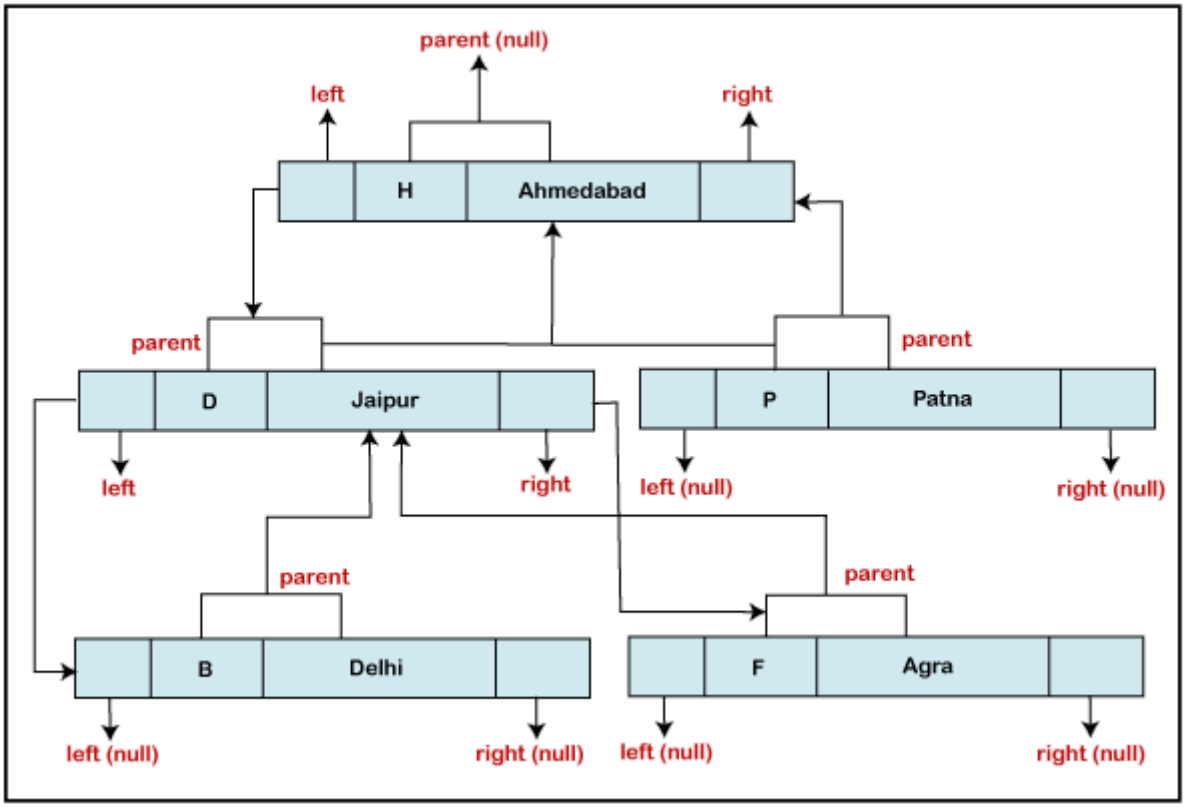
remove(Object key)
Removes the mapping for a key from this map if it is present (optional o

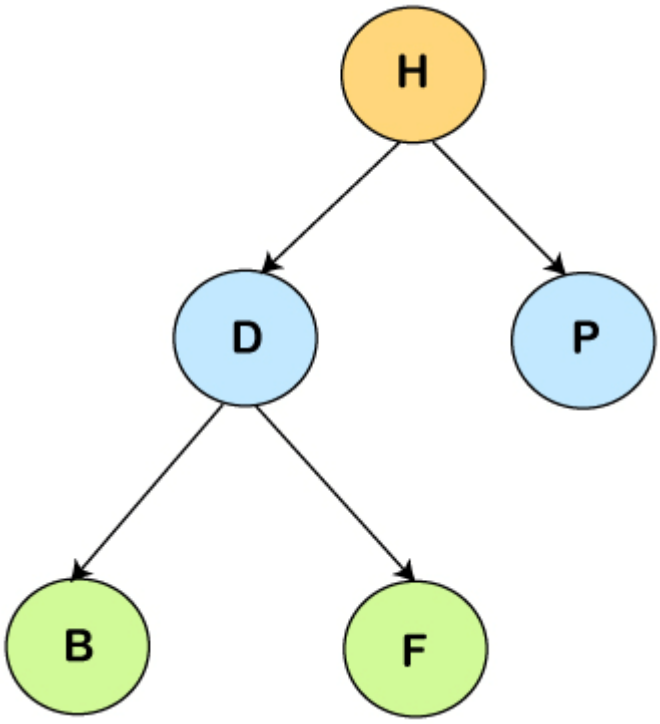
size()
Returns the number of key-value mappings in this map.

