

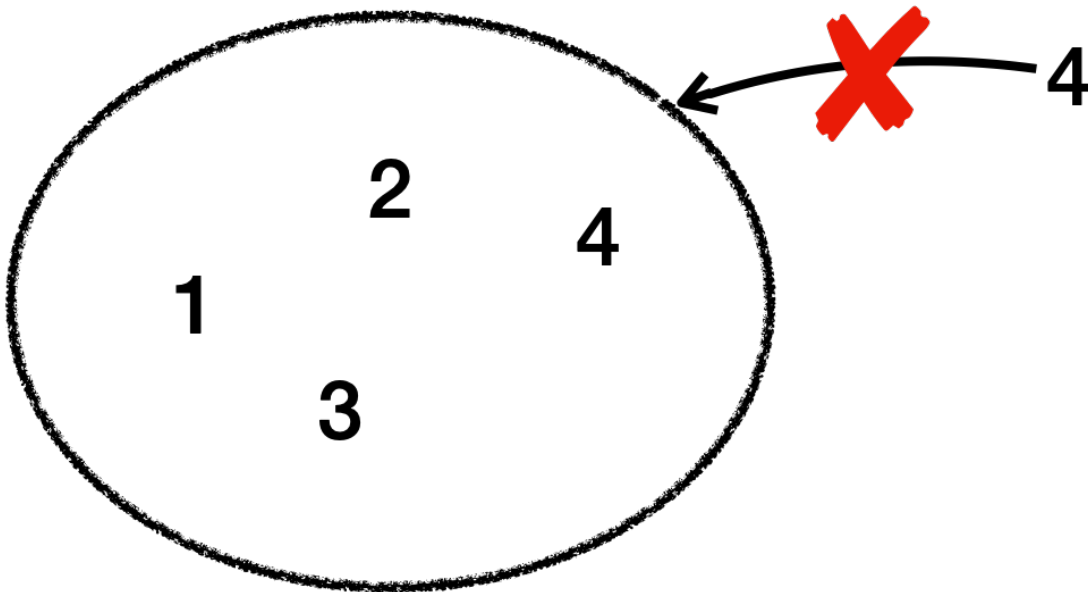
## 8. 컬렉션 프레임워크 - Set

#1.인강/0.자바/4.자바-중급2편

- /자바가 제공하는 Set1 - HashSet, LinkedHashSet
- /자바가 제공하는 Set2 - TreeSet
- /자바가 제공하는 Set3 - 예제
- /자바가 제공하는 Set4 - 최적화
- /문제와 풀이1
- /문제와 풀이2
- /정리

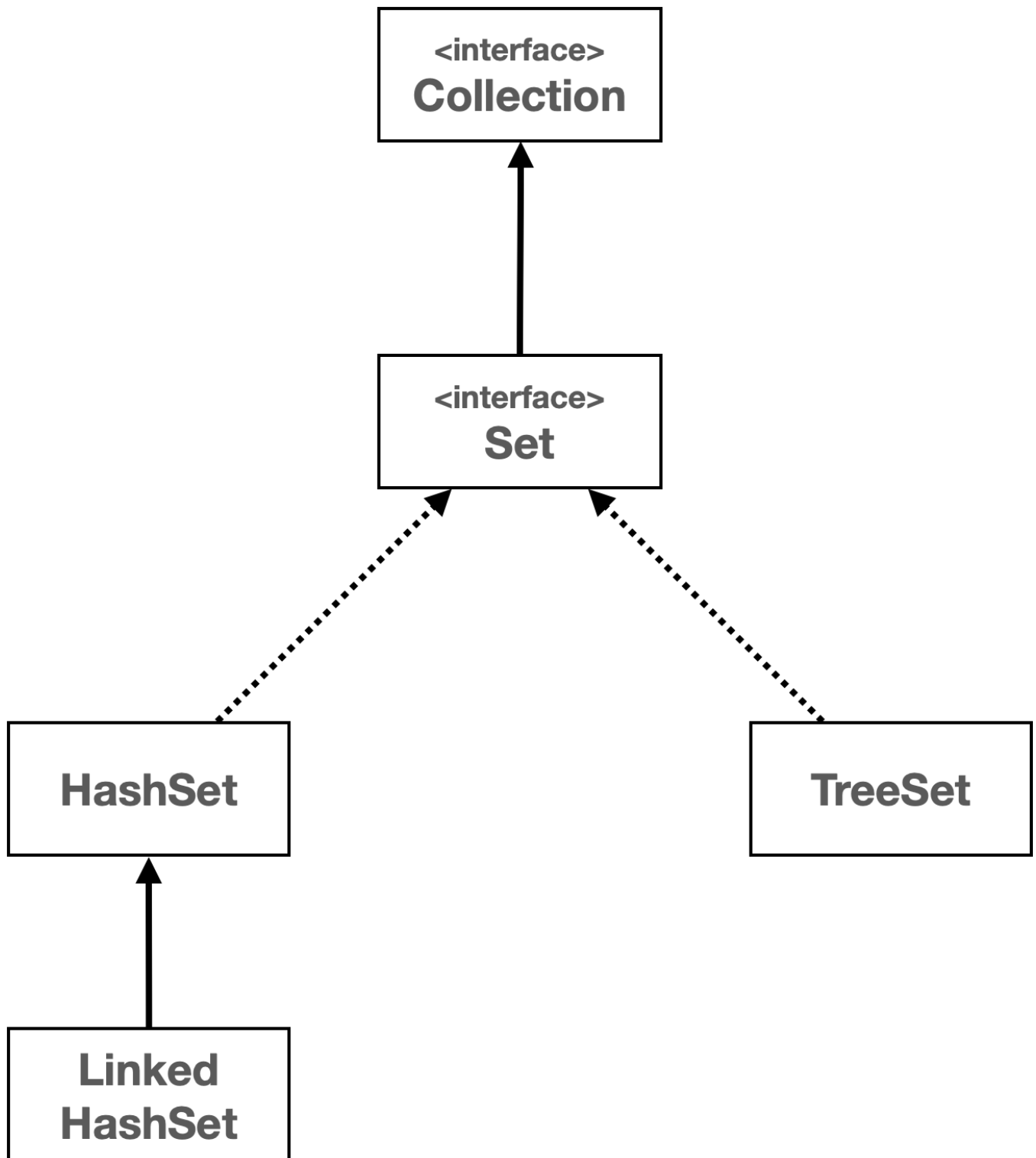
### 자바가 제공하는 Set1 - HashSet, LinkedHashSet

Set (세트, 셋) 자료 구조



셋은 중복을 허용하지 않고, 순서를 보장하지 않는 자료 구조이다.

컬렉션 프레임워크 - Set



### Collection 인터페이스

**Collection** 인터페이스는 `java.util` 패키지의 컬렉션 프레임워크의 핵심 인터페이스 중 하나이다. 이 인터페이스는 자바에서 다양한 컬렉션, 즉 데이터 그룹을 다루기 위한 메서드를 정의한다. **Collection** 인터페이스는 **List**, **Set**, **Queue**와 같은 다양한 하위 인터페이스와 함께 사용되며, 이를 통해 데이터를 리스트, 세트, 큐 등의 형태로 관리할 수 있다. 자세한 내용은 뒤에서 다룬다.

### Set 인터페이스

자바의 Set 인터페이스는 java.util 패키지의 컬렉션 프레임워크에 속하는 인터페이스 중 하나이다. Set 인터페이스는 중복을 허용하지 않는 유일한 요소의 집합을 나타낸다. 즉, 어떤 요소도 같은 Set 내에 두 번 이상 나타날 수 없다. Set은 수학적 집합 개념을 구현한 것으로, 순서를 보장하지 않으며, 특정 요소가 집합에 있는지 여부를 확인하는데 최적화되어 있다.

Set 인터페이스는 HashSet, LinkedHashSet, TreeSet 등의 여러 구현 클래스를 가지고 있으며, 각 클래스는 Set 인터페이스를 구현하며 각각의 특성을 가지고 있다.

### Set 인터페이스의 주요 메서드

메서드	설명
add(E e)	지정된 요소를 세트에 추가한다(이미 존재하는 경우 추가하지 않음).
addAll(Collection<? extends E> c)	지정된 컬렉션의 모든 요소를 세트에 추가한다.
contains(Object o)	세트가 지정된 요소를 포함하고 있는지 여부를 반환한다.
containsAll(Collection<?> c)	세트가 지정된 컬렉션의 모든 요소를 포함하고 있는지 여부를 반환한다.
remove(Object o)	지정된 요소를 세트에서 제거한다.
removeAll(Collection<?> c)	지정된 컬렉션에 포함된 요소를 세트에서 모두 제거한다.
retainAll(Collection<?> c)	지정된 컬렉션에 포함된 요소만을 유지하고 나머지 요소는 세트에서 제거한다.
clear()	세트에서 모든 요소를 제거한다.
size()	세트에 있는 요소의 수를 반환한다.
isEmpty()	세트가 비어 있는지 여부를 반환한다.
iterator()	세트의 요소에 대한 반복자를 반환한다.
toArray()	세트의 모든 요소를 배열로 반환한다.
toArray(T[] a)	세트의 모든 요소를 지정된 배열로 반환한다.

지금부터 Set의 주요 구현체를 하나씩 알아보자.

