컬렉션 프레임워크 - Map

💡Collection Framework 의 개념

컬렉션

- 동일한 타입을 묶어 관리하는 자료 구조
- 저장 용량을 동적으로 관리
    - 배열과 컬렉션의 차이
      배열: 생성시 크기를 지정하고 변경 X
      컬렉션: 저장 데이터 개수에 제한 X

프레임워크

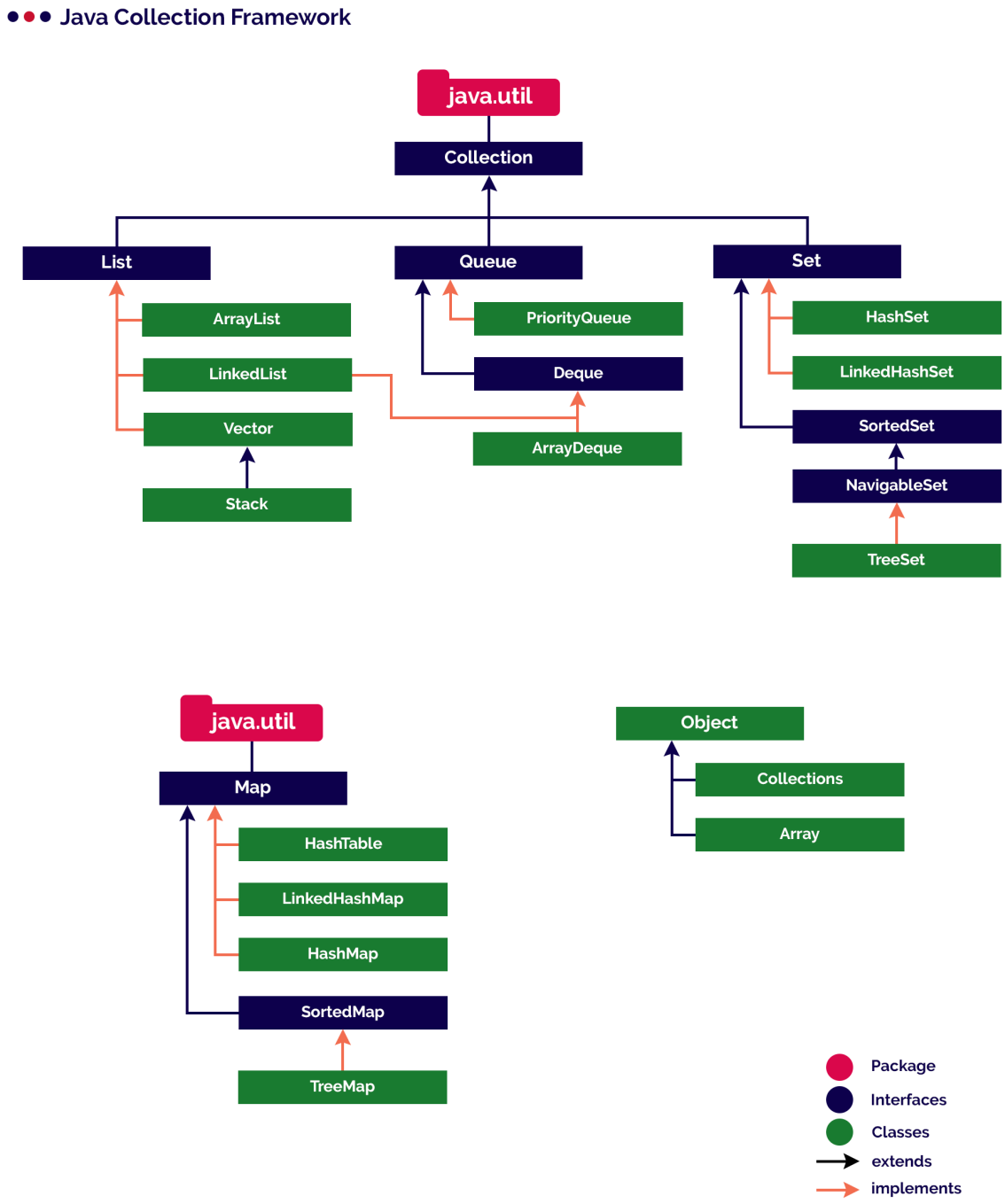
- 클래스와 인터페이스의 모임 (라이브러리)
- 클래스의 정의에 설계 원칙 또는 구조가 존재

