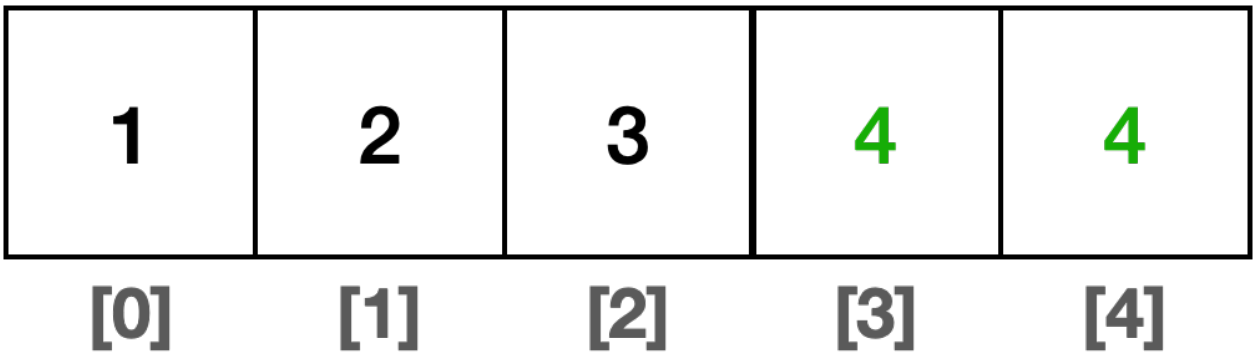
6. 컬렉션 프레임워크 - 해시(Hash)

#1.인강/0.자바/4.자바-중급2편

리스트(List) vs 세트(Set)

자료 구조에서의 List와 Set은 각각 특정한 방식으로 데이터를 저장하고 관리하는 데 사용된다.

List (리스트)

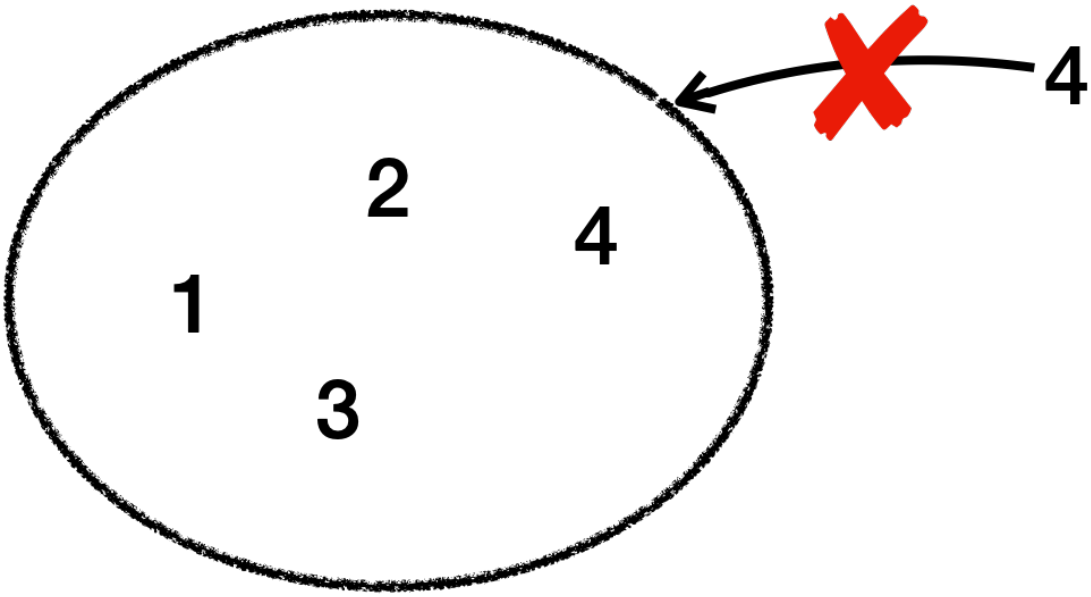


정의: 리스트는 요소들의 순차적인 컬렉션이다. 요소들은 특정 순서를 가지며, 같은 요소가 여러 번 나타날 수 있다.
특징:

- **순서 유지**: 리스트에 추가된 요소는 특정한 순서를 유지한다. 이 순서는 요소가 추가된 순서를 반영할 수 있다.
- **중복 허용**: 리스트는 동일한 값이나 객체의 중복을 허용한다. 예를 들어, 같은 숫자나 문자열을 리스트 안에 여러 번 저장할 수 있다.
- **인덱스 접근**: 리스트의 각 요소는 인덱스를 통해 접근할 수 있다. 이 인덱스는 보통 0부터 시작한다.

용도: 순서가 중요하거나 중복된 요소를 허용해야 하는 경우에 주로 사용된다.

Set (세트, 셋)



- **정의**: 세트(셋)는 유일한 요소들의 컬렉션이다. 참고로 세트보다는 셋으로 많이 불린다.
- **특징**:
  - **유일성**: 셋에는 중복된 요소가 존재하지 않는다. 셋에 요소를 추가할 때, 이미 존재하는 요소면 무시된다.
  - **순서 미보장**: 대부분의 셋 구현에서는 요소들의 순서를 보장하지 않는다. 즉, 요소를 출력할 때 입력 순서와 다를 수 있다.
  - **빠른 검색**: 셋은 요소의 유무를 빠르게 확인할 수 있도록 최적화되어 있다. 이는 데이터의 중복을 방지하고 빠른 조회를 가능하게 한다.
- **용도**: 중복을 허용하지 않고, 요소의 유무만 중요한 경우에 사용된다.

예시:

- **List**: 장바구니 목록, 순서가 중요한 일련의 이벤트 목록.
- **Set**: 회원 ID 집합, 고유한 항목의 집합.

직접 구현하는 Set0 - 시작

셋을 구현하는 것은 아주 단순하다. 인덱스가 없기 때문에 단순히 데이터를 넣고, 데이터가 있는지 확인하고, 데이터를 삭제하는 정도면 충분하다. 그리고 데이터를 추가할 때 중복 여부만 체크해주면 된다.

- `add(value)` : 셋에 값을 추가한다. 중복 데이터는 저장하지 않는다.
- `contains(value)` : 셋에 값이 있는지 확인한다.
- `remove(value)` : 셋에 있는 값을 제거한다.

최대한 단순하게 셋을 구현해보자. 예제의 단순함을 위해 `add()` , `contains()` 만 구현하자.

셋 직접 구현하기

```
package collection.set;

import java.util.Arrays;

public class MyHashSetV0 {

    private int[] elementData = new int[10];
    private int size = 0;

    // O(n)
    public boolean add(int value) {
        if (contains(value)) {
            return false;
        }
        elementData[size] = value;
        size++;
        return true;
    }

    // O(n)
    public boolean contains(int value) {
        for (int data : elementData) {
            if (data == value) {
                return true;
            }
        }
        return false;
    }

    public int getSize() {
        return size;
    }

    @Override
    public String toString() {
        return "MyHashSetV0{" +
                "elementData=" + Arrays.toString(Arrays.copyOf(elementData,
size)) +
                ", size=" + size +
                '}';
    }
```

```
}
```

- 예제에서는 단순함을 위해 배열에 데이터를 저장한다. 배열의 크기도 10으로 고정했다.
- `add(value)` : 셋에 중복된 값이 있는지 체크하고, 중복된 값이 있으면 `false`를 반환한다. 중복된 값이 없으면 값을 저장하고 `true`를 반환한다.
- `contains(value)` : 셋에 값이 있는지 확인한다. 값이 있으면 `true`를 반환하고, 값이 없으면 `false`를 반환한다.
- `toString()` : 배열을 출력하는 부분에 `Arrays.copyOf()`를 사용해서 배열에 데이터가 입력된 만큼만 출력한다.

```
package collection.set;

public class MyHashSetV0Main {

    public static void main(String[] args) {
        MyHashSetV0 set = new MyHashSetV0();
        set.add(1); // O(1)
        set.add(2); // O(n)
        set.add(3); // O(n)
        set.add(4); // O(n)
        System.out.println(set);

        boolean result = set.add(4);//중복 데이터 저장
        System.out.println("중복 데이터 저장 결과 = " + result);
        System.out.println(set);

        System.out.println("set.contains(3): " + set.contains(3)); // O(n)
        System.out.println("set.contains(99): " + set.contains(99)); // O(n)
    }
}
```

실행 결과

```
MyHashSetV0{elementData=[1, 2, 3, 4], size=4}
중복 데이터 저장 결과 = false
MyHashSetV0{elementData=[1, 2, 3, 4], size=4}
set.contains(3): true
set.contains(99): false
```

- `add()`로 데이터를 추가할 때 셋에 중복 데이터가 있는지 전체 데이터를 항상 확인해야 한다. 따라서 O(n)으로 입력 성능이 나쁘다.
  - 중복 데이터 검색 O(n) + 데이터 입력 O(1) ➜ O(n)
- `contains()`로 데이터를 찾을 때는 배열에 있는 모든 데이터를 찾고 비교해야 하므로 평균 O(n)이 걸린다.