- HashSet

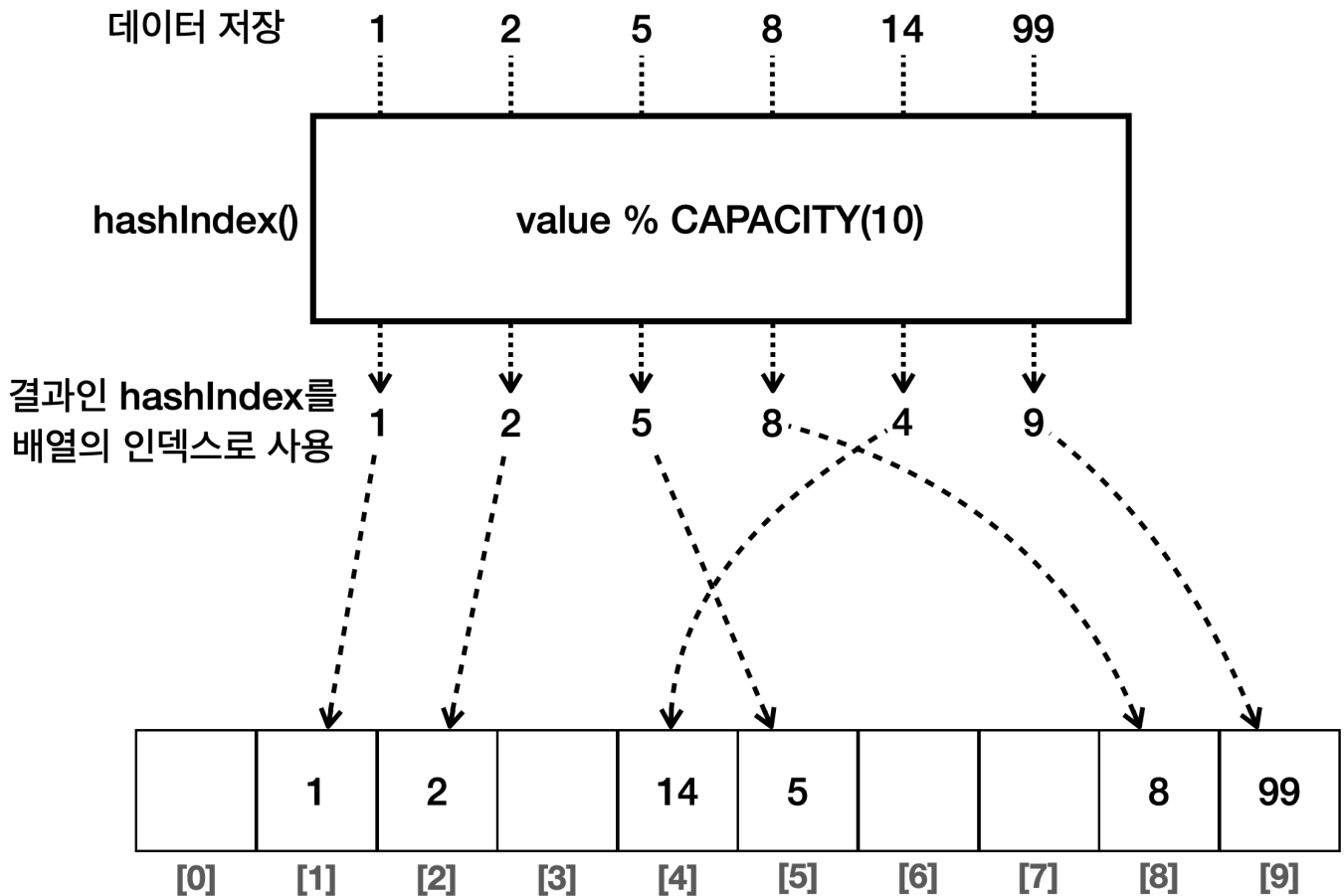
- `LinkedHashSet`
- `TreeSet`

## 1. HashSet

- **구현:** 해시 자료 구조를 사용해서 요소를 저장한다.
- **순서:** 요소들은 특정한 순서 없이 저장된다. 즉, 요소를 추가한 순서를 보장하지 않는다.
- **시간 복잡도:** `HashSet` 의 주요 연산(추가, 삭제, 검색)은 평균적으로  $O(1)$  시간 복잡도를 가진다.
- **용도:** 데이터의 유일성만 중요하고, 순서가 중요하지 않은 경우에 적합하다.

앞서 우리가 구현한 `MyHashSet` 이 바로 `HashSet` 이다.

### HashSet 구현



- 그림을 단순화 했지만, `hashCode()`, `equals()` 를 모두 사용한다.

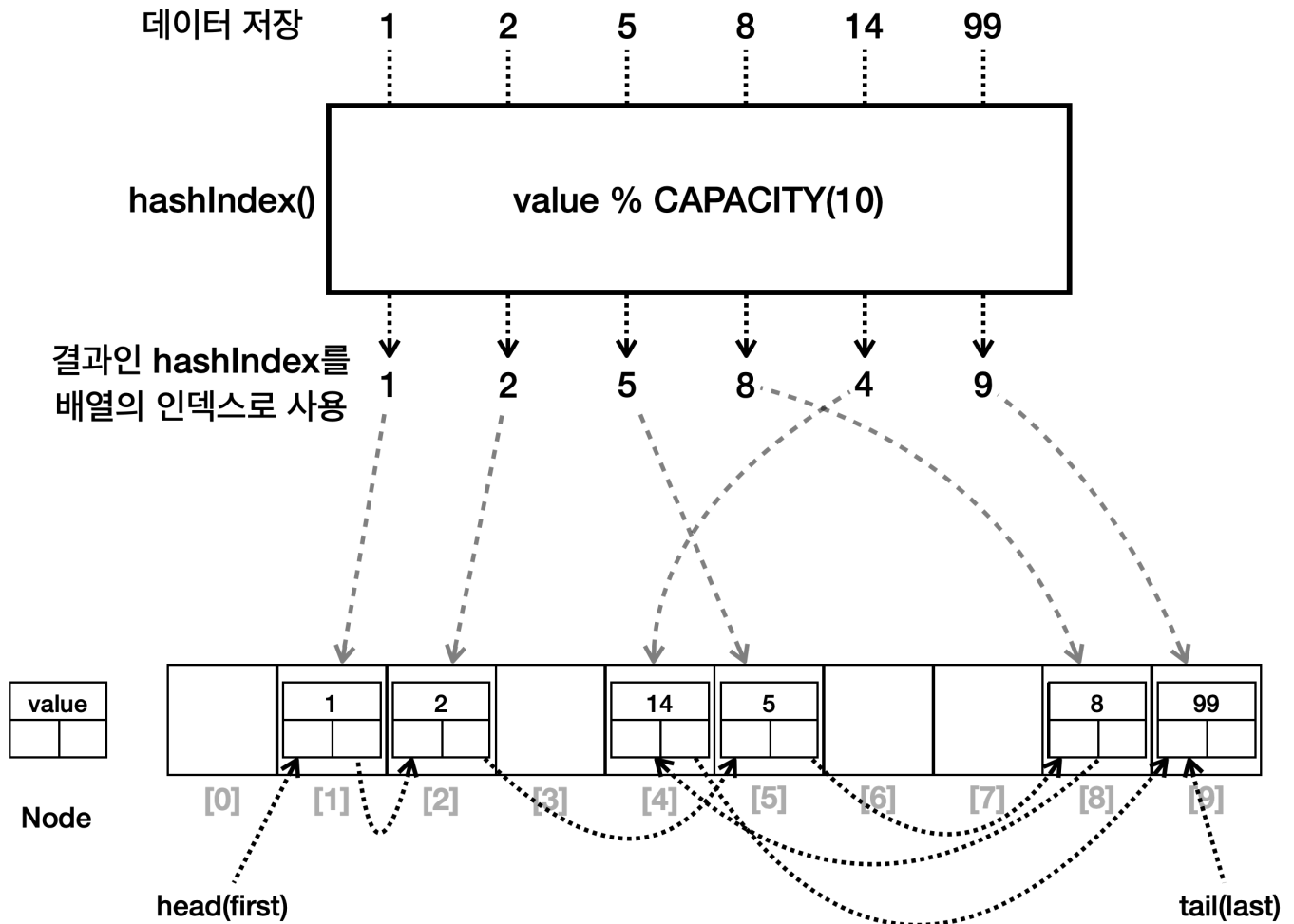
## 2. LinkedHashSet

- **구현:** `LinkedHashSet` 은 `HashSet` 에 연결 리스트를 추가해서 요소들의 순서를 유지한다.
- **순서:** 요소들은 추가된 순서대로 유지된다. 즉, 순서대로 조회 시 요소들이 추가된 순서대로 반환된다.
- **시간 복잡도:** `LinkedHashSet` 도 `HashSet` 과 마찬가지로 주요 연산에 대해 평균  $O(1)$  시간 복잡도를 가진다.

다.

- **용도:** 데이터의 유일성과 함께 삽입 순서를 유지해야 할 때 적합하다.
- **참고:** 연결 링크를 유지해야 하기 때문에 HashSet 보다는 조금 더 무겁다.

### LinkedHashSet 구현



- LinkedHashSet은 HashSet에 연결 링크만 추가한 것이다.
- HashSet에 LinkedList를 합친 것으로 이해하면 된다.
- 이 연결 링크는 데이터를 입력한 순서대로 연결된다.
  - head(first) 부터 순서대로 링크를 따라가면 입력 순서대로 데이터를 순회할 수 있다.
  - 양방향으로 연결된다. (그림에서는 이해를 돕기 위해 화살표는 다음 순서로만 보여주었다. 실제로는 양방향이다.)
- 여기서는 1, 2, 5, 8, 14, 99 순서대로 입력된다. 링크를 보면 1, 2, 5, 8, 14, 99 순서로 연결 되어 있는 것을 확인할 수 있다.
- 이 링크를 first 부터 순서대로 따라가면서 출력하면 순서대로 출력할 수 있다.