컬렉션 프레임워크

- 데이터를 저장하는 자료 구조와 데이터를 처리하는 알고리즘을 구조화하여 클래스로 구현한 것
- 컬렉션을 쉽고 편리하게 다룰 수 있는 다양한 클래스 제공

Collection Framework 의 구조

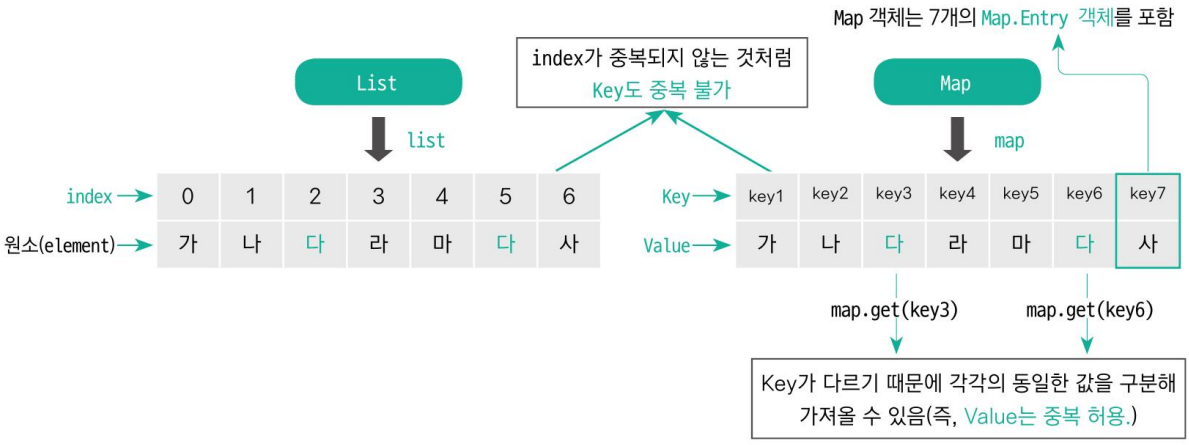
인터페이스	특징	구현 클래스
List	**순서가 있는** 데이터의 집합으로 데이터의 **중복을 허용함**	Vector, **ArrayList**, **LinkedList**, Stack, Queue
Set	**순서가 없는** 데이터의 집합으로 데이터의 **중복을 허용하지 않음**	HashSet, TreeSet
Map	**키와 값의 한 쌍**으로 이루어지는 데이터의 집합으로 **순서가 없음** (키는 중복을 허용하지 않고, 값은 중복을 허용함)	HashMap, Hashtable, TreeMap, Properties



💡Map <K, V> 컬렉션

공통 특징

- Collection 인터페이스를 상속받지 않고, 별도의 인터페이스로 존재
- **키(Key)** 와 **값(Value) 한 쌍**으로 데이터를 저장
- 저장 **순서 없음**
- Key 중복 **불가**, Value 중복 **가능**