정리

우리가 만든 셋은 구조는 단순하지만, 데이터 추가, 검색 모두 O(n)으로 성능이 좋지 않다. 특히 데이터가 많을 수 록 효율은 매우 떨어진다. 검색의 경우 이전에 보았던 `ArrayList`, `LinkedList`도 O(n)이어서 어느정도 받아들 수 있지만, 데이터의 추가가 특히 문제이다. 데이터를 추가할 때마다 중복 데이터가 있는지 체크하기 위해 셋의 전체 데이터를 확인해야 한다. 이때 O(n)으로 성능이 떨어진다. 데이터를 추가할 때마다 이렇게 성능이 느린 자료 구조는 사용하기 어렵다.

결국 중복 데이터를 찾는 부분이 성능의 발목을 잡는 것이다. 이런 부분을 어떻게 개선할 수 있을까?

해시 알고리즘1 - 시작

해시(hash) 알고리즘을 사용하면 데이터를 찾는 검색 성능을 평균 O(1)로 비약적으로 끌어올릴 수 있다.

해시 알고리즘을 이해하기 위해 먼저 간단한 예제를 만들어보자.

문제

- 입력: 0~9 사이의 여러 값이 입력된다. 중복된 값은 입력되지 않는다.
- 찾기: 0~9 사이의 값이 하나 입력된다. 입력된 값 중에 찾는 값이 있는지 확인해보자.

```
package collection.set;

import java.util.Arrays;

public class HashStart1 {

    public static void main(String[] args) {
        Integer[] inputArray = new Integer[4];
        inputArray[0] = 1;
        inputArray[1] = 2;
        inputArray[2] = 5;
        inputArray[3] = 8;
```

```
        System.out.println("inputArray = " + Arrays.toString(inputArray));

        int searchValue = 8;
        //4번 반복 O(n)
        for (int inputValue : inputArray) {
            if (inputValue == searchValue) {
                System.out.println(inputValue);
            }
        }
    }
}
```

실행 결과

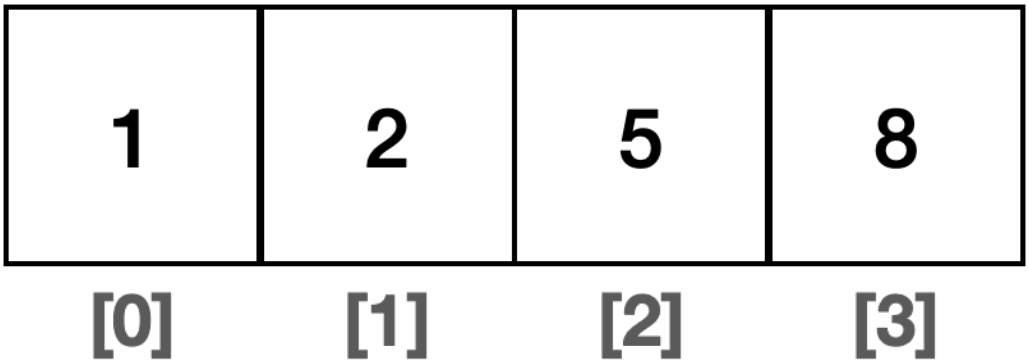
```
inputArray = [1, 2, 5, 8]
8
```

입력 값은 1, 2, 5, 8이다. 이 값을 배열에 넣고, 배열에서 검색 값 8을 찾아보자. 이 값을 찾으려면 배열에 들어있는 데이터를 모두 찾아서 값을 비교해야 한다. 따라서 배열에서 특정 데이터를 찾는 성능은 O(n)으로 느리다. 물론 데이터가 많아질 수록 더 느려진다.
여기서 문제의 핵심은 찾기 성능이 O(n)으로 느리다는 점이다.

해시 알고리즘2 - index 사용

배열은 인덱스의 위치를 사용해서 데이터를 찾을 때 O(1)로 매우 빠른 특징을 가지고 있다.
반면에 데이터를 검색할 때는 배열에 들어있는 데이터 하나하나를 모두 비교해야 하므로 인덱스를 활용할 수 없다.
그런데 만약에 데이터를 검색할 때도 인덱스를 활용해서 데이터를 한 번에 찾을 수 있다면 어떻게 될까?
이렇게만 할 수 있다면 O(n) ➜ O(1)로 바꾸어서 성능을 획기적으로 끌어올릴 수 있을 것이다.

물론 인덱스와 데이터의 값은 서로 다르기 때문에 이것은 불가능하다.



- 인덱스 0 ➜ 데이터 1
- 인덱스 1 ➜ 데이터 2
- 인덱스 2 ➜ 데이터 5
- 인덱스 3 ➜ 데이터 8

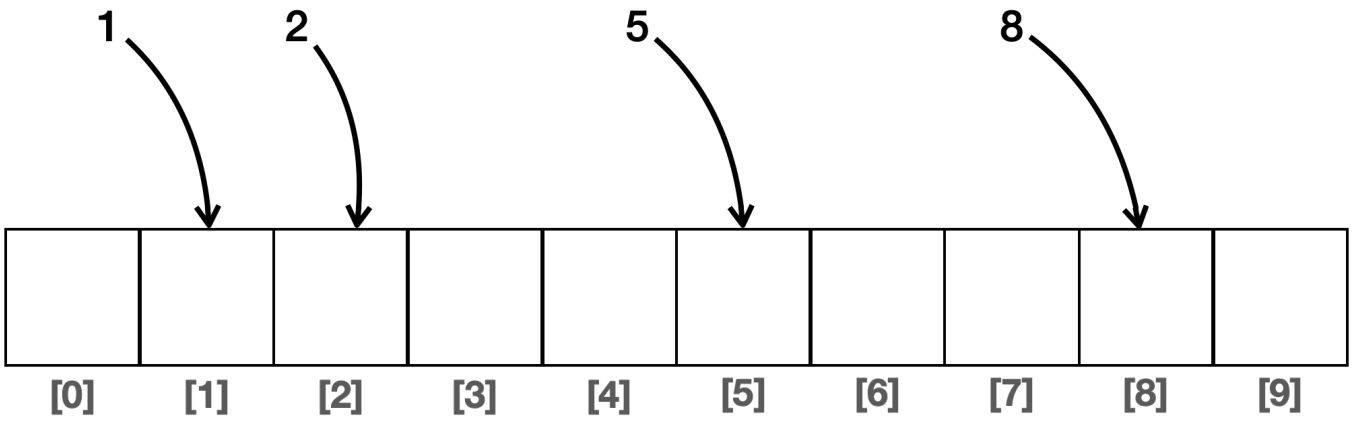
인덱스 0에는 데이터 1이 들어있고, 인덱스 3에는 데이터 8이 들어있다.

결국 여기에 8이라는 데이터가 있는지 찾으려면 순서대로 모든 데이터를 비교해야 한다.

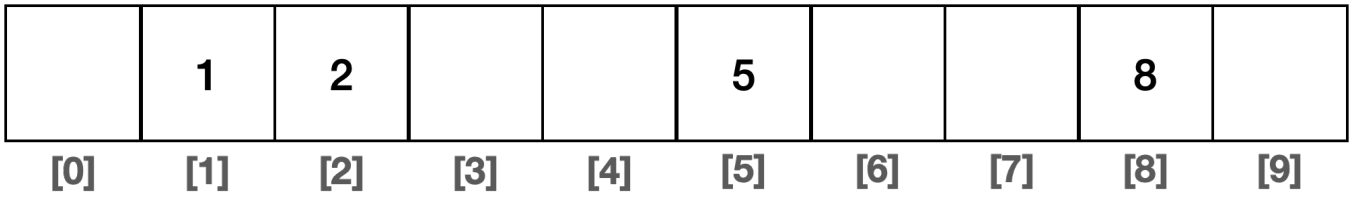
여기서 생각의 틀을 완전히 뒤집어보자.

데이터의 값 자체를 배열의 인덱스와 맞추어 저장하면 어떨까? 그러니까 데이터의 값 자체를 배열의 인덱스로 사용하는 것이다!

데이터의 값과 배열의 인덱스를 맞추어 입력 - 시도



데이터의 값과 배열의 인덱스를 맞추어 입력 - 결과



- 인덱스 1 ➜ 데이터 1
- 인덱스 2 ➜ 데이터 2
- 인덱스 5 ➜ 데이터 5
- 인덱스 8 ➜ 데이터 8

인덱스 1에는 데이터 1이 들어있고, 인덱스 8에는 데이터 8이 들어있다.
저장하는 데이터와 인덱스를 완전히 맞춘 것이다. 이제 인덱스 번호가 데이터가 되고, 데이터가 인덱스 번호가 되었다.

이제 데이터를 검색해보자.

- 데이터 1을 찾으려면 `array[1]`을 입력하면 된다.
- 데이터 2를 찾으려면 `array[2]`을 입력하면 된다.
- 데이터 5를 찾으려면 `array[5]`을 입력하면 된다.
- 데이터 8를 찾으려면 `array[8]`을 입력하면 된다.