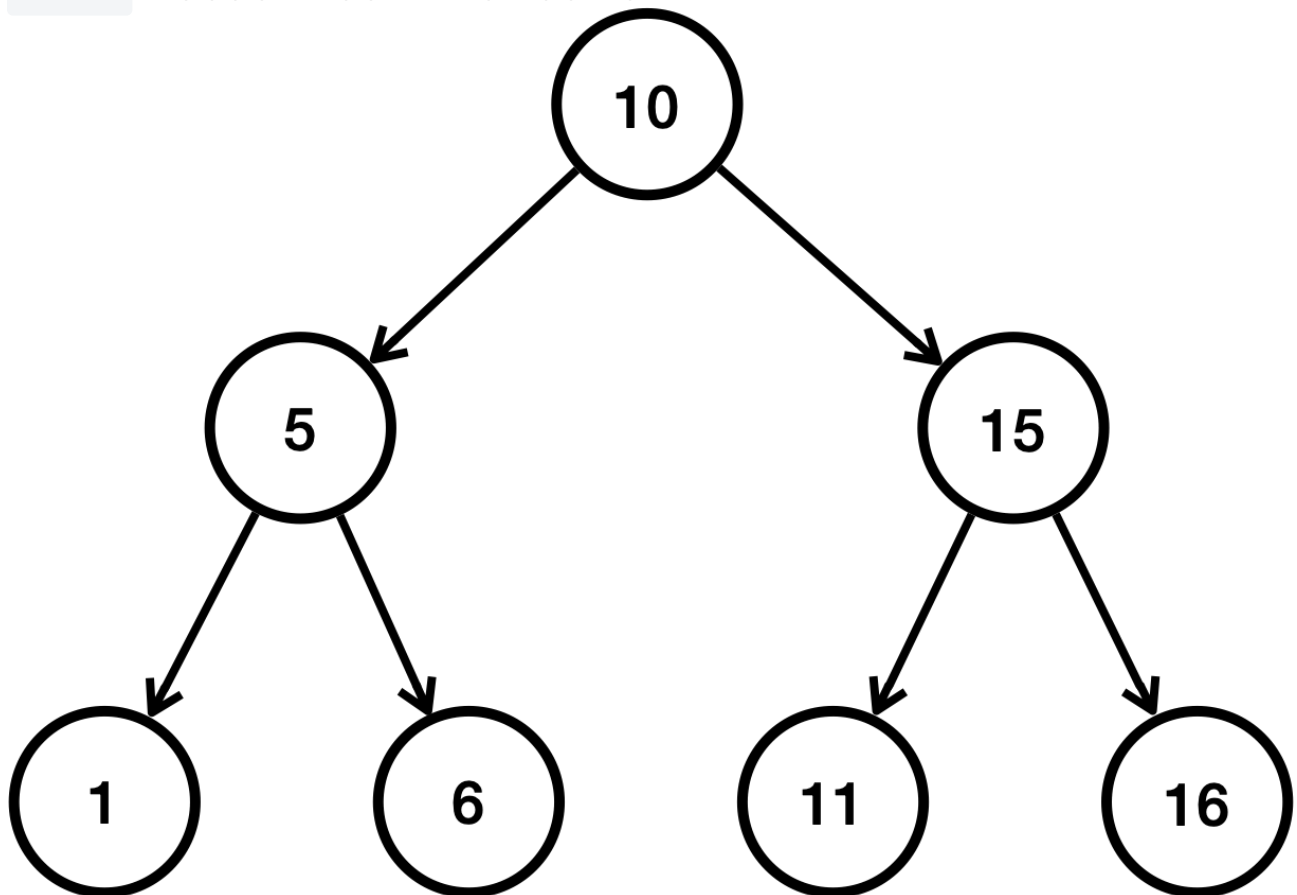
## 자바가 제공하는 Set2 - TreeSet

### 3. TreeSet

- **구현:** TreeSet은 이진 탐색 트리를 개선한 레드-블랙 트리를 내부에서 사용한다.
- **순서:** 요소들은 정렬된 순서로 저장된다. 순서의 기준은 비교자(Comparator)로 변경할 수 있다. 비교자는 뒤에서 다룬다.
- **시간 복잡도:** 주요 연산들은  $O(\log n)$ 의 시간 복잡도를 가진다. 따라서 HashSet 보다는 느리다.
- **용도:** 데이터들을 정렬된 순서로 유지하면서 집합의 특성을 유지해야 할 때 사용한다. 예를 들어, 범위 검색이나 정렬된 데이터가 필요한 경우에 유용하다. 참고로 입력된 순서가 아니라 데이터 값의 순서이다. 예를 들어 3, 1, 2를 순서대로 입력해도 1, 2, 3 순서로 출력된다.

### 트리 구조

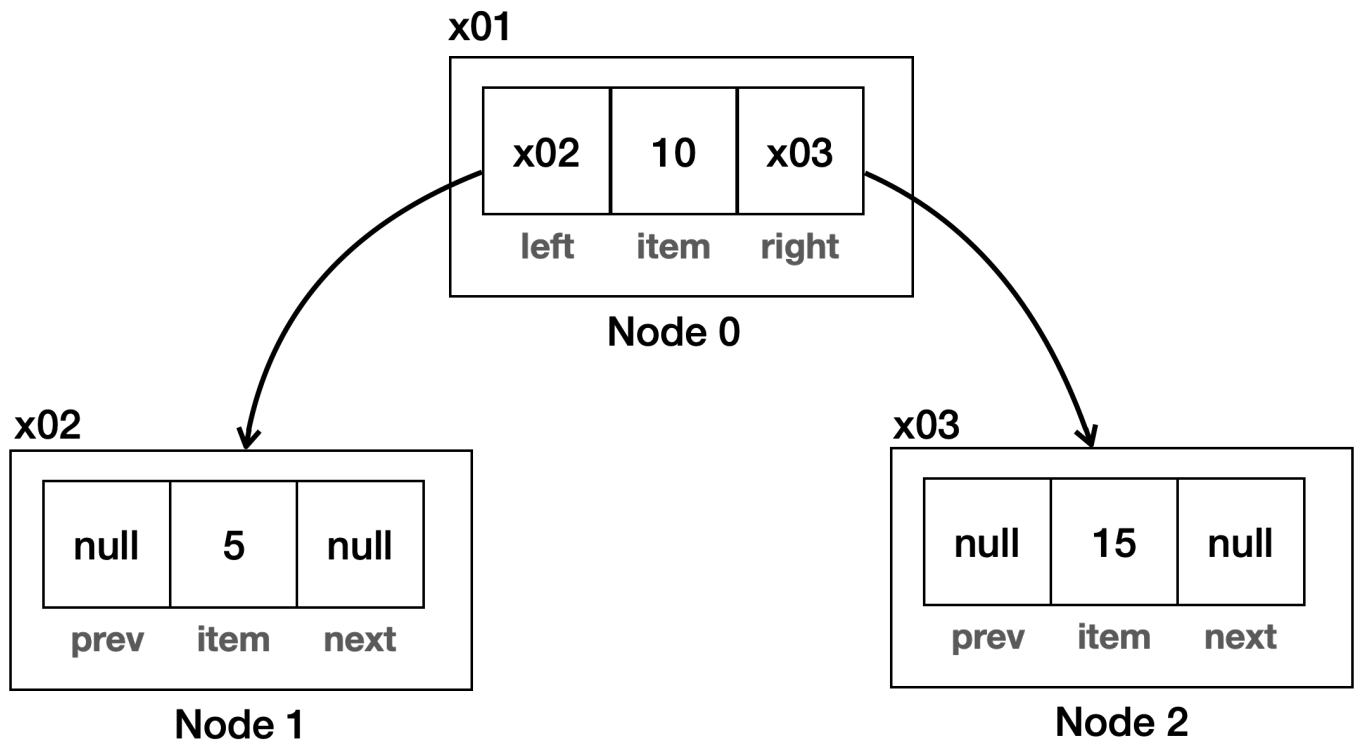
TreeSet을 이해하려면 트리 구조를 먼저 알아야 한다.



- 트리는 부모 노드와 자식 노드로 구성된다.
- 가장 높은 조상을 루트(root)라 한다. 이 그림을 뒤집어보면 왜 트리라고 하는지, 처음을 루트라고 하는지 이해가 될 것이다.
- 자식이 2개까지 올 수 있는 트리를 **이진 트리**라 한다.
- 여기에 노드의 왼쪽 자손은 더 작은 값을 가지고, 오른쪽 자손은 더 큰 값을 가지는 것을 **이진 탐색 트리**라 한다.

- TreeSet은 이진 탐색 트리를 개선한 레드-블랙 트리를 사용한다. 기본 개념은 비슷하므로 이진 탐색 트리의 원리를 알아보자.

## 트리 구조의 구현



```

class Node {
    Object item;
    Node left;
    Node right;
}
  
```

- 트리 구조는 왼쪽, 오른쪽 노드를 알고 있으면 된다.
- 앞서 다룬 연결 리스트의 구현을 떠올려보면 쉽게 이해가 될 것이다.