values()
Returns a **Collection** view of the values contained in this map.

Interface `Map` in `JAVA`

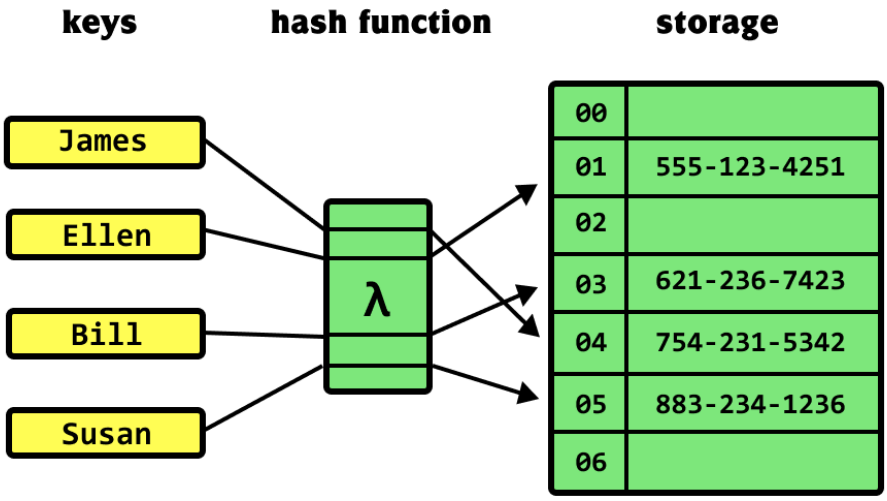
1. TreeMap


parent (null)
left
right
H
Ahmedabad
parent
parent
D
Jaipur
P
Patna
left
right
left (null)
right (null)
parent
parent
B
Delhi
F
Agra
left (null)
right (null)
left (null)
right (null)


H
D
P
B
F

Key를 기준으로 정렬되어 삽입되기 때문에 Sorted Map이라고도 할 수 있다.

2. HashMap



Key의 hash function 결과값으로 Value table의 index 값을 얻어냄.

Hasing

```java
public static int hashCode(byte[] value) {
    int h = 0;
    int length = value.length >> 1;
    for (int i = 0; i < length; i++) {
        h = 31 * h + getChar(value, i);
    }
    return h;
}
```

`hashCode` method in `JAVA StringUTF16` class

```java
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

`hash` method in `JAVA HashMap` class

※ Hash function은 구현 방식이 정해진것이 아님!

1. `key`를 hash화 하고 변수 `h`에 저장함.
2. 변수 `h`를 비트연산자 `>>>`로 16비트만큼 우측으로 밀어냄
   ex) 1101 1100 0010 0110(-23590) >>> 8 = 0000 0000 1101 1100(220)

   ※ `>>`는 빈 비트를 1로 채움
   ex) 1101 1100 0010 0110(-23590) >> 8 = 1111 1111 1101 1100(-23732)
3. 변수 `h`와 과정 2. 의 변수를 XOR 연산으로 계산함.
   ex) 1101 1100 0010 0110 ⊻ 0000 0000 1101 1100
   = 1101 1100 1111 1010(-23802)

작동 방식

Bucket과 Index

Q1) “a”의 해시값 = 97

“apple”의 해시값 = 93,029,210

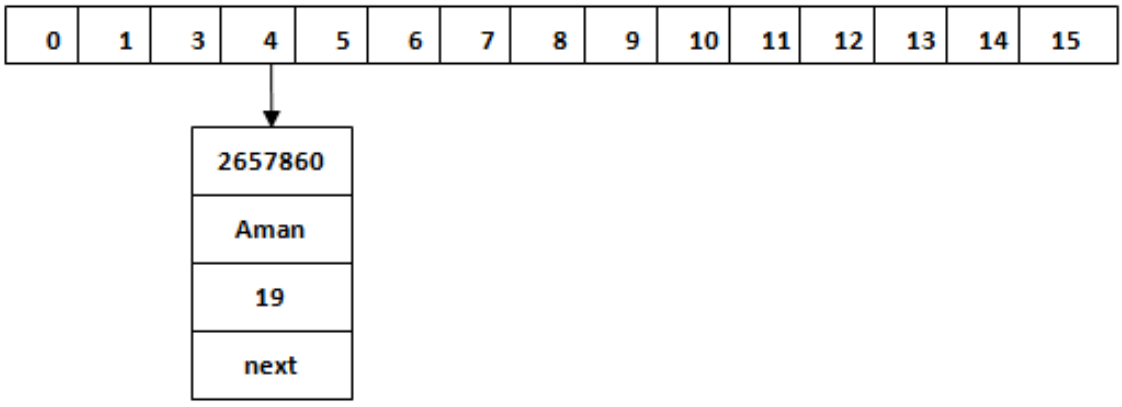
```
Key: a
Key as Ascii: 97
Key as Hash Code: 97