이제 데이터가 인덱스 번호가 된다. 따라서 배열의 인덱스를 활용해서 단번에 필요한 데이터를 찾을 수 있다.
데이터의 값을 인덱스 번호로 사용하는 아주 간단한 아이디어 하나로 O(n)의 검색 연산을 O(1)의 검색 연산으로 바꿀 수 있다!

실제 코드로 구현해보자.

```
package collection.set;

import java.util.Arrays;

public class HashStart2 {

    public static void main(String[] args) {
        //입력: 1, 2, 5, 8
        //[null, 1, 2, null, null, 5, null, null, 8, null]
        Integer[] inputArray = new Integer[10];
        inputArray[1] = 1;
        inputArray[2] = 2;
        inputArray[5] = 5;
        inputArray[8] = 8;
        System.out.println("inputArray = " + Arrays.toString(inputArray));

        int searchValue = 8;
        Integer result = inputArray[searchValue]; // O(1)
        System.out.println(result);
    }
}
```

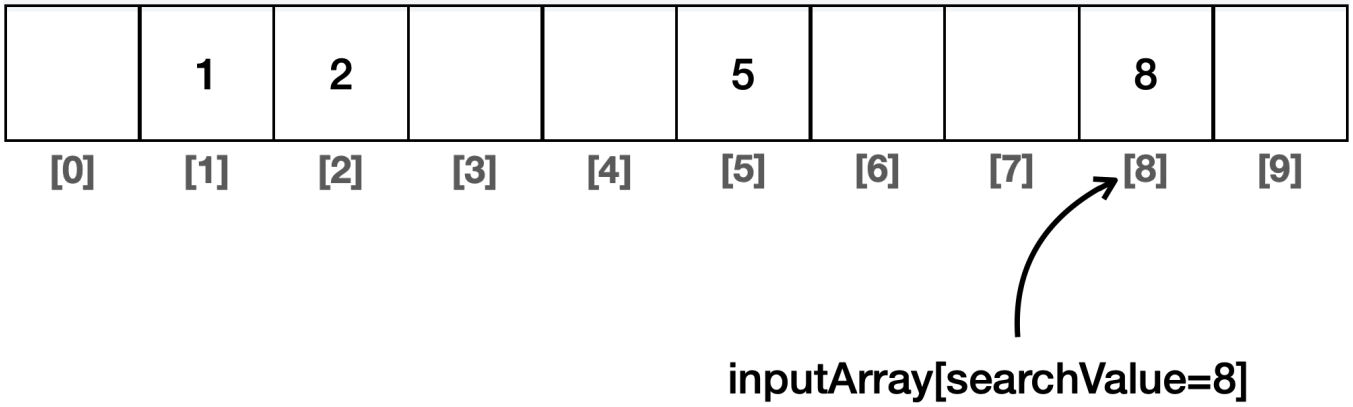
실행 결과

```
inputArray = [null, 1, 2, null, null, 5, null, null, 8, null]
8
```

데이터를 입력할 때 배열에 순서대로 입력하는 것이 아니라, 데이터의 값을 인덱스로 사용해서 저장했다.

데이터를 조회할 때 데이터의 값을 인덱스로 사용해서 조회한다.

```
int searchValue = 8;
Integer result = inputArray[searchValue]; // O(1)
```



배열에서 인덱스로 조회하는 것은 O(1)로 매우 빠르다.

정리

데이터의 값 자체를 배열의 인덱스로 사용했다. 배열에서 인덱스로 데이터를 찾는 것은 매우 빠르다. 그 덕분에 O(n)의 검색 성능을 O(1)로 획기적으로 개선할 수 있었다.

문제

입력 값의 범위 만큼 큰 배열을 사용해야 한다. 따라서 배열에 낭비되는 공간이 많이 발생한다. 이 문제를 더 알아보자.

해시 알고리즘3 - 메모리 낭비

이번에는 입력 값의 범위를 0~99로 넓혀보자.

문제

- 입력: **0~99** 사이의 여러 값이 입력된다. 중복된 값은 입력되지 않는다.
- 찾기: **0~99** 사이의 값이 하나 입력된다. 입력된 값 중에 찾는 값이 있는지 확인해보자.

검색 속도를 높이기 위해 앞서 학습한 것 처럼 데이터의 값을 배열의 인덱스로 사용하자.
따라서 데이터가 `0~99` 까지 입력될 수 있다면 인덱스도 `0~99` 까지 사용할 수 있어야 한다. 따라서 크기 100의 배열이 필요하다.

```
package collection.set;

import java.util.Arrays;

public class HashStart3 {

    public static void main(String[] args) {
        //{1, 2, 5, 8, 14, 99}
        //[null, 1, 2, null, null, 5, null, null, 8, .., 14 ....., 99]
        Integer[] inputArray = new Integer[100];
        inputArray[1] = 1;
        inputArray[2] = 2;
        inputArray[5] = 5;
        inputArray[8] = 8;
        inputArray[14] = 14;
        inputArray[99] = 99;
        System.out.println("inputArray = " + Arrays.toString(inputArray));

        int searchValue = 99;
        Integer result = inputArray[searchValue]; // O(1)
        System.out.println(result);
    }
}
```

실행 결과

```
inputArray = [null, 1, 2, null, null, 5, null, null, 8, null, null, null,
null, null, 14, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
```

```
null, null, null, null, null, null, null, null, null, 99]

99
```

한계

데이터의 값을 인덱스로 사용한 덕분에 O(1)의 매우 빠른 검색 속도를 얻을 수 있다. 그리고 이 코드는 정상적으로 수행된다. 하지만 낭비되는 메모리 공간이 너무 많다.

만약 입력 값의 범위를 `0~99`를 넘어서 `int` 숫자의 모든 범위를 입력할 수 있도록 하려면 배열의 크기를 얼마로 잡아야 할까?

- **0~99 까지 범위 입력**
  - 사이즈 100의 배열이 필요: 4byte * 100 (단순히 값의 크기로만 계산)
- **int 범위 입력**
  - `int` 범위: -2,147,483,648 ~ 2,147,483,647
  - 약 42억 사이즈의 배열 필요 (+- 모두 포함)
  - 4byte * 42억 = 약 17기가바이트 필요 (단순히 값의 크기로만 계산)

데이터의 값을 인덱스로 사용할 때, 입력할 수 있는 값의 범위가 `int`라면 한번의 연산에 최신 컴퓨터의 메모리가 거의 다 소모되어 버린다. 만약 사용자가 1, 2, 1000, 200000의 네 개의 값만 입력한다면 나머지 대부분의 메모리가 빈 공간으로 낭비될 것이다. 뿐만 아니라 처음 배열을 생성하기 위해 메모리를 할당하는데도 너무 오랜 시간이 소모된다. 따라서 데이터의 값을 인덱스로 사용하는 방식은 입력 값의 범위가 넓다면 사용하기 어려워 보인다.

데이터의 값을 인덱스로 사용하는 방법은 매우 빠른 성능을 보장하지만, 입력 값의 범위가 조금만 커져도 메모리 낭비가 너무 심하다. 따라서 그대로 사용하기에는 문제가 있다.

해시 알고리즘4 - 나머지 연산

앞에서 이야기한 것 처럼 모든 숫자를 입력할 수 있다고 가정하면, 입력값의 범위가 너무 넓어져서 데이터의 값을 인덱스로 사용하기는 어렵다. 하지만 입력 값의 범위가 넓어도 해당 범위의 값을 모두 다 입력하는 것은 아니다.
앞의 예에서 0 ~ 99 범위의 값 중에 1, 2, 5, 8, 14, 99만 입력했다. 따라서 대부분의 공간은 낭비되었다.

공간도 절약하면서, 넓은 범위의 값을 사용할 수 있는 방법이 있는데, 바로 나머지 연산을 사용하는 것이다.
저장할 수 있는 배열의 크기(CAPACITY)를 10이라고 가정하자. 그 크기에 맞추어 나머지 연산을 사용하면 된다.

나머지 연산

- `1 % 10 = 1`
- `2 % 10 = 2`
- `5 % 10 = 5`
- `8 % 10 = 8`
- `14 % 10 = 4`
- `99 % 10 = 9`

여기서 14, 99는 10보다 큰 값이다. 따라서 일반적인 방법으로는 크기가 10인 배열의 인덱스로 사용할 수 없다.
하지만 나머지 연산의 결과를 사용하면 14는 4로, 99는 9로 크기가 10인 배열의 인덱스로 활용할 수 있다.
나머지 연산의 결과는 절대로 배열의 크기를 넘지 않는다. 예를 들어 나머지 연산에 10을 사용하면 결과는 `0~9` 까지만 나온다. 절대로 10이 되거나 10을 넘지 않는다. 따라서 연산 결과는 배열의 크기를 넘지 않으므로 안전하게 인덱스로 사용할 수 있다.