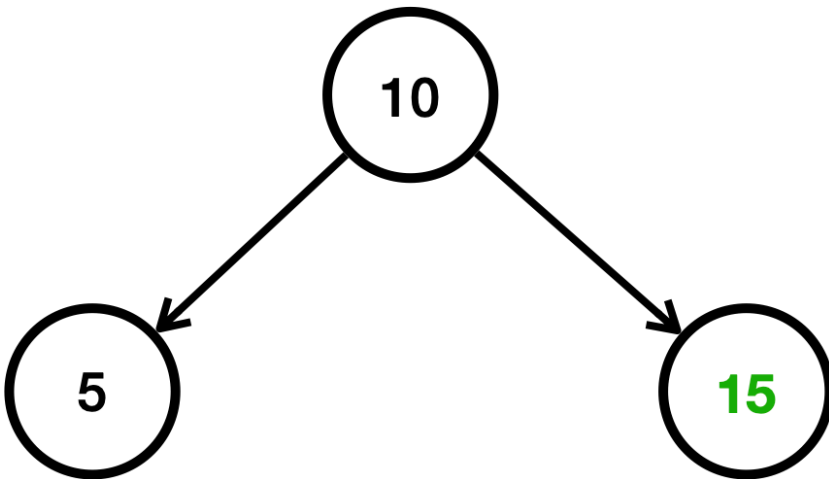
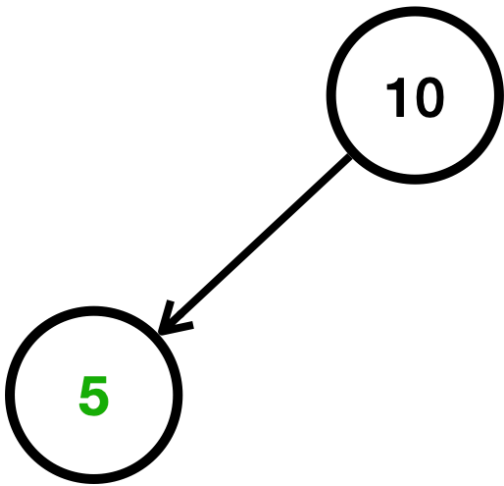
## 이진 탐색 트리 - 입력 예시

이진 탐색 트리의 핵심은 데이터를 입력하는 시점에 정렬해서 보관한다는 점이다.

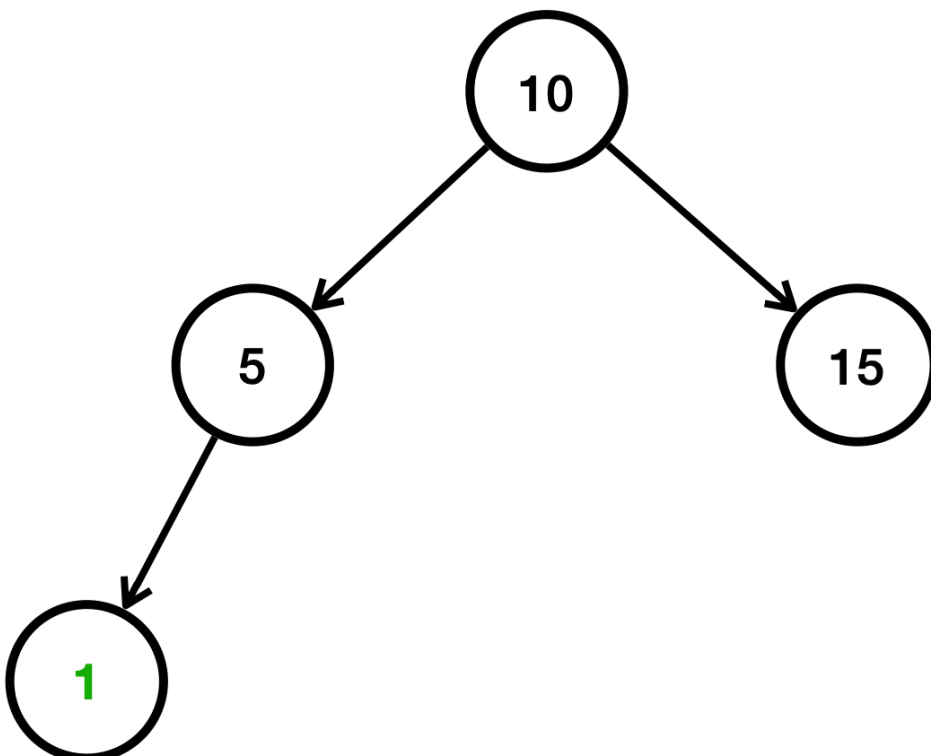
그리고 작은 값은 왼쪽에 큰 값은 오른쪽에 저장하면 된다.

데이터를 10, 5, 15, 1, 6, 11, 16 순서대로 입력한다고 가정해보자.

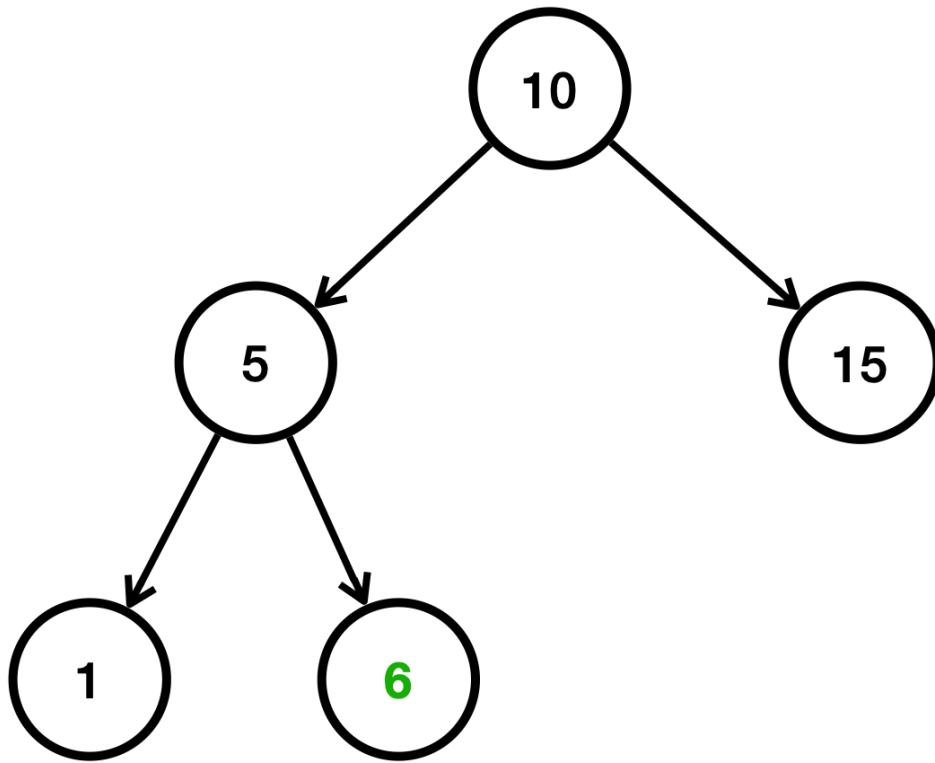
처음에 10을 입력했다고 가정하자. 다음으로 5, 15를 입력한다.



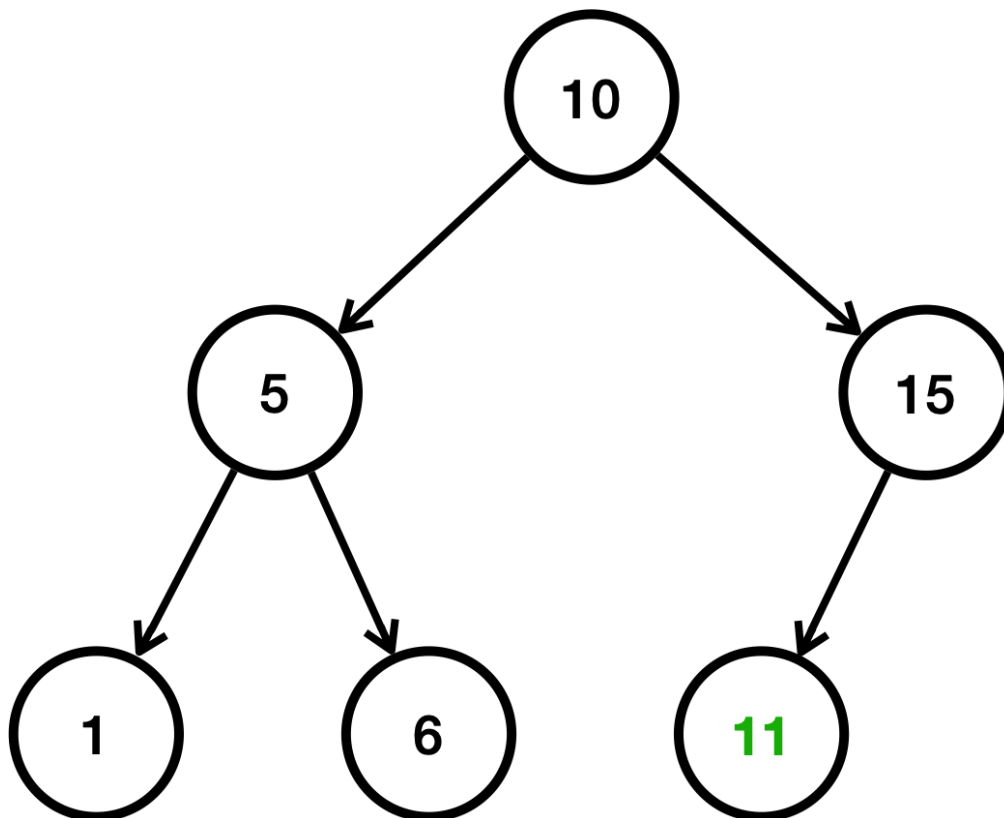
- 5는 10보다 작으므로 왼쪽에 저장된다.
- 15는 10보다 크므로 오른쪽에 저장된다.