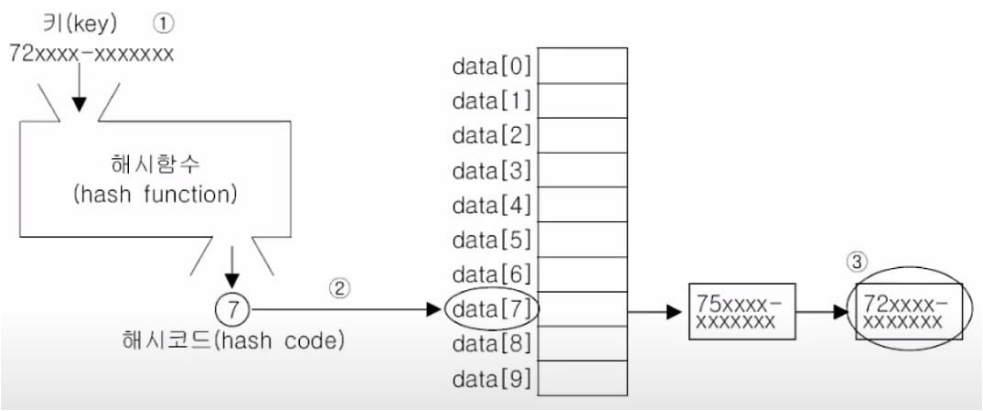


Map <K, V> 인터페이스의 주요 메서드

구분	리턴 타입	메서드명	기능
데이터 추가	V	put(K key, V value)	입력매개변수의 (key, vaule)를 Map 객체에 추가
	void	putAll(Map<? extends K, ? extends V> m)	입력매개변수의 Map 객체를 통째로 추가
데이터 변경	V	replace(K key, V value)	Key에 해당하는 값을 Value 값으로 변경(old 값 리턴) (단, 해당 Key가 없으면 null 리턴)
	boolean	replace(K key, V oldValue, V newValue)	(key, oldValue)의 쌍을 갖는 엔트리에서 oldValue를 newValue로 변경(단, 해당 엔트리가 없으면 false를 리턴)
데이터 정보 추출	V	get(Object key)	매개변수의 Key 값에 해당하는 oldValue를 리턴
	boolean	containsKey(Object key)	매개변수의 Key 값이 포함돼 있는지 여부
	boolean	containsValue(Object value)	매개변수의 Value 값이 포함돼 있는지 여부
	Set<K>	keySet()	Map 데이터들 중 Key들만 뽑아 Set 객체로 리턴
	Set<Entry<K, V>>	entrySet()	Map의 각 엔트리들을 Set 객체로 담아 리턴
	int	size()	Map에 포함된 엔트리의 개수
데이터 삭제	V	remove(Object key)	입력매개변수의 Key를 갖는 엔트리 삭제(단, 해당 Key가 없으면 아무런 동작을 하지 않음)
	boolean	remove(Object key, Object value)	입력매개변수의 (key, value)를 갖는 엔트리 삭제(단, 해당 엔트리가 없으면 아무런 동작을 하지 않음)
	void	clear()	Map 객체 내의 모든 데이터 삭제

HashMap <K, V>

- Map<K, V>의 대표적인 구현 클래스
- 해싱(hasing)기법으로 해시테이블에 데이터를 저장



- Key 중복 불가하지만, 같은 값을 다른 Key 로 저장하는것은 가능
- 입력 순서와 출력 순서는 동일하지 않을 수 있음
- key 와 value 에 null 을 허용 O
- 초기 용량의 기본값은 16이고, 개수가 16을 넘어가면 자동으로 늘어남

주요 메서드

```
HashMap<Integer, String> hm1 = new HashMap<Integer, String>();

// 요소의 저장
hm1.put(3, "three");
hm1.put(1, "one");
hm1.put(2, "two");

// 요소의 출력
System.out.println(hm1.toString());

Map<Integer, String> hm2 = new HashMap<Integer, String>();
hm2.putAll(hm1);
System.out.println(hm2.toString());
```

```
{1=one, 2=two, 3=three}
```

```
{1=one, 2=two, 3=three}
```

입력 순서와 불일치

```
// 데이터 변경 replace(K key, V value)
hm2.replace(1, "하나");
hm2.replace(4, "넷");  // 동작하지 않음
System.out.println(hm2.toString());
// replace(K key, V oldValue, V newValue)
hm2.replace(1, "하나", "하나하나");
System.out.println(hm2.toString());

// 요소의 추출 V get(object key)
```