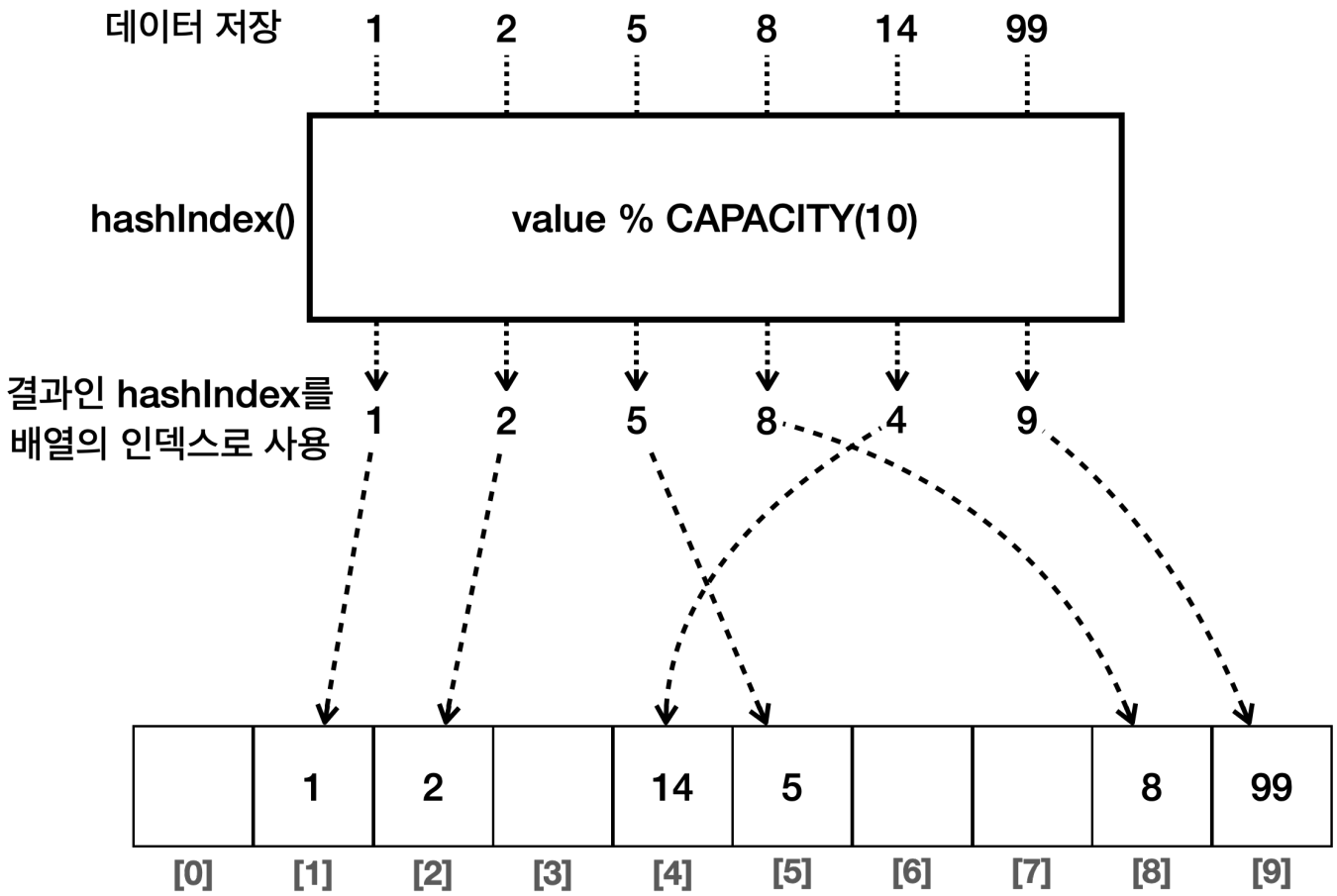
- 예)
- `9 % 10 -> 9`
- `10 % 10 -> 0`
- `11 % 10 -> 1`

해시 인덱스

이렇게 배열의 인덱스로 사용할 수 있도록 원래의 값을 계산한 인덱스를 해시 인덱스(hashIndex)라 한다.
14의 해시 인덱스는 4, 99의 해시 인덱스는 9이다.
이렇게 나머지 연산을 통해 해시 인덱스를 구하고, 해시 인덱스를 배열의 인덱스로 사용해보자.

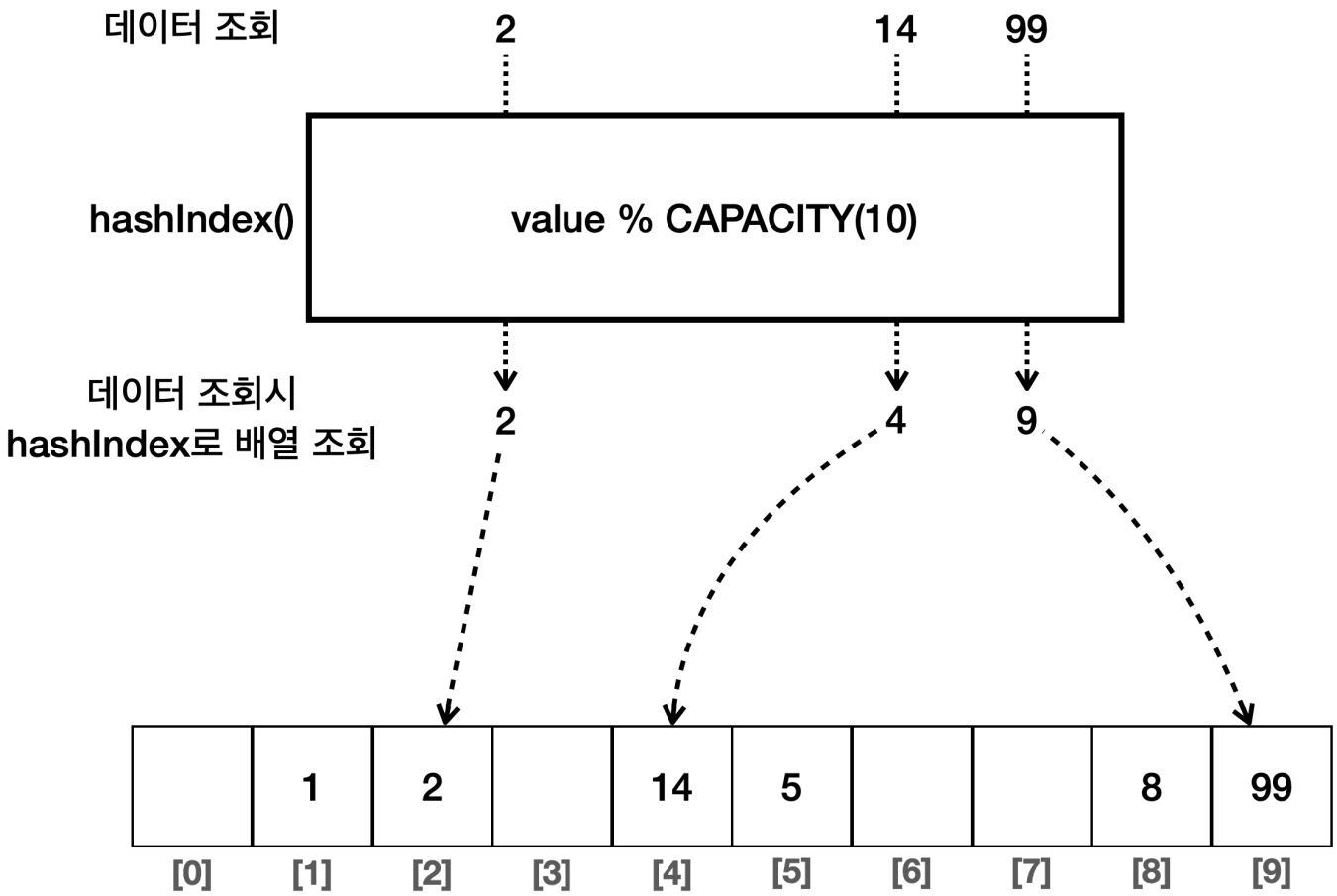
해시 인덱스와 데이터 저장

1, 2, 5, 8, 14 ,99의 값을 크기가 10인 배열에 저장해보자.