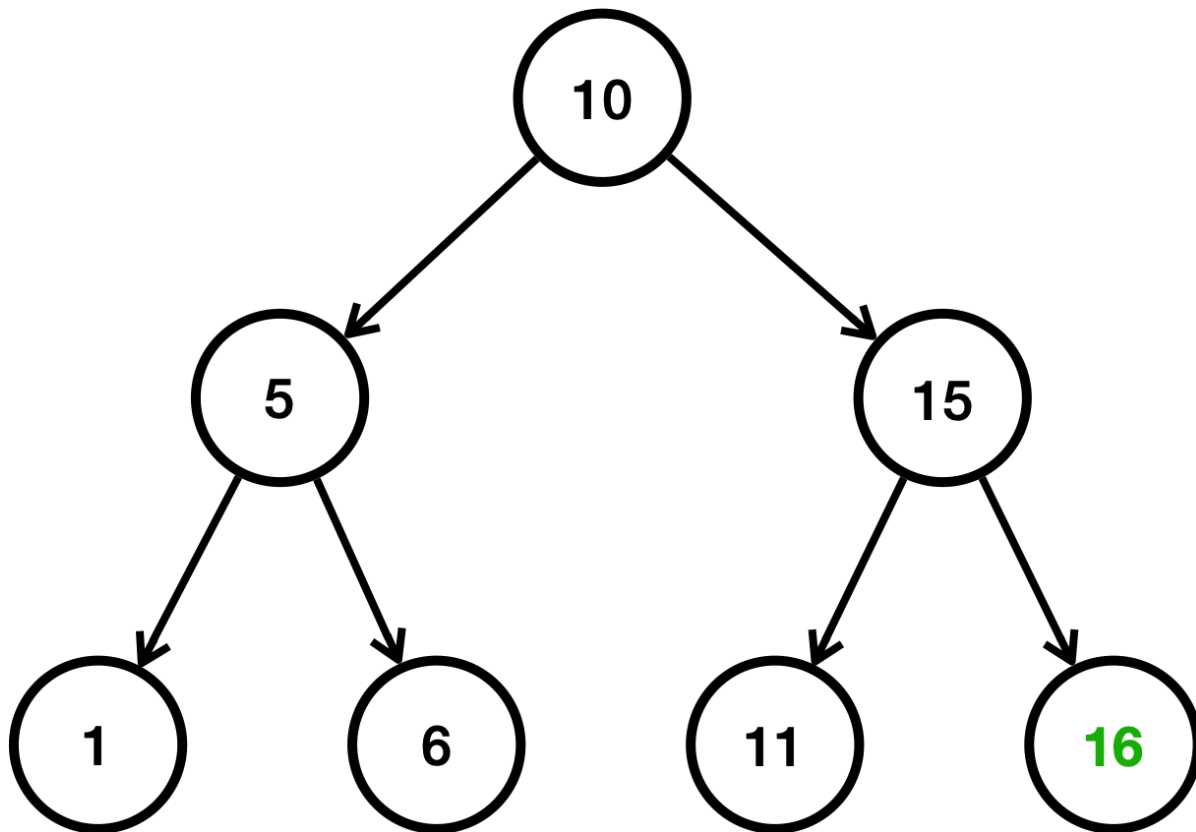






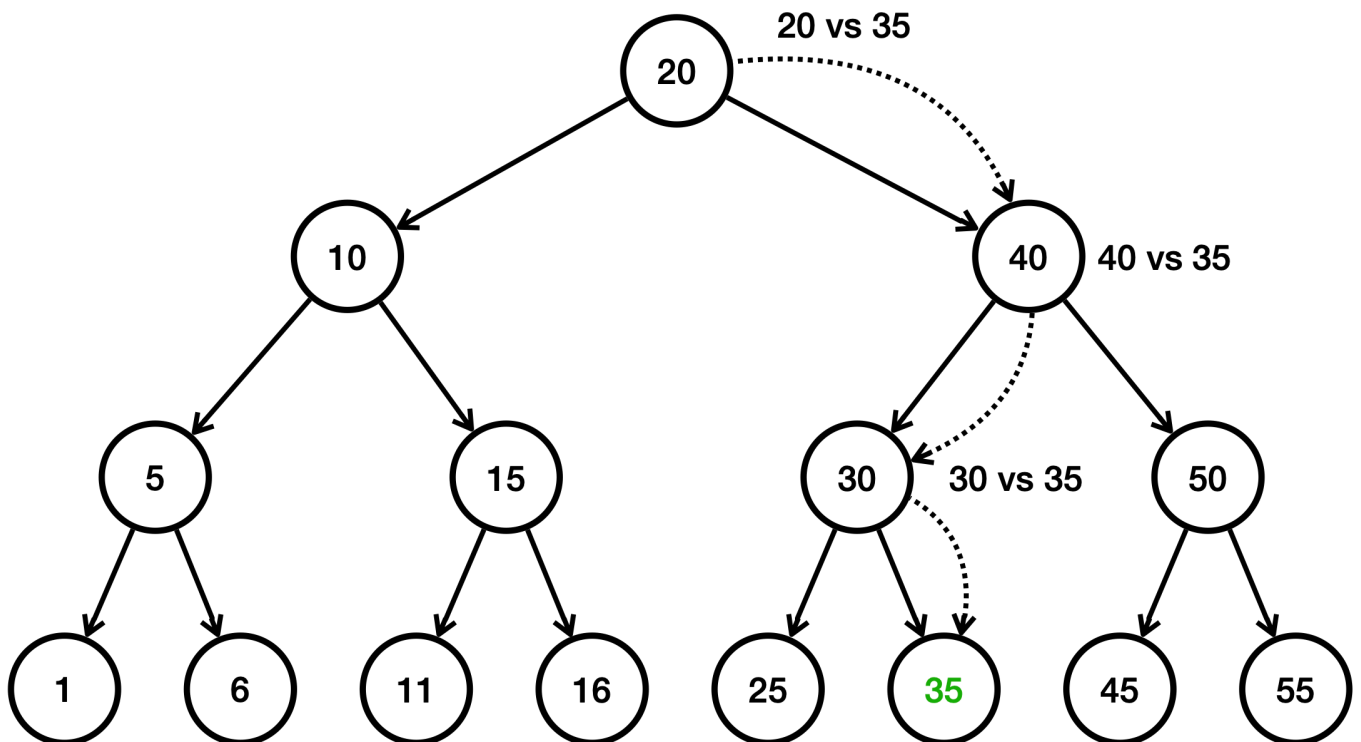
- 1은 10보다 작다. 따라서 왼쪽으로 찾아간다. 1은 5보다 작다 따라서 왼쪽에 저장된다.
- 6은 10보다 작다. 따라서 왼쪽으로 찾아간다. 6은 5보다 크다. 따라서 오른쪽에 저장된다.





- 11은 10보다 크다. 따라서 오른쪽으로 찾아간다. 11은 15보다 작다. 따라서 왼쪽에 저장된다.
- 16은 10보다 크다. 따라서 오른쪽으로 찾아간다. 16은 15보다 크다. 따라서 오른쪽에 저장된다.

#### 이진 탐색 트리 - 검색



- 여기에는 총 15개의 데이터가 들어있다. 여기서 숫자 35를 검색한다고 가정해보자.
- 1번: 루트인 20과 35를 비교한다. 35가 더 크므로 오른쪽으로 찾아간다.

- 2번: 40과 35를 비교한다. 35가 더 작으므로 왼쪽으로 찾아간다.
- 3번: 30과 35를 비교한다. 35가 더 크므로 오른쪽으로 찾아간다.
- 4번: 노드에 있는 값을 비교한다. 35와 같으므로 35를 찾는다.