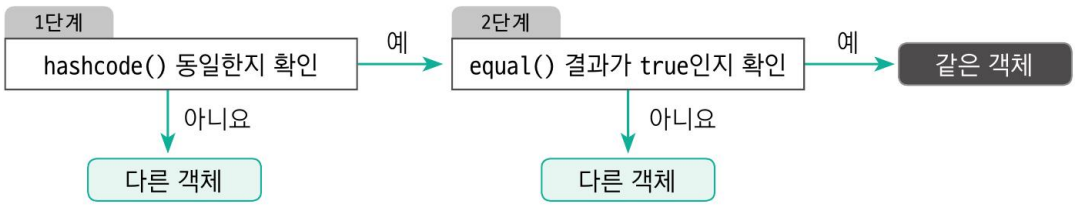
```
System.out.println(hm2.get(1));
System.out.println(hm2.get(2));
// containsKey(object key)
System.out.println(hm2.containsKey(1));
System.out.println(hm2.containsKey(5));
// containsValue(object value)
System.out.println(hm2.containsValue("하나하나"));
System.out.println(hm2.containsValue("가나다"));
// Set<K> keySet()
Set<Integer> keySet = hm2.keySet();
System.out.println(keySet.toString());
// Set<Map.Entry<K, V>> entrySet()
Set<Map.Entry<Integer, String>> entrySet = hm2.entrySet();
System.out.println(entrySet);
// size
System.out.println(hm2.size());
```

```
{1=하나, 2=two, 3=three}
{1=하나하나, 2=two, 3=three}
하나하나
two
true
false
true
false
[1, 2, 3]
[1=하나하나, 2=two, 3=three]
3
```

```
// 데이터 삭제 remove(Object key)
hm2.remove(1);
hm2.remove(4);  // 동작 안 함
System.out.println(hm2.toString());
// remove(Object key, Object value)
hm2.remove(2, "나다라");
hm2.remove(3, "다다다");  // 동작 안 함
System.out.println(hm2.toString());
// clear()
hm2.clear();
System.out.println(hm2.toString());
```

```
{2=two, 3=three}
{3=three}
{}
```

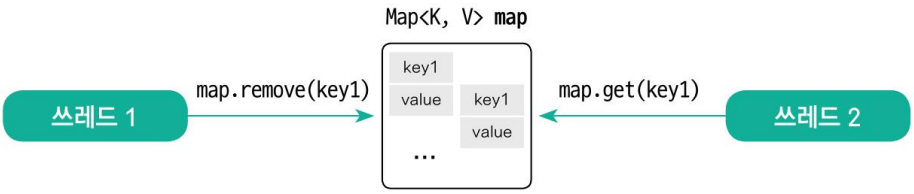
Key 값의 중복 여부



- Key 객체의 hashCode() 값이 같고, equals() 메서드가 true를 리턴하면 같은 객체로 인식
- hashCode() 는 객체가 저장된 번지와 연관된 값 (실제 번지와는 다름)
- equals() 는 == 와 동일한 연산 (저장 번지 비교)
- hashCode() 와 equals() 가 Overriding 되어 있어야 함

Hashtable <K, V>

- HashMap 클래스의 과거 버전
- Hashtable 보다는 HashMap 클래스를 사용하는 것이 좋음
- 입력 순서와 출력 순서는 동일하지 않을 수 있음
- key 와 value 에 null 을 허용 X
- 동기화 메서드로 구현되어 멀티쓰레드에 적합



주요 메소드

- 생성자 변경을 제외한 나머지는 HashMap<K, V> 와 동일

```
Map<Integer, String> hTable1 = new Hashtable<Integer, String>();
```

`HashMap<>()` 생성자를 `Hashtable<>()` 생성자로 변경

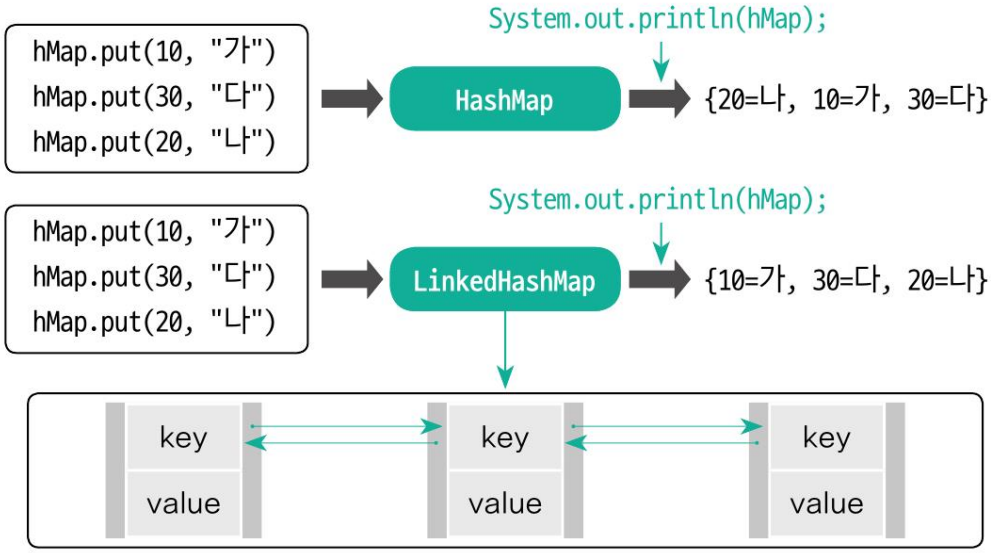
```
// 요소의 저장 put(K key, V value)
hTable1.put(3, "three");
hTable1.put(1, "one");
hTable1.put(2, "two");
System.out.println(hTable1.toString());
```

```
{3=three, 2=two, 1=one}
```