- 저장할 값에 나머지 연산자를 사용해서 해시 인덱스를 구한다.
  - `1 % 10 = 1`
  - `2 % 10 = 2`
  - `5 % 10 = 5`
  - `8 % 10 = 8`
  - `14 % 10 = 4`
  - `99 % 10 = 9`
- 해시 인덱스를 배열의 인덱스로 사용해서 데이터를 저장한다.
  - 예) `inputArray[hashIndex] = value`
  - 인덱스만 해시 인덱스를 사용하고, 값은 원래 값을 저장한다.
- 배열의 인덱스를 사용하기 때문에 하나의 값을 저장하는데 O(1)로 빠른 성능을 제공한다.
  - 해시 인덱스 생성 O(1) + 해시 인덱스를 사용해 배열에 값 저장O(1) ➜ O(1)

해시 인덱스와 데이터 조회



- 조회할 값에 나머지 연산자를 사용해서 해시 인덱스를 구한다.
  - `2 % 10 = 2`
  - `14 % 10 = 4`
  - `99 % 10 = 9`
- 해시 인덱스를 배열의 인덱스로 사용해서 데이터를 조회한다.
  - 예) `int value = inputArray[hashIndex]`
  - 인덱스만 해시 인덱스를 사용하고, 값은 원래 값을 조회한다.
- 배열의 인덱스를 사용하기 때문에 하나의 값을 찾는데 O(1)로 빠른 성능을 제공한다.
  - 해시 인덱스 생성 O(1) + 해시 인덱스를 사용해 배열에서 값 조회O(1) ➜ O(1)

코드로 구현해보자.

```
package collection.set;

import java.util.Arrays;

public class HashStart4 {

    static final int CAPACITY = 10;

    public static void main(String[] args) {
```

```
        //{1, 2, 5, 8, 14, 99}
        System.out.println("hashIndex(1) = " + hashIndex(1));
        System.out.println("hashIndex(2) = " + hashIndex(2));
        System.out.println("hashIndex(5) = " + hashIndex(5));
        System.out.println("hashIndex(8) = " + hashIndex(8));
        System.out.println("hashIndex(14) = " + hashIndex(14));
        System.out.println("hashIndex(99) = " + hashIndex(99));

        Integer[] inputArray = new Integer[CAPACITY];
        add(inputArray, 1);
        add(inputArray, 2);
        add(inputArray, 5);
        add(inputArray, 8);
        add(inputArray, 14);
        add(inputArray, 99);
        System.out.println("inputArray = " + Arrays.toString(inputArray));

        //검색
        int searchValue = 14;
        int hashIndex = hashIndex(searchValue);
        System.out.println("searchValue hashIndex = " + hashIndex);
        Integer result = inputArray[hashIndex]; // O(1)
        System.out.println(result);
    }

    private static void add(Integer[] inputArray, int value) {
        int hashIndex = hashIndex(value);
        inputArray[hashIndex] = value;
    }

    static int hashIndex(int value) {
        return value % CAPACITY;
    }
}
```

실행 결과

```
hashIndex(1) = 1
hashIndex(2) = 2
hashIndex(5) = 5
hashIndex(8) = 8
hashIndex(14) = 4
```

```
hashIndex(99) = 9
inputArray = [null, 1, 2, null, 14, 5, null, null, 8, 99]
searchValue hashIndex = 4
14
```

hashIndex()

- 해시 인덱스를 반환한다.
- 해시 인덱스는 입력 값을 계산해서 인덱스로 사용하는 것을 뜻한다. 여기서는 입력 값을 배열의 크기로 나머지 연산해서 구한다.

add()

- 해시 인덱스를 먼저 구한다.
- 구한 해시 인덱스의 위치에 데이터를 저장한다.

조회

- 해시 인덱스를 구하고, 배열에 해시 인덱스를 대입해서 값을 조회한다.
- `inputArray[hashIndex]`

정리

- 입력 값의 범위가 넓어도 실제 모든 값이 들어오지는 않기 때문에 배열의 크기를 제한하고, 나머지 연산을 통해 메모리가 낭비되는 문제도 해결할 수 있다.
- 해시 인덱스를 사용해서 O(1)의 성능으로 데이터를 저장하고, O(1)의 성능으로 데이터를 조회할 수 있게 되었다. 덕분에 자료 구조의 조회 속도를 비약적으로 향상할 수 있게 되었다.

한계 - 해시 충돌

그런데 지금까지 설명한 내용은 저장할 위치가 충돌할 수 있다는 한계가 있다.

예를 들어 1, 11의 두 값은 이렇게 10으로 나누면 같은 값 1이 된다. 둘다 같은 해시 인덱스가 나와버리는 것이다.

- `1 % 10 = 1`
- `11 % 10 = 1`

다음의 경우도 마찬가지이다.

- `99 % 10 = 9`
- `9 % 10 = 9`

다음시간에는 해시 충돌에 대해 더 알아보고 어떻게 해시 충돌 문제를 해결할 수 있는지 알아보자.

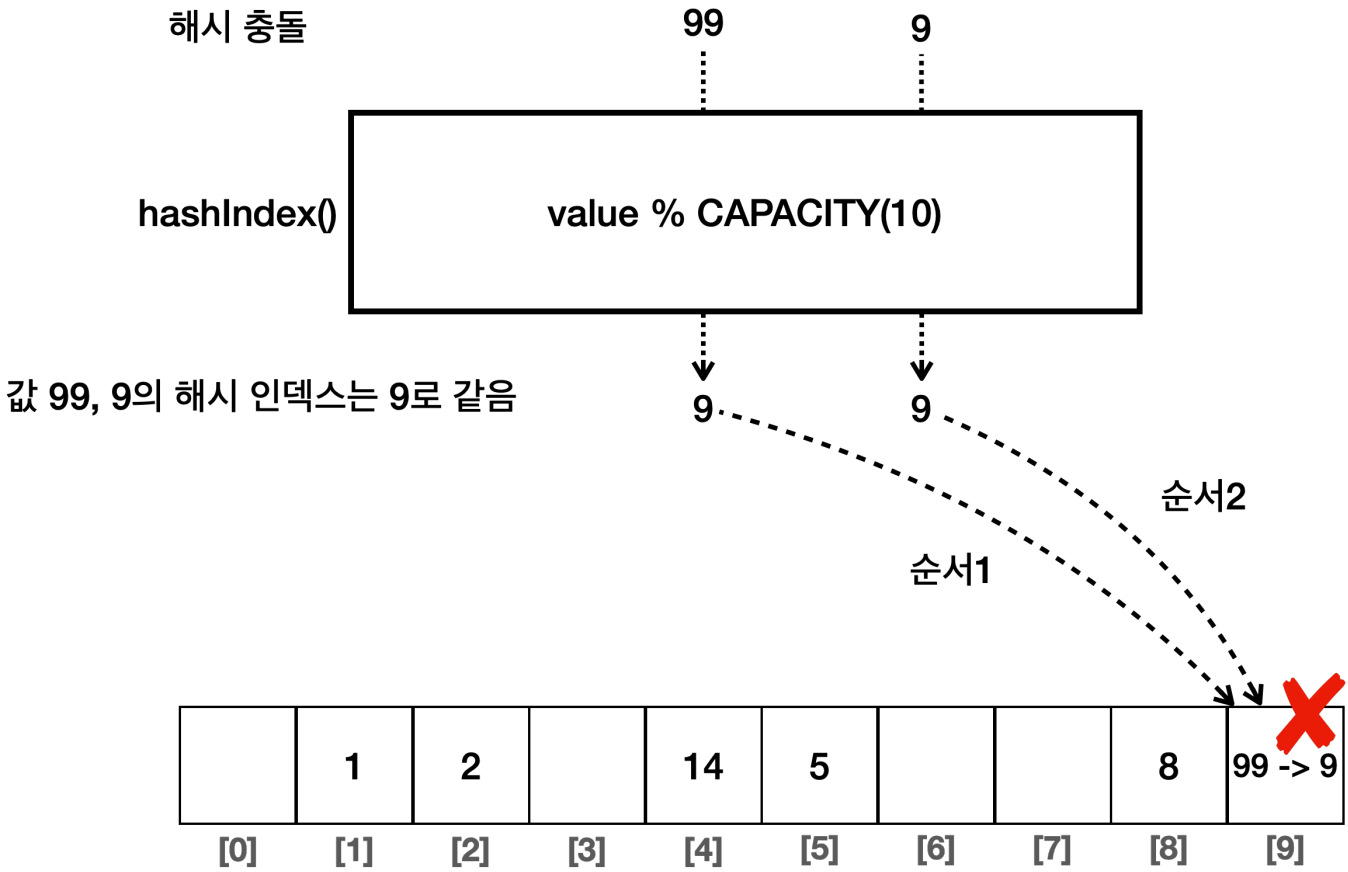
해시 알고리즘5 - 해시 충돌 설명

해시 충돌

99, 9의 두 값은 10으로 나누면 9가 된다. 따라서 다른 값을 입력했지만 같은 해시 코드가 나오게 되는데 이것을 해시 충돌이라 한다.

- `99 % 10 = 9`
- `9 % 10 = 9`

해시 충돌이 발생하면 어떤 문제가 나타나는지 알아보자.



- 먼저 99의 값을 저장한다. 해시 인덱스는 9이므로 9번 인덱스에 99 값을 저장한다.
- 다음으로 9의 값을 저장한다. 해시 인덱스는 9이므로 9번 인덱스에 9 값을 저장한다.
- 결과적으로 배열의 인덱스 9에는 처음에 저장한 값 99는 사라지고, 마지막에 저장한 값 9만 남게된다.

이 문제를 해결하는 가장 단순한 방법은 CAPACITY를 값의 입력 범위만큼 키우면 된다. 여기서는 99까지만 입력하므로 CAPACITY를 100으로 늘리면 된다. 그러면 충돌이 발생하지 않는다. 하지만 앞서 보았듯이 이 방법은 메모리 낭비가 심하고, 모든 `int` 숫자를 다 받는 문제를 해결할 수 없다.