Key: apple
Key as Ascii: 97 112 112 108 101
Key as Hash Code: 93029210
```

그렇다면 우리는 키 “apple”의 Value를 저장하기 위해 93,028,210 길이를 가진 Array를 선언해야 하나?

```java
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, next: null);
```

`putVal` method in `JAVA HashMap`



Bucket의 길이와 위에서 구했던 hash 값을 & 비트 연산자로 계산함.
ex) 0000 0000 0000 1111(15) & 1101 1100 1111 1010(-23802)
= 0000 0000 0000 1010(10)

```
Key: a
Key as Ascii: 97
Key as Hash Code: 97
Key as Hasing: 97
Index: 1

Key: apple
Key as Ascii: 97 112 112 108 101
Key as Hash Code: 93029210
Key as Hashing: 93030097
Index: 1

Key: My key is VERY long!
Key as Ascii: 77 121 32 107 101 121 32 105 115 32 86 69 82 89 32 108 111 110 103 33
Key as Hash Code: -1332215524
Key as Hashing: -1332170364
Index: 4
```

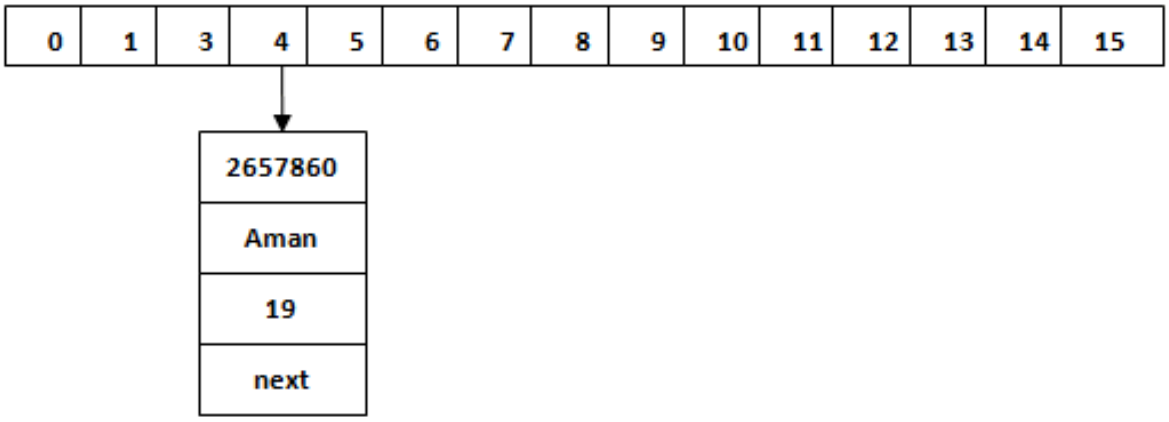
`Index of "a"` = 1

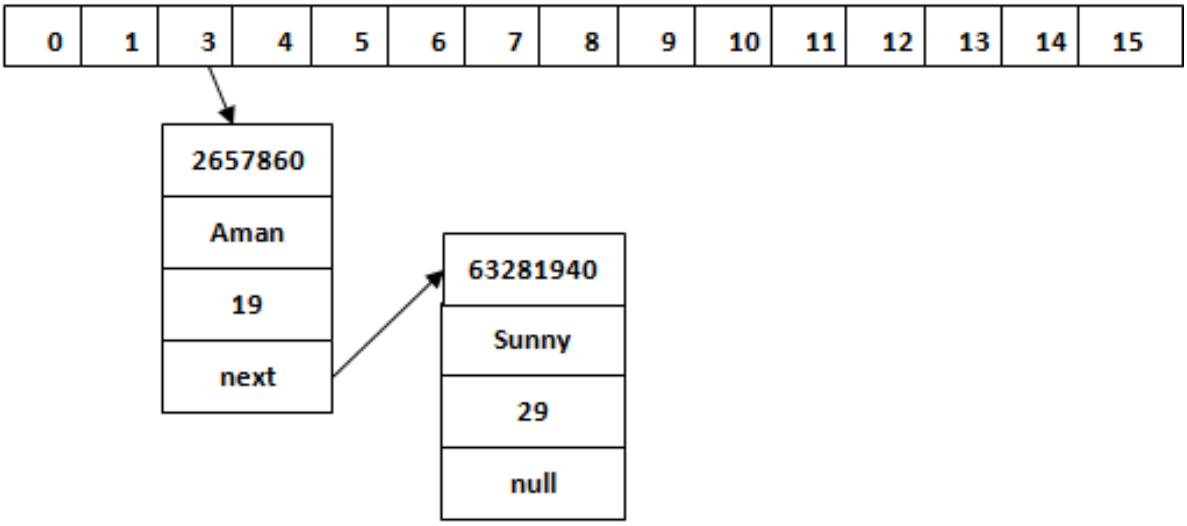
`Index of "apple"` = 1

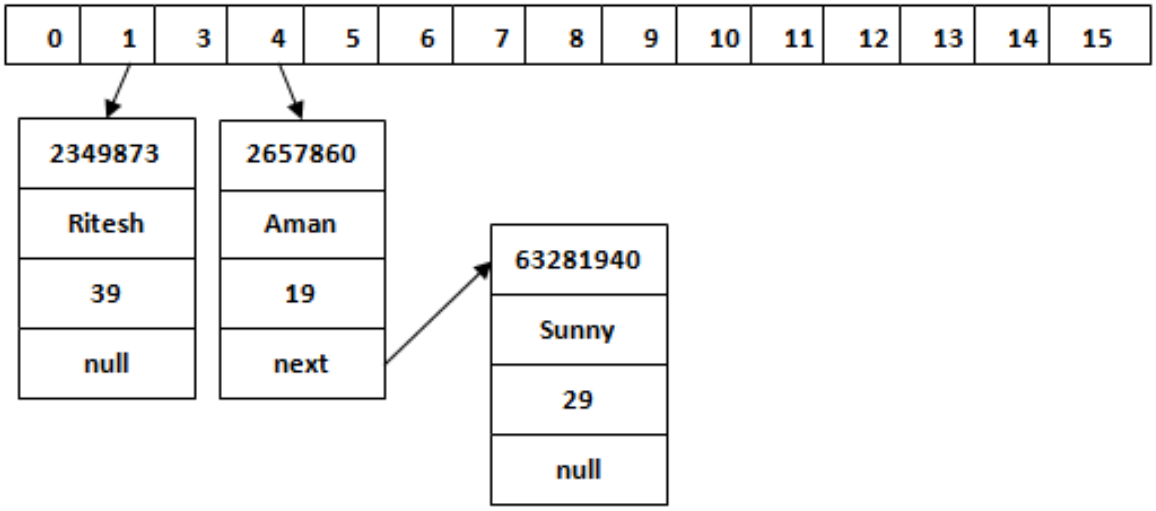
`Index of "My key is VERY long!"` = 4

예시

0010 1000 1000 1110 0100 0100(2,657,860) & 0000 … 1111(15) = 0100(4)