입력 순서와 불일치

LinkedHashMap <K, V>

- HashMap<K, V> 과 동일한 기능 수행
- 입력 순서 = 출력 순서



주요 메소드

- 출력이 입력의 순으로 나오는 것을 제외하면 HashMap<K, V> 와 동일

```
Map<Integer, String> lnMap1 = new LinkedHashMap<Integer, String>();
```

`LinkedHashMap<>()` 생성자로 변경

```
// 요소의 저장 put(K key, V value)
lhMap1.put(3, "three");
lhMap1.put(1, "one");
lhMap1.put(2, "two");
System.out.println(lhMap1.toString());

// 요소의 추출 Set<K> keySet()
Set<Integer> keySet = lhMap1.keySet();
System.out.println(keySet.toString());
```

```
{3=three, 1=one, 2=two}
[3, 1, 2]
```

HashMap, Hashtable 과 다르게 입력 순서와 출력 순서 동일

TreeMap <K, V>

- Map<K, V> 의 기본 기능에 정렬 및 검색 기능 추가된 컬렉션
- 입력 순서와 관계없이 Key 값의 크기순으로 출력 (Key 값은 대소 비교가 가능해야함)

Map<K, V>로 객체 타입을 선언할 때

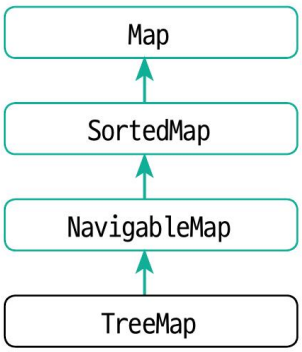
```
Map<Integer, String> treeMap = new TreeMap<Integer, String>();
treeMap.
```

Map<K, V> 메서드만 사용 가능

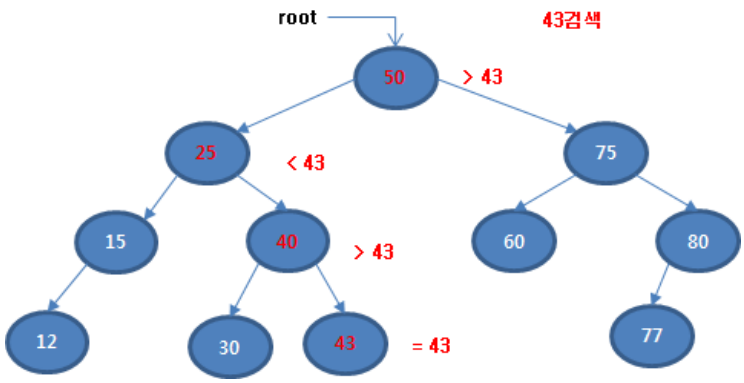
TreeMap<K, V>로 객체 타입을 선언할 때

```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
treeMap.
```

Map<K, V> 메서드와 추가된 정렬/검색 메서드 사용 가능



- 데이터를 이진 탐색 트리(binary search tree) 형태로 저장
  - 이진 탐색 트리?
    - 하나의 노드에 연결된 노드가 최대 2개의 하위(자식) 노드를 가짐
    - 부모보다 작은 값은 왼쪽, 큰 값은 오른쪽에 저장



- 범위 검색(from ~ to)과 정렬에 유리
- HashMap 보다 데이터 추가, 삭제에 시간이 더 걸림
- Key 중복 불가하지만, 같은 값을 다른 Key 로 저장하는것은 가능