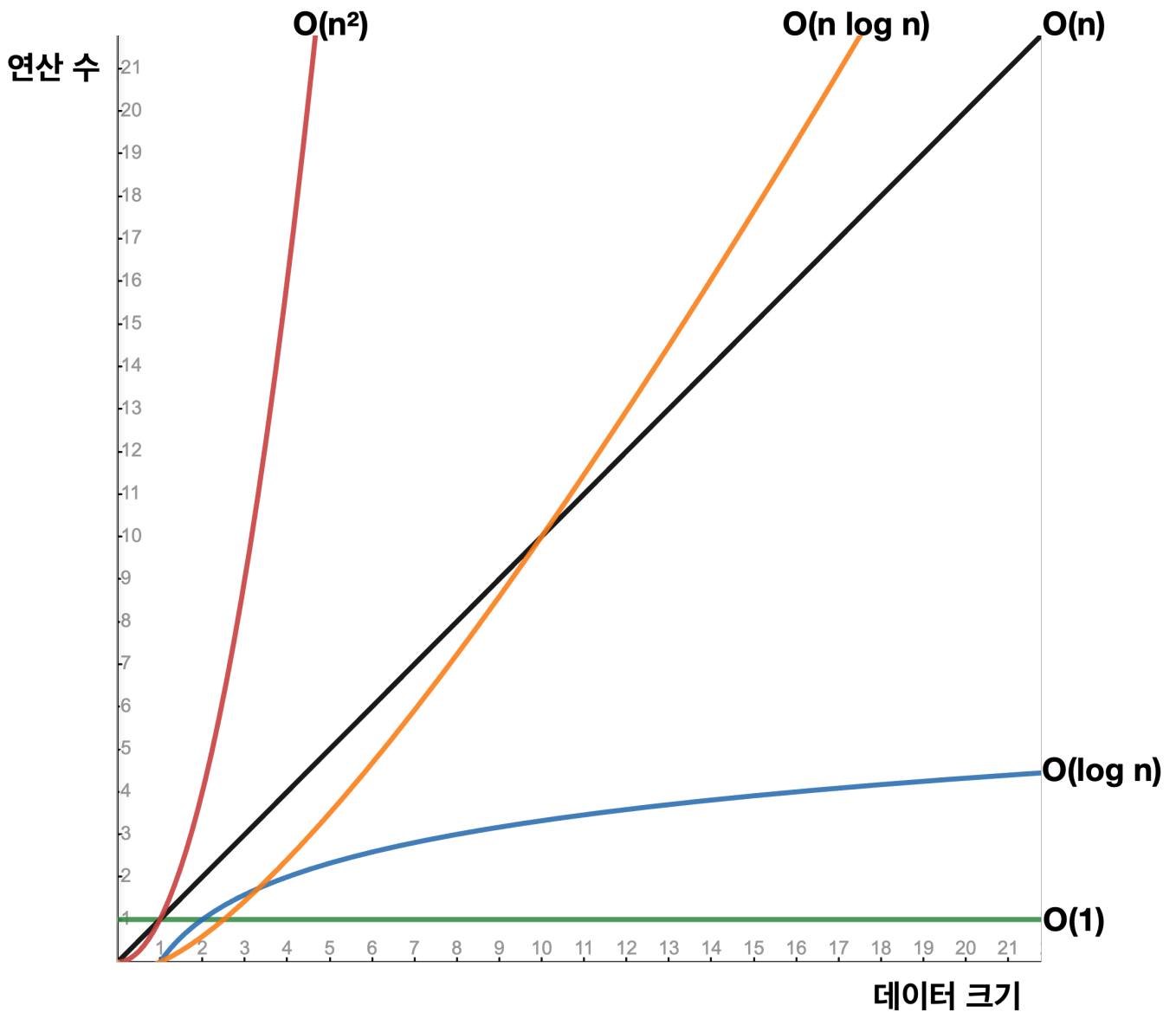
데이터가 총 15개인데 4번의 계산으로 필요한 결과를 얻을 수 있었다. 이것은  $O(n)$ 인 리스트의 검색보다는 빠르고,  $O(1)$ 인 해시의 검색 보다는 느리다.

- 리스트의 경우  $O(n)$ 이므로 15번의 연산이 필요하다.
- 해시 검색은  $O(1)$ 이므로 1번의 연산이 필요하다.

이진 탐색 트리 계산의 핵심은 한번에 절반을 날린다는 점이다. 계산을 단순화 하기 위해 16개의 데이터가 있다고 가정하자.

- 16개의 데이터가 있다. 루트에서 처음 비교를 통해 절반의 데이터를 찾지 않아도 된다. 따라서  $16 / 2 = 8$ 이 된다.
- 8개의 데이터가 있다. 비교를 통해 절반의 데이터만 남는다. 따라서  $8 / 2 = 4$ 가 된다.
- 4개의 데이터가 있다. 비교를 통해 절반의 데이터만 남는다. 따라서  $4 / 2 = 2$ 가 된다.
- 2개의 데이터가 있다. 비교를 통해 절반의 데이터만 남는다. 따라서  $2 / 2 = 1$ 이 된다.
- 1이 남았으므로 이 값이 맞는지 확인하면 된다.

**이진 탐색 트리의 빅오 -  $O(\log n)$**



16개의 경우 단 4번의 비교만으로 최종 노드에 도달할 수 있다. 이것을 정리하면 다음과 같다.

- 2개의 데이터 → 2로 1번 나누기,  $\log_2(2)=1$
- 4개의 데이터 → 2로 2번 나누기,  $\log_2(4)=2$
- 8개의 데이터 → 2로 3번 나누기,  $\log_2(8)=3$
- 16개의 데이터 → 2로 4번 나누기,  $\log_2(16)=4$
- 32개의 데이터 → 2로 5번 나누기,  $\log_2(32)=5$
- 64개의 데이터 → 2로 6번 나누기,  $\log_2(64)=6$
- ...
- 1024개의 데이터 → 2로 10번 나누기,  $\log_2(1024)=10$

1024개의 데이터를 단 10번의 계산으로 원하는 결과를 찾을 수 있다. 데이터의 크기가 늘어나도 늘어난 만큼 한 번의 계산에 절반을 날려버리기 때문에,  $O(n)$ 과 비교해서 데이터의 크기가 클수록 효과적이다.

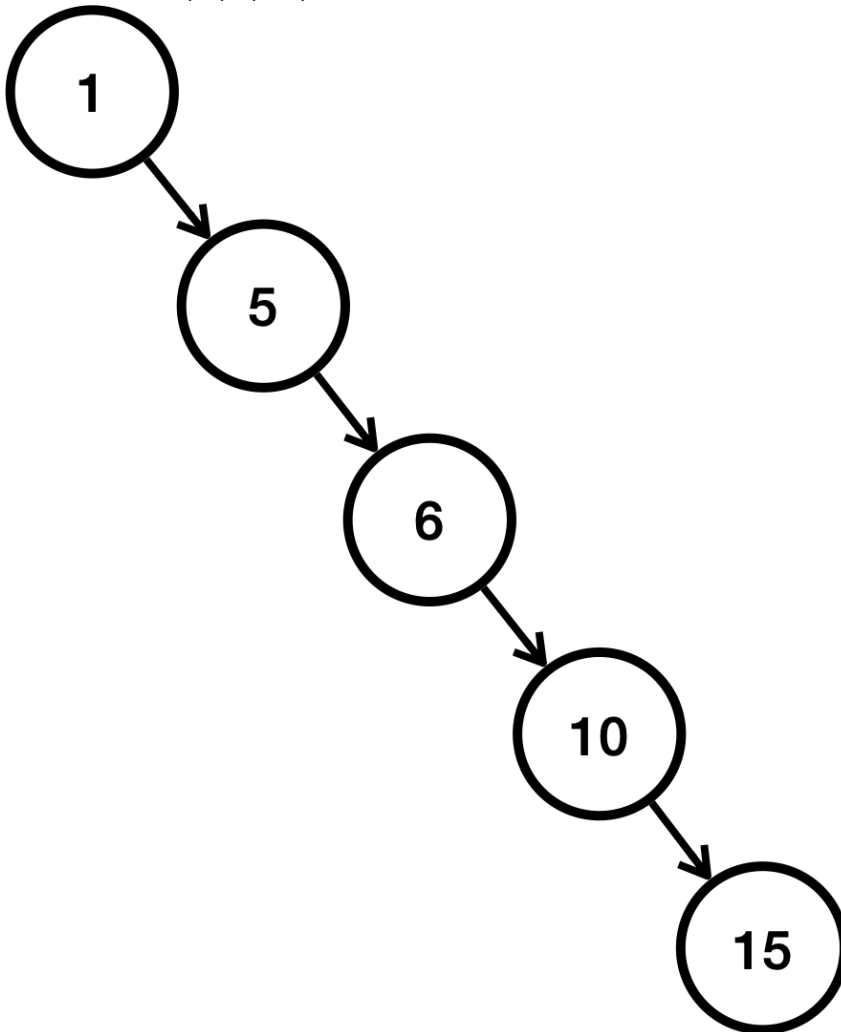
이것을 수학으로  $\log_2(n)$  으로 표현한다. 여기서 어려운 수학 공식이 핵심이 아니다. 로그는 쉽게 이야기해서 2로 몇번 나누어서 1에 도달할 수 있는지 계산하면 된다.

빅오 표기법에서 상수는 사용하지 않으므로 상수를 제외하고 단순히  $O(\log n)$  로 표현한다.

### 이진 탐색 트리와 성능

이진 탐색 트리의 검색, 삽입, 삭제의 평균 성능은  $O(\log n)$  이다. 하지만 트리의 균형이 맞지 않으면 최악의 경우  $O(n)$ 의 성능이 나온다.

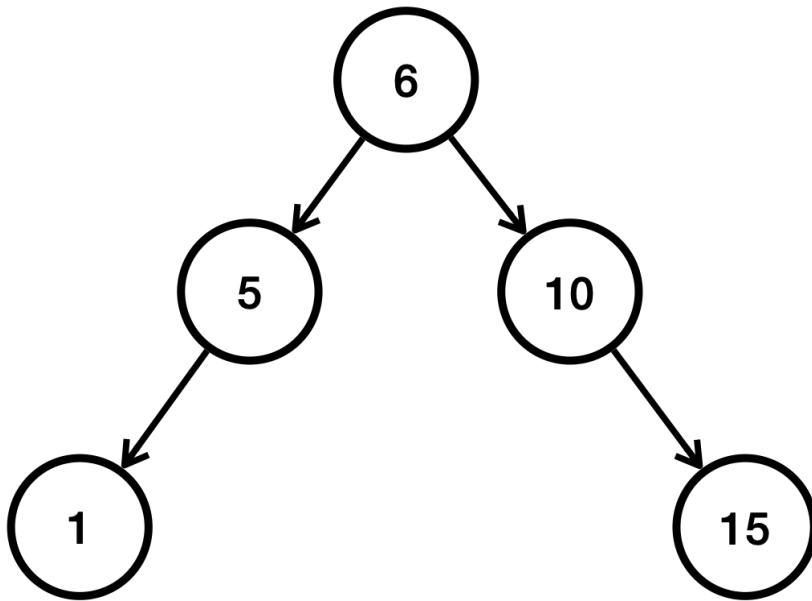
만약 데이터를 1, 5, 6, 10, 15 순서로 입력했다고 가정해보자.