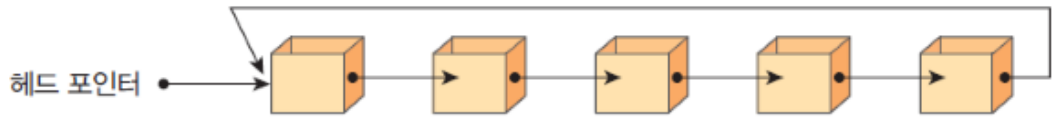
`Hashing`을 이용한 `Index` 생성만 안다면, HashMap은 N 개의 Linked List와 같다.

다른 자료구조와의 차이

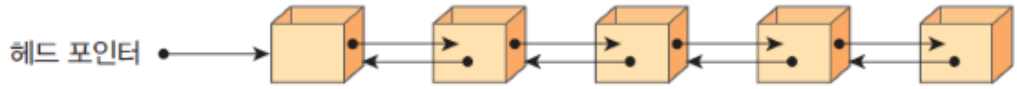
다른 자료구조

Linked List, Stack, Queue

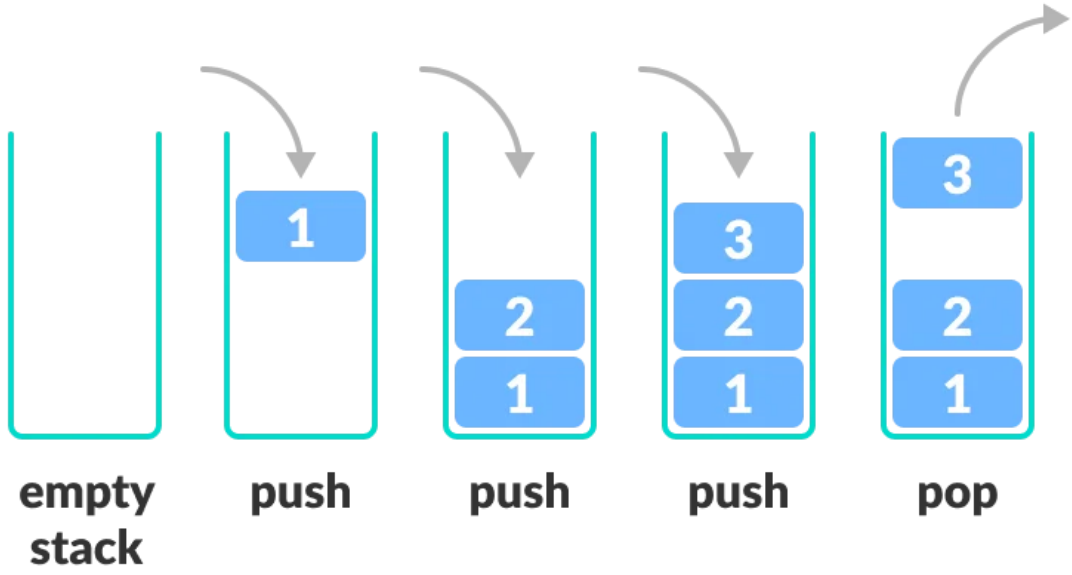
단순 연결 리스트

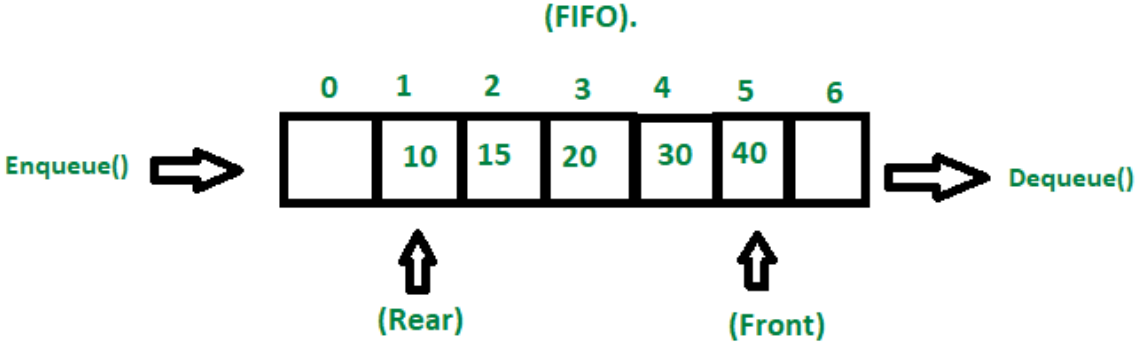


원형 연결 리스트



이중 연결 리스트





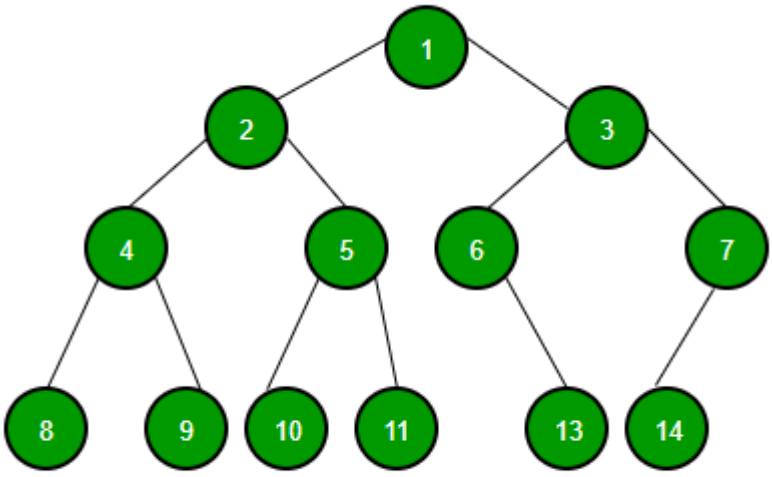
데이터가 많아질 수록 탐색이 어려워짐.

탐색 시간 = $O(n)$

Stack, Queue의 경우, 명확한 용도가 있어 활용 방식이 제한됨.

Linked List의 경우, Data Structure로써의 용도보다는 메모리 관리 방식을 새롭게 만들어 냈다는 의의가 존재