주요 메소드

- 기본적인 Map<K, V>의 주요 메소드 종류와 활용법은 다른 클래스와 동일
- 추가로 사용할 수 있는 정렬/검색 메서드

구분	리턴 타입	메서드명	기능
데이터 검색	K	firstKey()	Map 원소 중 가장 작은 Key 값 리턴
	Map.Entry<K, V>	firstEntry()	Map 원소 중 가장 작은 Key 값을 갖는 엔트리 리턴
	K	lastKey()	Map 원소 중 가장 큰 Key 값 리턴
	Map.Entry<K,V>	lastEntry()	Map 원소 중 가장 큰 Key 값을 갖는 엔트리 리턴
	K	lowerKey(K key)	매개변수로 입력된 Key 값보다 작은 Key 값 중 가장 큰 Key 값 리턴
	Map.Entry<K, V>	lowerEntry(K key)	매개변수로 입력된 Key 값보다 작은 Key 값 중 가장 큰 Key 값을 갖는 엔트리 리턴
	K	higherKey(K key)	매개변수로 입력된 Key 값보다 큰 Key 값 중 가장 작은 Key 값 리턴
	Map.Entry<K, V>	higherEntry(K key)	매개변수로 입력된 Key 값보다 큰 Key 값 중 가장 작은 Key 값을 갖는 엔트리 리턴
데이터 추출	Map.Entry<K, V>	pollFirstEntry()	Map 원소 중 가장 작은 Key 값을 갖는 엔트리를 꺼내 리턴
	Map.Entry<K, V>	pollLastEntry()	Map 원소 중 가장 큰 Key 값을 갖는 엔트리를 꺼내 리턴
데이터 부분 집합 생성	SortedMap<K, V>	headMap(K toKey)	toKey 미만의 Key 값을 갖는 모든 엔트리를 포함한 Map 객체 리턴(toKey 미포함)
	NavigableMap<K, V>	headMap(K toKey, boolean inclusive)	toKey 미만/이하의 Key 값을 갖는 모든 엔트리를 포함한 Map 객체 리턴(inclusive=true이면 toKey 포함, inclusive=false이면 toKey 미포함)
	SortedMap<K, V>	tailMap(K fromKey)	fromKey 이상인 Key 값을 갖는 모든 엔트리를 포함한 Map 객체 리턴(fromKey 포함)
	NavigableMap<K, V>	tailMap(K fromKey, boolean inclusive)	fromKey 초과/이상인 Key 값을 갖는 모든 엔트리를 포함한 Map 객체 리턴(inclusive=true이면 fromKey 포함, inclusive=false이면 fromKey 미포함)
	SortedMap<K, V>	subSet(K fromKey, K toKey)	fromKey 이상 toKey 미만의 Key 값을 갖는 모든 엔트리를 포함한 Map 객체 리턴(fromKey 포함, toKey 미포함)
	NavigableMap<K, V>	subSet(K fromKey, boolean frominclusive, K toKey, boolean toinclusive)	fromKey 초과/이상 toKey 미만/이하인 Key 값을 갖는 모든 엔트리를 포함한 Map 객체 리턴(frominclusive=true/false이면 fromKey 포함/미포함, toinclusive=true/false이면 toKey 포함/미포함)
데이터 정렬	NavigableSet<K>	descendingKeySet()	Map에 포함된 모든 Key 값의 정렬을 반대로 변환한 Set 객체 리턴
	NavigableMap<K, V>	descendingMap()	Map에 포함된 모든 Key 값의 정렬을 반대로 변환한 Map 객체 리턴

크기 비교

```
TreeMap<Integer, String> treeMap1 = new TreeMap<Integer, String>();

// Integer 크기 비교
Integer intValue1 = new Integer(20);
Integer intValue2 = new Integer(10);

// intValue1 > intValue2
treeMap1.put(intValue1, "가나다");
treeMap1.put(intValue2, "나다라");
System.out.println(treeMap1.toString());
```

```
{10=나다라, 20=가나다}
```

```
// String 크기 비교
TreeMap<String, Integer> treeMap2 = new TreeMap<String, Integer>();
String str1 = "가나";
String str2 = "다라";

// str1 < str2
treeMap2.put(str1, 10);
treeMap2.put(str2, 20);
System.out.println(treeMap2.toString());
```

```
{가나=10, 다라=20}
```