- 이렇게 오른쪽으로 치우치게 되면, 결과적으로 15를 검색 했을 때 데이터의 수인 5만큼 검색을 해야 한다.
- 따라서 이런 최악의 경우  $O(n)$ 이 성능이 나온다.

### 이진 탐색 트리 개선

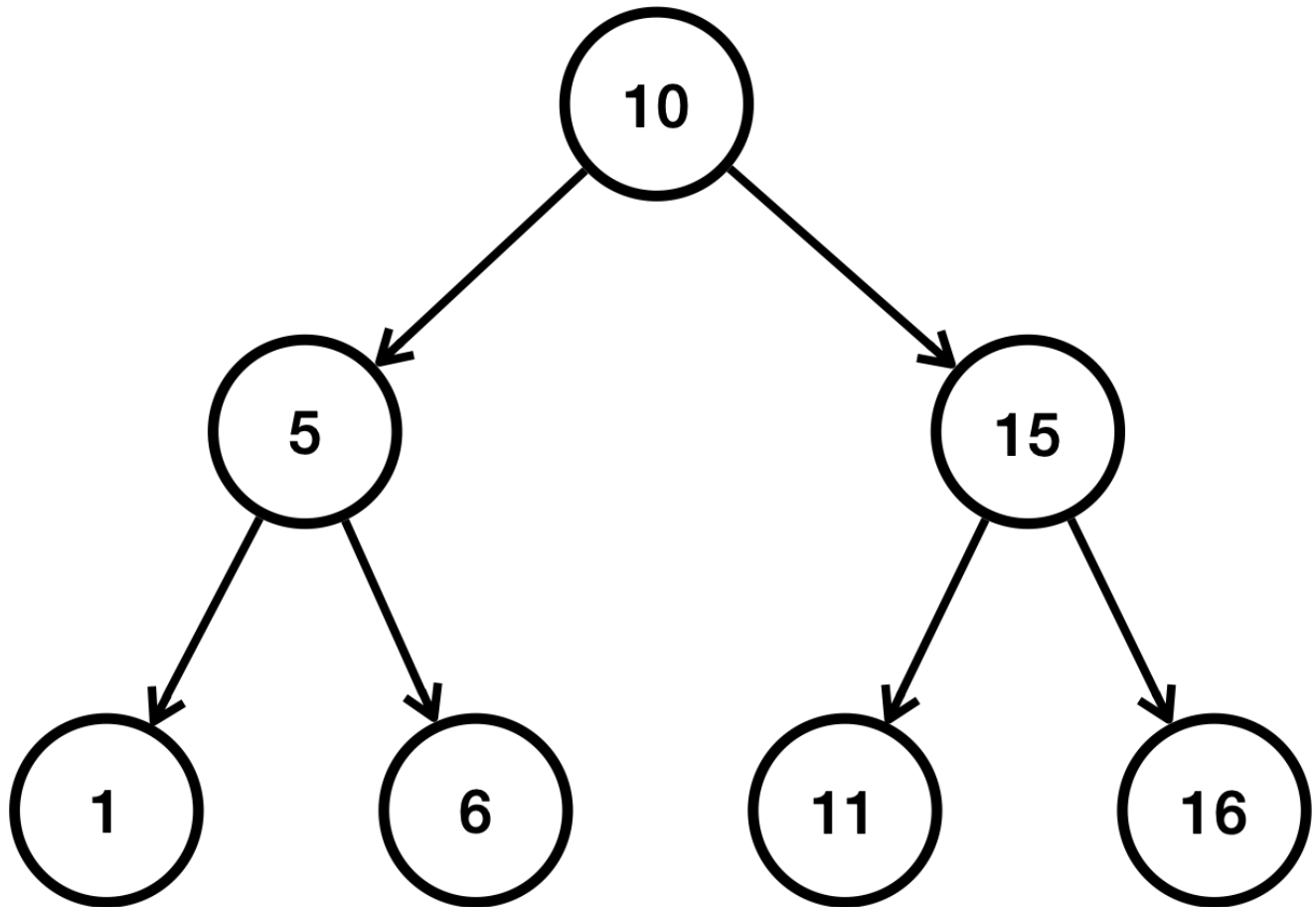
이런 문제를 해결하기 위한 다양한 해결 방안이 있는데 트리의 균형이 너무 깨진 경우 동적으로 균형을 다시 맞추는 것이다.



- 앞서 중간에 있는 6을 기준으로 다시 정렬한다.
- AVL 트리, 레드-블랙 트리 같은 균형을 맞추는 다양한 알고리즘이 존재한다.
- 자바의 TreeSet은 레드-블랙 트리를 사용해서 균형을 지속해서 유지한다. 따라서 최악의 경우에도  $O(\log n)$ 의 성능을 제공한다.

#### 이진 탐색 트리 - 순회

- 이진 탐색 트리의 핵심은 입력 순서가 아니라, 데이터의 값을 기준으로 정렬해서 보관한다는 점이다.
- 따라서 정렬된 순서로 데이터를 차례로 조회할 수 있다. (순회 할 수 있다.)
- 데이터를 차례로 순회하려면 중위 순회라는 방법을 사용하면 된다. 왼쪽 서브트리를 방문한 다음, 현재 노드를 처리하고, 마지막으로 오른쪽 서브트리를 방문한다. 이 방식은 이진 탐색 트리의 특성상, 노드를 오름차순(숫자가 점점 커짐)으로 방문한다.



#### 중위 순회 순서

쉽게 이야기해서 자신의 왼쪽의 모든 노드를 처리하고, 자신의 노드를 처리하고, 자신의 오른쪽 모든 노드를 처리하는 방식이다.

- 10의 기준에서 왼쪽 서브트리를 방문한다.
  - 5의 기준에서 왼쪽 서브트리를 방문한다.
    - ◆ 1을 출력한다.
  - 5 자신을 출력한다.
  - 5의 기준으로 오른쪽 서브트리를 방문한다.
    - ◆ 6을 출력한다.
- 10 자신을 출력한다.
- 10의 기준에서 오른쪽 서브트리를 방문한다.
  - 15의 기준에서 왼쪽 서브트리를 방문한다.
    - ◆ 11을 출력한다.
  - 15 자신을 출력한다.
  - 15의 기준으로 오른쪽 서브트리를 방문한다.
    - ◆ 16을 출력한다.

순서대로 1, 5, 6, 10, 11, 15, 16이 출력된 것을 확인할 수 있다.

**참고:** 여기서는 TreeSet 을 알고 사용하기 위한 정도의 기본적인 트리 이론을 다루었다. 트리 구조에 대해서 더 자세

히 알고 싶다면 자료 구조와 알고리즘을 학습하자.

## 자바가 제공하는 Set3 - 예제

HashSet, LinkedHashSet, TreeSet 에서 학습한 내용을 코드로 확인해보자.

```
package collection.set.javaset;

import java.util.*;

public class JavaSetMain {

    public static void main(String[] args) {
        run(new HashSet<>());
        run(new LinkedHashSet<>());
        run(new TreeSet<>());
    }

    private static void run(Set<String> set) {
        System.out.println("set = " + set.getClass());
        set.add("C");
        set.add("B");
        set.add("A");
        set.add("1");
        set.add("2");

        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();
    }
}
```

- HashSet, LinkedHashSet, TreeSet 모두 Set 인터페이스를 구현하기 때문에 구현체를 변경하면서 실행할 수 있다.
- iterator() 를 호출하면 컬렉션을 반복해서 출력할 수 있다.
  - iterator.hasNext() : 다음 데이터가 있는지 확인한다.



- `iterator.next()`: 다음 데이터를 반환한다.

## 실행 결과

```
set = class java.util.HashSet
```

```
A 1 B 2 C
```

```
set = class java.util.LinkedHashSet
```

```
C B A 1 2
```

```
set = class java.util.TreeSet
```

```
1 2 A B C
```

- `HashSet`: 입력한 순서를 보장하지 않는다.
- `LinkedHashSet`: 입력한 순서를 정확히 보장한다.
- `TreeSet`: 데이터 값을 기준으로 정렬한다.

## 참고 - `TreeSet`의 정렬 기준

`TreeSet`을 사용할 때 데이터를 정렬하려면 크다, 작다라는 기준이 필요하다. 1, 2, 3이나 "A", "B", "C" 같은 기본 데이터는 크다 작다라는 기준이 명확하기 때문에 정렬할 수 있다. 하지만 우리가 직접 만든 `Member`와 같은 객체는 크다 작다는 기준을 어떻게 알 수 있을까? 이런 기준을 제공하려면 `Comparable`, `Comparator` 인터페이스를 구현해야 한다. 이 부분은 뒤에서 설명한다.

# 자바가 제공하는 Set4 - 최적화

## 자바 `HashSet`과 최적화

- 자바의 `HashSet`은 우리가 직접 구현한 내용과 거의 같지만 다음과 같은 최적화를 추가로 진행한다.

## 최적화

- 해시 기반 자료 구조를 사용하는 경우 통계적으로 입력한 데이터의 수가 배열의 크기를 75% 정도 넘어가면 해시 인덱스가 자주 충돌한다. 따라서 75%가 넘어가면 성능이 떨어지기 시작한다.
  - 해시 충돌로 같은 해시 인덱스에 들어간 데이터를 검색하려면 모두 탐색해야 한다. 따라서 성능이  $O(n)$ 으로 좋지 않다.
- 하지만 데이터가 동적으로 계속 추가되기 때문에 적절한 배열의 크기를 정하는 것은 어렵다.
- 자바의 `HashSet`은 데이터의 양이 배열 크기의 75%를 넘어가면 배열의 크기를 2배로 늘리고 2배 늘어난 크기를 기준으로 모든 요소에 해시 인덱스를 다시 적용한다.