Binary Tree

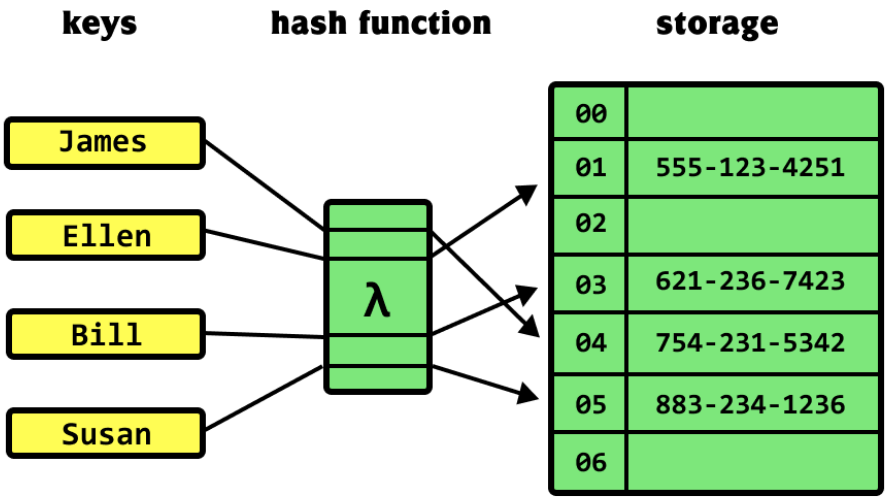


파생 형식의 데이터 관리로, 특정 주제에 대한 데이터 관리가 용이함. ex) File system, Deep-Learning

B-tree 등과 같이 정렬 알고리즘이 적용되어 있다면 데이터 정렬 및 탐색에도 이점이 존재함.

탐색 시간 = $O(logn)$

HashMap



Key 값의 확장성이 존재함.

Why? `JAVA Object`는 `hashCode()` method가 존재하므로 구현만 한다면 Key에 단순 변수가 아니라 Object도 가능함.

특정 Key를 알고 있다면 탐색이 용이함.

Key를 통해 Index를 얻어내면 Table에서 해당 index에 접근한 뒤 list에서 탐색할 수 있음.

정말 최악의 경우 한 Index에 모든 Node가 몰리는 경우가 존재할 수 있겠으나, 사실 상 불가능함.

탐색 시간 = $O(1)$