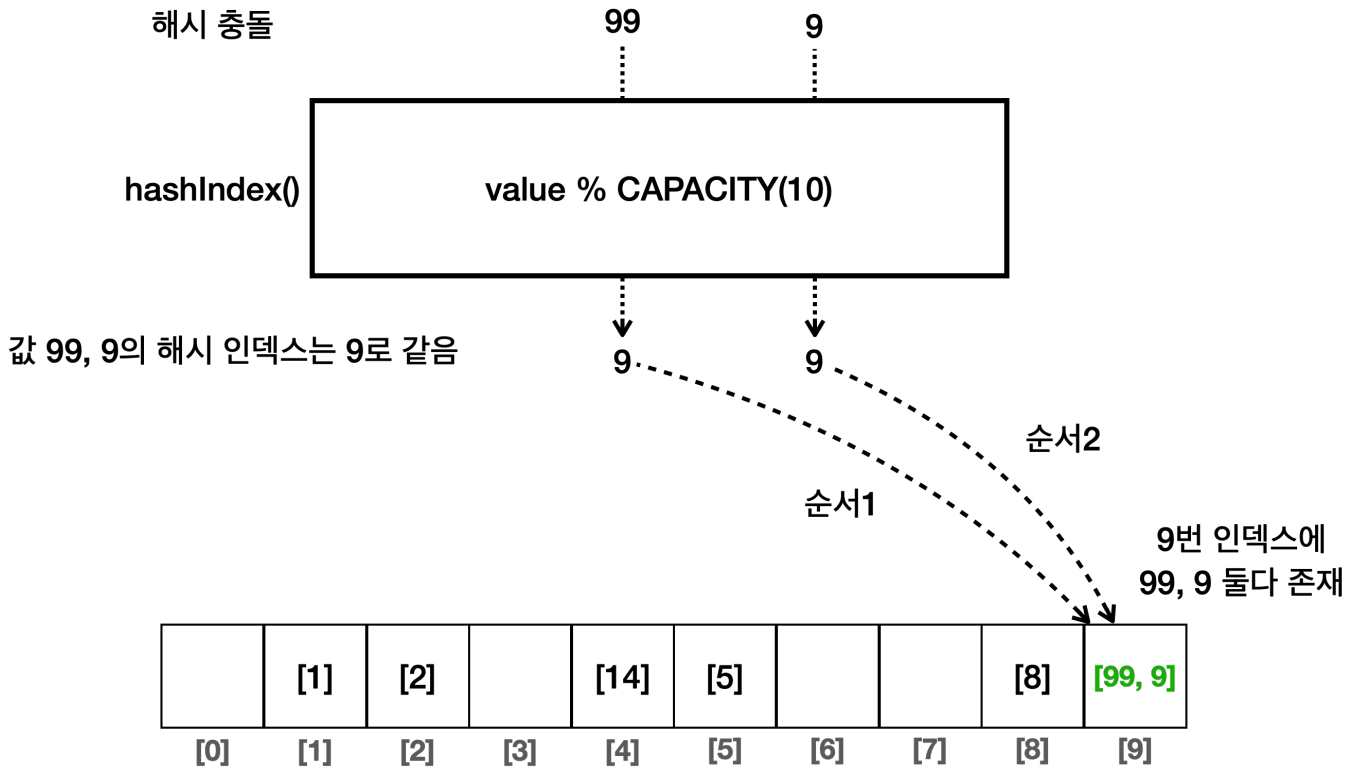
해시 충돌 해결

해시 충돌을 인정하면 문제 해결의 실마리가 보인다.

해시 충돌은 낮은 확률로 일어날 수 있다고 가정하는 것이다.

해결 방안은 바로 해시 충돌이 일어났을 때 단순하게 같은 해시 인덱스의 값을 같은 인덱스에 함께 저장해버리는 것이다.

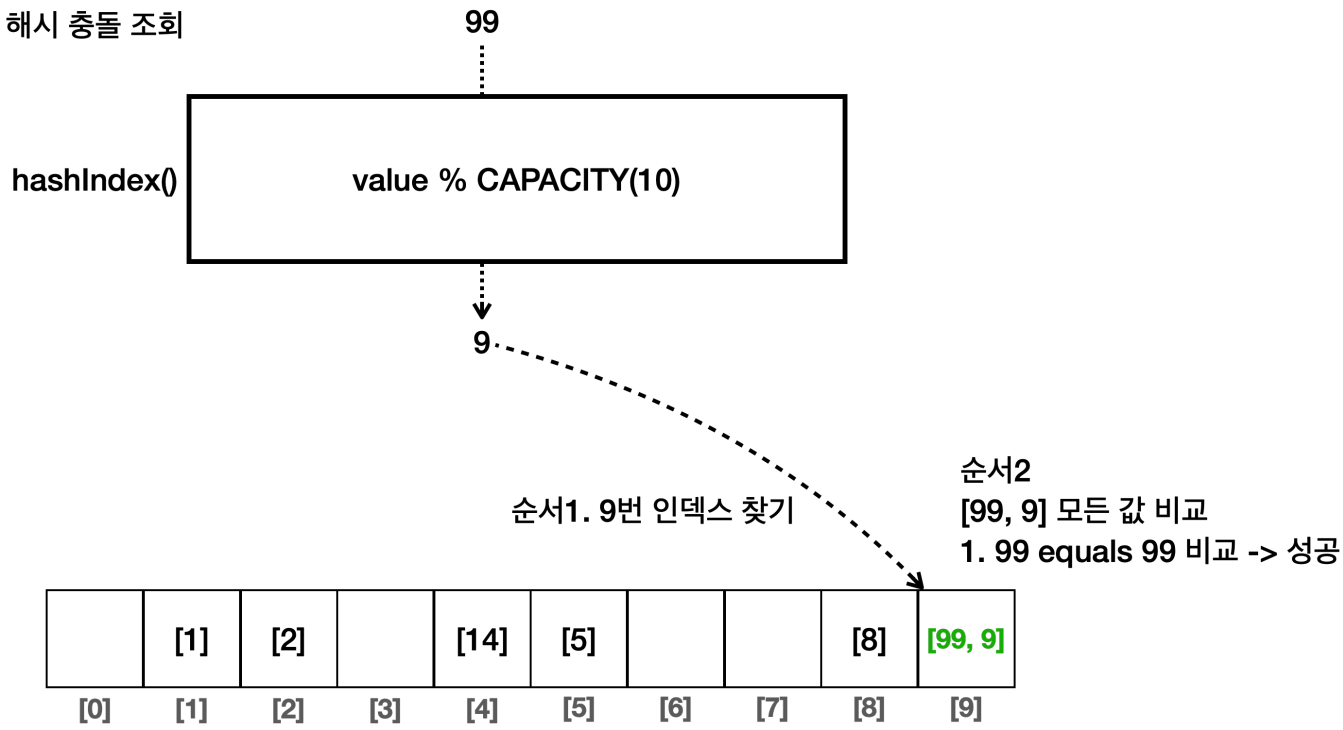
해시 충돌과 저장



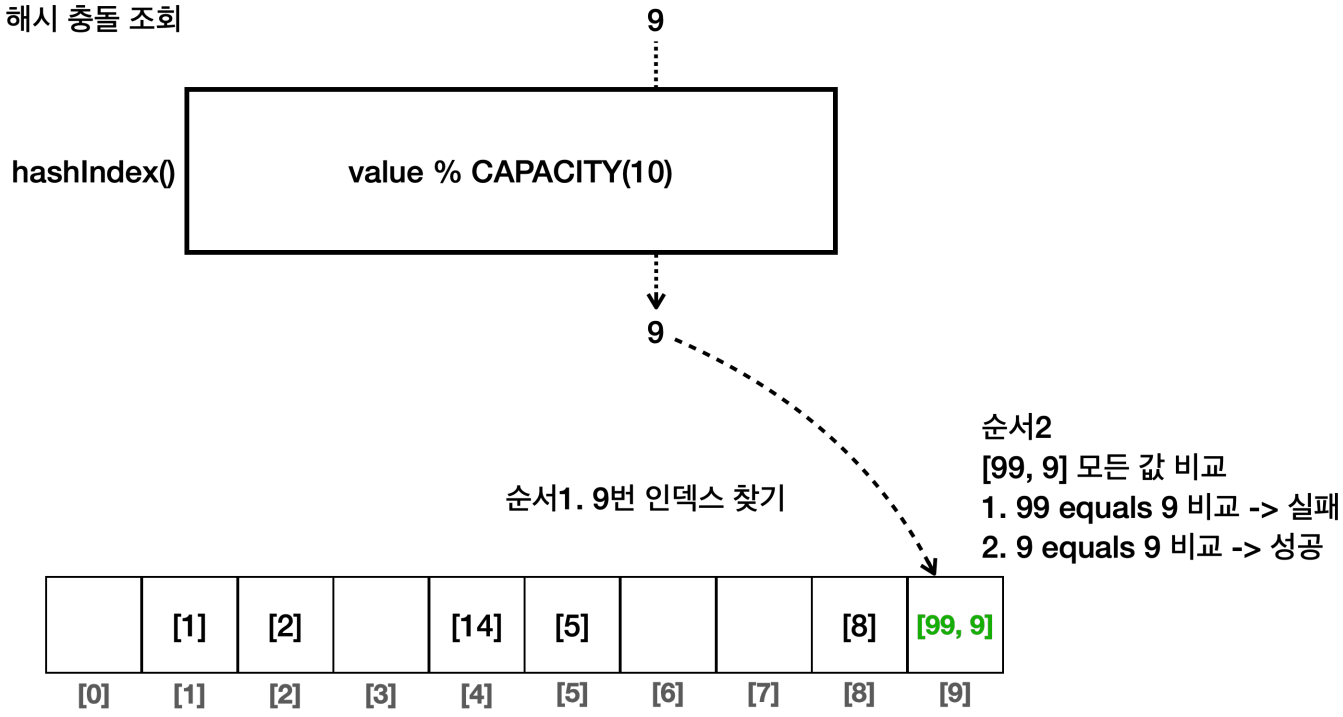
물론 여러 데이터를 배열의 하나의 공간에 함께 저장할 수는 없다. 대신에 **배열 안에 배열을 만들면 된다**. 물론 배열 안에 리스트 같은 다른 자료구조를 사용해도 된다.