- 해시 인덱스를 다시 적용하는 시간이 걸리지만, 결과적으로 해시 충돌이 줄어든다.
- 자바 HashSet 의 기본 크기는 16 이다.

데이터 양 75% 이상 증가

**hashCode % CAPACITY(5)**

	[1,6]	[2,7]		
[0]	[1]	[2]	[3]	[4]

- 데이터 양이 75% 이상 증가하면 그 만큼 해시 인덱스의 충돌 가능성도 높아진다.

배열의 크기 2배 증가, 해시 다시 계산

**hashCode % CAPACITY(10)**

	[1]	[2]				[6]	[7]		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

- 데이터양이 75% 이상이면 배열의 크기를 2배로 증가하고, 모든 데이터의 해시 인덱스를 커진 배열에 맞추어 다시 계산한다. 이 과정을 재해싱(rehashing)이라 한다.
- 인덱스 충돌 가능성이 줄어든다.
- 여기서 데이터가 다시 75% 이상 증가하면 다시 2배 증가와 재계산을 반복한다.

**정리**

실무에서는 Set 이 필요한 경우 HashSet 을 가장 많이 사용한다. 그리고 입력 순서 유지, 값 정렬의 필요에 따라서 LinkedHashMap, TreeSet 을 선택하면 된다.

# 문제와 풀이1

## 문제1 - 중복 제거

### 문제 설명

- 여러 정수가 입력된다. 여기서 중복 값을 제거하고 값을 출력해라.
- 30, 20, 20, 10, 10이 출력되면 중복을 제거하고 출력하면 된다. 출력 순서는 관계없다.
  - 출력 예): 30, 20, 10 또는 10, 20, 30 또는 20, 10, 30등과 같이 출력 순서는 관계 없다.

```
package collection.set.test;

import java.util.HashSet;
import java.util.Set;

public class UniqueNamesTest1 {

    public static void main(String[] args) {
        Integer[] inputArr = {30, 20, 20, 10, 10};

        // 코드 작성
    }
}
```

### 실행 결과

```
20
10
30
```

### 정답

```
package collection.set.test;

import java.util.HashSet;
import java.util.Set;

public class UniqueNamesTest1 {

    public static void main(String[] args) {
        Integer[] inputArr = {30, 20, 20, 10, 10};
        Set<Integer> set = new HashSet<>();
        for (Integer s : inputArr) {
```

```

        set.add(s);
    }

    for (Integer s : set) {
        System.out.println(s);
    }
}

```

- HashSet 을 사용하면 중복 데이터는 저장되지 않는다.
- 단순히 HashSet 에 값을 입력하고 HashSet 을 출력하면 된다.
- HashSet 은 순서를 보장하지 않는다.

## 문제2 - 중복 제거와 입력 순서 유지

### 문제 설명

- 여러 정수가 입력된다. 여기서 중복 값을 제거하고 값을 출력해라.
- 30, 20, 20, 10, 10이 출력되면 중복을 제거하고 출력하면 된다.
- 단 **입력 순서대로 출력**해라.
  - 출력 예): 30, 20, 10

```

package collection.set.test;

import java.util.HashSet;
import java.util.Set;

public class UniqueNamesTest2 {

    public static void main(String[] args) {
        Integer[] inputArr = {30, 20, 20, 10, 10};

        // 코드 작성
    }
}

```

### 실행 결과

```

30
20
10

```

## 정답

```
package collection.set.test;

import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

public class UniqueNamesTest2 {

    public static void main(String[] args) {
        Integer[] inputArr = {30, 20, 20, 10, 10};
        Set<Integer> set = new LinkedHashSet<>(List.of(inputArr));

        for (Integer s : set) {
            System.out.println(s);
        }
    }
}
```

- 입력 순서대로 출력하려면 `LinkedHashSet` 을 사용하면 된다.

배열을 `Set` 에 입력할 때 직접 배열을 반복하면서 `Set` 에 입력하는 방법도 있지만 더 간단히 해결하는 방법이 있다. `Set` 구현체의 생성자에 배열은 전달할 수 없지만 `List` 는 전달할 수 있다. 다음과 같이 배열을 `List` 로 변환한다.

### 배열을 리스트로 변환하기

```
List<Integer> list = Arrays.asList(inputArr);
List<Integer> list = List.of(inputArr);
```

### 편리한 리스트 생성

```
List<Integer> list = Arrays.asList(1, 2, 3);
List<Integer> list = List.of(1, 2, 3);
```

`Arrays.asList()` 메서드는 자바 1.2부터 존재했다. 자바 9 이상을 사용한다면 `List.of()` 를 권장한다. 둘에 대한 자세한 내용은 뒤에서 설명한다.

## 문제3 - 중복 제거와 데이터 순서 유지

### 문제 설명

- 여러 정수가 입력된다. 여기서 중복 값을 제거하고 값을 출력해라.

- 30, 20, 20, 10, 10이 출력되면 중복을 제거하고 출력하면 된다.
- 데이터의 값 순서로 출력해라.
  - 출력 예): 10, 20, 30

```
package collection.set.test;

import java.util.HashSet;
import java.util.Set;

public class UniqueNamesTest3 {

    public static void main(String[] args) {
        Integer[] inputArr = {30, 20, 20, 10, 10};

        // 코드 작성
    }
}
```

#### 실행 결과

```
10
20
30
```

#### 정답

```
package collection.set.test;

import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class UniqueNamesTest3 {

    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<>(List.of(30, 20, 20, 10, 10));

        for (Integer s : set) {
            System.out.println(s);
        }
    }
}
```

- 데이터의 값 순서대로 출력하려면 `TreeSet` 을 사용하면 된다.

## 문제와 풀이2

### 문제4 - 합집합, 교집합, 차집합

#### 문제 설명

- 두 숫자의 집합이 제공된다.
  - 집합1: 1, 2, 3, 4, 5
  - 집합2: 3, 4, 5, 6, 7
- 두 집합의 합집합, 교집합, 차집합을 구해라. 출력 순서는 관계없다.
  - 합집합: 두 집합의 합이다. 참고로 중복은 제거한다. [1, 2, 3, 4, 5, 6, 7]
  - 교집합: 두 집합의 공통 값이다. 참고로 중복은 제거한다. [3, 4, 5]
  - 차집합: 집합1에서 집합2와 같은 값을 뺀 나머지 [1, 2]
- 다음 실행 결과를 참고하자.
- `Set` 인터페이스의 주요 메서드를 참고하자.

#### SetOperationsTest - 코드 작성

```
package collection.set.test;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class SetOperationsTest {
    public static void main(String[] args) {
        Set<Integer> set1 = new HashSet<>(List.of(1, 2, 3, 4, 5));
        Set<Integer> set2 = new HashSet<>(List.of(3, 4, 5, 6, 7));

        //코드 작성
    }
}
```

#### 실행 결과

합집합: [1, 2, 3, 4, 5, 6, 7]  
교집합: [3, 4, 5]

차집합: [1, 2]

## 정답

```
package collection.set.test;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class SetOperationsTest {
    public static void main(String[] args) {
        Set<Integer> set1 = new HashSet<>(List.of(1, 2, 3, 4, 5));
        Set<Integer> set2 = new HashSet<>(List.of(3, 4, 5, 6, 7));

        Set<Integer> union = new HashSet<>(set1);
        union.addAll(set2);

        Set<Integer> intersection = new HashSet<>(set1);
        intersection.retainAll(set2);

        Set<Integer> difference = new HashSet<>(set1);
        difference.removeAll(set2);

        System.out.println("합집합: " + union);
        System.out.println("교집합: " + intersection);
        System.out.println("차집합: " + difference);
    }
}
```

## 문제5 - Equals, hashCode

### 문제 설명

- RectangleTest, 실행 결과를 참고해서 다음 Rectangle 클래스를 완성하자.
- Rectangle 클래스는 width, height 가 모두 같으면 같은 값으로 정의한다.

```
package collection.set.test;

public class Rectangle {
```



```

    private int width;
    private int height;

    // 코드 작성
}

```

```

package collection.set.test;

import java.util.HashSet;
import java.util.Set;

public class RectangleTest {

    public static void main(String[] args) {
        Set<Rectangle> rectangleSet = new HashSet<>();
        rectangleSet.add(new Rectangle(10, 10));
        rectangleSet.add(new Rectangle(20, 20));
        rectangleSet.add(new Rectangle(20, 20)); //중복

        for (Rectangle rectangle : rectangleSet) {
            System.out.println("rectangle = " + rectangle);
        }
    }
}

```

### 실행 결과

```

rectangle = Rectangle{width=10, height=10}
rectangle = Rectangle{width=20, height=20}

```

### 정답

```

package collection.set.test;

import java.util.Objects;

public class Rectangle {

    private int width;
    private int height;

    public Rectangle(int width, int height) {

```

```

        this.width = width;
        this.height = height;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Rectangle rectangle = (Rectangle) o;
        return width == rectangle.width && height == rectangle.height;
    }

    @Override
    public int hashCode() {
        return Objects.hash(width, height);
    }

    @Override
    public String toString() {
        return "Rectangle{" +
            "width=" + width +
            ", height=" + height +
            '}';
    }
}

```

- HashSet 자료 구조에 저장된다. 해시 알고리즘을 사용하므로 반드시 hashCode(), equals() 를 재정의해야 한다.

## 정리