해시 충돌과 조회

해시 충돌이 난 경우 내부의 데이터를 하나씩 비교해보면 원하는 결과를 찾을 수 있다. 예를 들어 99를 조회한다고 가정해보자.

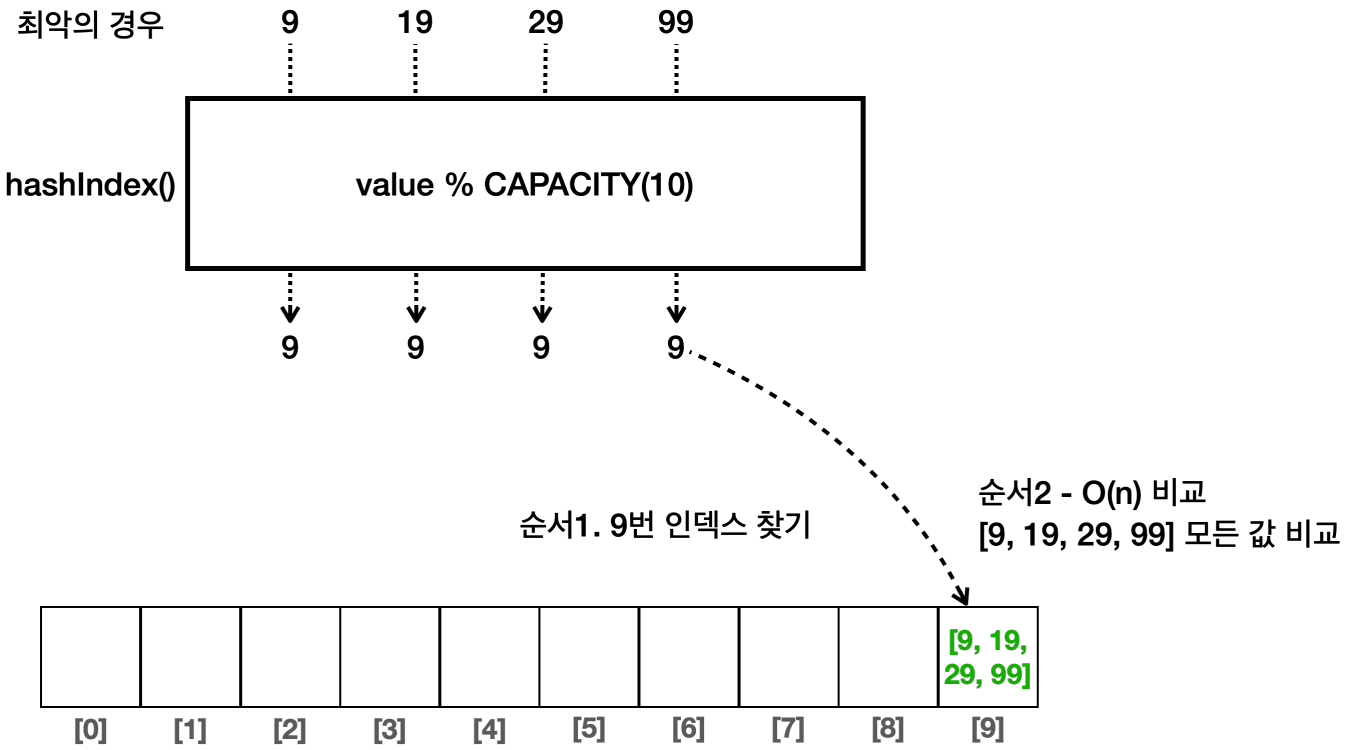


- 99의 해시 인덱스는 9이다. 배열에서 9번 인덱스를 찾는다.
- 배열 안에는 또 배열이 들어있다. 여기에 있는 모든 값을 검색할 값과 하나씩 비교한다.
  - [99, 9]의 데이터가 들어있는데, 첫 비교해서 원하는 데이터를 찾을 수 있다.



- 9의 해시 인덱스는 9이다. 배열에서 9번 인덱스를 찾는다.
- 배열 안에는 또 배열이 들어있다. 여기에 있는 모든 값을 검색할 값과 하나씩 비교한다.
  - [99, 9]의 데이터가 들어있다. 첫 비교에서 `99 equals 9`는 거짓이므로 실패한다. 다음 비교에서 `9 equals 9`이므로 원하는 데이터를 찾았다.
  - 비교시 `equals`를 사용했지만 기본형이라면 물론 `==`을 사용해도 된다.

최악의 경우



- 값을 9, 19, 29, 99만 저장한다고 가정해보자. 이 경우 모든 해시 인덱스가 9가 된다.
- 따라서 9번 인덱스에 데이터가 모두 저장된다.
- 이렇게 되면 데이터를 찾을 때 결국 9번에 가서 저장한 데이터의 수 만큼 값을 반복해서 비교해야 한다.
- 따라서 최악의 경우 O(n)의 성능을 보인다.

정리

해시 인덱스를 사용하는 방식은 최악의 경우 O(n)의 성능을 보인다. 하지만 확률적으로 보면 어느 정도 넓게 퍼지기 때문에 평균으로 보면 대부분 O(1)의 성능을 제공한다. 해시 충돌이 가끔 발생해도 내부에서 값을 몇 번만 비교하는 수준이기 때문에 대부분의 경우 매우 빠르게 값을 찾을 수 있다.

해시 알고리즘6 - 해시 충돌 구현

해시 충돌 상황까지 고려해서 코드를 구현해보자.

```
package collection.set;
```

```java
import java.util.Arrays;
import java.util.LinkedList;

public class HashStart5 {

    static final int CAPACITY = 10;

    public static void main(String[] args) {
        //{1, 2, 5, 8, 14, 99 ,9}
        LinkedList<Integer>[] buckets = new LinkedList[CAPACITY];
        for (int i = 0; i < CAPACITY; i++) {
            buckets[i] = new LinkedList<>();
        }

        add(buckets, 1);
        add(buckets, 2);
        add(buckets, 5);
        add(buckets, 8);
        add(buckets, 14);
        add(buckets, 99);
        add(buckets, 9); //중복
        System.out.println("buckets = " + Arrays.toString(buckets));

        //검색
        int searchValue = 9;
        boolean contains = contains(buckets, searchValue);
        System.out.println("buckets.contains(" + searchValue + ") = " +
contains);
    }

    private static void add(LinkedList<Integer>[] buckets, int value) {
        int hashIndex = hashIndex(value);
        LinkedList<Integer> bucket = buckets[hashIndex]; // O(1)
        if (!bucket.contains(value)) { // O(n)
            bucket.add(value);
        }
    }

    private static boolean contains(LinkedList<Integer>[] buckets, int
searchValue) {
        int hashIndex = hashIndex(searchValue);
        LinkedList<Integer> bucket = buckets[hashIndex]; // O(1)
        return bucket.contains(searchValue); // O(n)
```

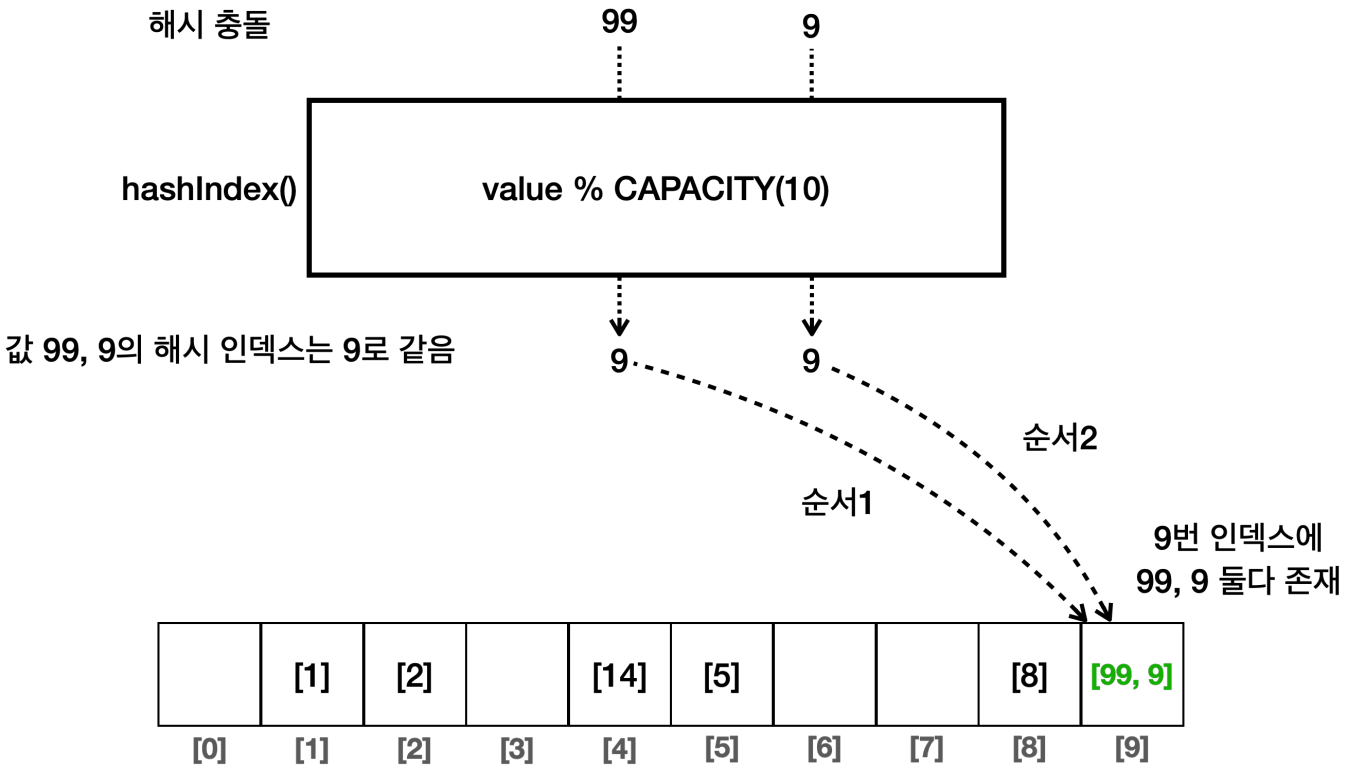
```
    }

    static int hashIndex(int value) {
        return value % CAPACITY;
    }
}
```

실행 결과

```
buckets = [[], [1], [2], [], [14], [5], [], [], [8], [99, 9]]
bucket.contains(9) = true
```

결과는 그림과 같다.



배열 선언

```
LinkedList<Integer>[] buckets = new LinkedList[CAPACITY]
```

먼저 배열의 이름을 `buckets` 즉 바구니들이라고 지었다. 배열 안에 단순히 값이 들어가는 것이 아니라, 해시 충돌을 고려해서 배열 안에 또 배열이 들어가야 한다. 그래야 해시 충돌이 발생했을 때 여러 값을 담을 수 있다.
여기서는 배열 안에 배열 대신에 편리하게 사용할 수 있는 연결 리스트를 사용했다. `LinkedList`는 하나의 바구니이다. 이런 바구니를 여러개 모아서 배열을 선언했다. 즉 배열 안에 연결 리스트가 들어있고, 연결 리스트 안에 데이터가

들어가는 구조이다.
쉽게 이야기해서 바구니들(배열) 안에 각각의 바구니(연결 리스트)가 있고, 바구니(연결 리스트) 안에 데이터가 들어가는 구조이다.

데이터 등록

```
private static void add(LinkedList<Integer>[] buckets, int value) {
    int hashIndex = hashIndex(value);
    LinkedList<Integer> bucket = buckets[hashIndex]; //O(1)
    if (!bucket.contains(value)) { //O(n)
        bucket.add(value);
    }
}
```

- 데이터를 등록할 때 먼저 해시 인덱스(hashIndex)를 구한다.
- 해시 인덱스로 배열의 인덱스를 찾는다. 배열에는 연결 리스트가 들어있다.
  - 해시 인덱스를 통해 바구니들 사이에서 바구니인 연결 리스트를 하나 찾은 것이다.
- 셋은 중복된 값을 저장하지 않는다. 따라서 바구니에 값을 저장하기 전에 `contains()`를 사용해서 중복 여부를 확인한다. 만약 바구니에 같은 데이터가 없다면 데이터를 저장한다.
  - 연결 리스트의 `contains()`는 모든 항목을 다 순회하기 때문에 O(n)의 성능이다.
  - 하지만 해시 충돌이 발생하지 않으면 데이터가 1개만 들어있기 때문에 O(1)의 성능을 제공한다.

데이터 검색

```
private static boolean contains(LinkedList<Integer>[] buckets, int
searchValue) {
    int hashIndex = hashIndex(searchValue);
    LinkedList<Integer> bucket = buckets[hashIndex]; //O(1)
    return bucket.contains(searchValue); //O(n)
}
```

- 해시 인덱스로 배열의 인덱스를 찾는다. 여기에는 연결 리스트가 들어있다.
- 연결 리스트의 `bucket.contains(searchValue)` 메서드를 사용해서 찾는 데이터가 있는지 확인한다.
  - 연결 리스트의 `contains()`는 모든 항목을 다 순회하기 때문에 O(n)의 성능이다.
  - 하지만 해시 충돌이 발생하지 않으면 데이터가 1개만 들어있기 때문에 O(1)의 성능을 제공한다.

해시 인덱스 충돌 확률

해시 충돌이 발생하면 데이터를 추가하거나 조회할 때, 연결 리스트 내부에서 O(n)의 추가 연산을 해야 하므로 성능이

떨어진다. 따라서 해시 충돌은 가급적 발생하지 않도록 해야 한다.
해시 충돌이 발생할 확률은 입력하는 데이터의 수와 배열의 크기와 관련이 있다. 입력하는 데이터의 수와 비교해서 배열의 크기가 클 수록 충돌 확률은 낮아진다.
배열의 크기인 `CAPACITY` 값을 변경하면서 실행해 보자.

CAPACITY = 1: `[[1, 2, 5, 8, 14, 99, 9]]`
CAPACITY = 5: `[[5], [1], [2], [8], [14, 99, 9]]`
CAPACITY = 10: `[[], [1], [2], [], [14], [5], [], [], [8], [99, 9]]`
CAPACITY = 11: `[[99], [1], [2], [14], [], [5], [], [], [8], [9], []]`
CAPACITY = 15: `[[], [1], [2], [], [], [5], [], [], [8], [99, 9], [], [], [], [], [14]]`

- **CAPACITY = 1**: 배열의 크기가 하나밖에 없을 때는 모든 해시가 충돌한다.
- **CAPACITY = 5**: 배열의 크기가 입력하는 데이터 수 보다 작은 경우 해시 충돌이 자주 발생한다.
- **CAPACITY = 10**: 저장할 데이터가 7개 인데, 배열의 크기는 10이다. **7/10 약 70% 정도**로 약간 여유있게 데이터가 저장된다. 이 경우 가끔 충돌이 발생한다.
- **CAPACITY = 11**: 저장할 데이터가 7개 인데, 배열의 크기는 11이다. 가끔 충돌이 발생한다. 여기서는 충돌이 발생하지 않았다.
- **CAPACITY = 15**: 저장할 데이터가 7개 인데, 배열의 크기는 15이다. 가끔 충돌이 발생한다. 여기서는 충돌이 하나 발생했다.

아주 간단한 예제로 알아보았지만, 통계적으로 입력한 데이터의 수가 배열의 크기를 75% 넘지 않으면 해시 인덱스는 자주 충돌하지 않는다. 반대로 75%를 넘으면 자주 충돌하기 시작한다.
배열의 크기를 크게 만들면 해시 충돌은 줄어서 성능은 좋아지지만, 많은 메모리가 낭비된다. 반대로 배열의 크기를 너무 작게 만들면 해시가 자주 충돌해서 성능이 나빠진다. 상황에 따라 다르겠지만 보통 75%를 적절한 크기로 보고 기준으로 잡는 것이 효과적이다.

정리
해시 인덱스를 사용하는 경우

- 데이터 저장
  - 평균: O(1)
  - 최악: O(n)
- 데이터 조회
  - 평균: O(1)
  - 최악: O(n)

해시 인덱스를 사용하는 방식은 사실 최악의 경우는 거의 발생하지 않는다. 배열의 크기만 적절하게 잡아주면 대부분

O(1)에 가까운 매우 빠른 성능을 보여준다.

이제 O(n)을 O(1)로 바꿀 수 있는 매우 효율적인 해시 알고리즘에 대해서 배웠다.
지금까지 설명한 내용을 바탕으로 `MyHashSetV0`를 개선해보자.