



collection 소개서



Java Collection Framework

CONTENTS

1

- Java Collection Framework란?
- 주요 특징 및 이점
- 자바 컬렉션 프레임워크의 중요성

2

- Iterable 인터페이스
- Collection 인터페이스
- List, Set, Queue 인터페이스 설명

3

- List 인터페이스
- List 인터페이스 구현 클래스
 - ArrayList
 - LinkedList

4

- Queue 인터페이스

- Queue 인터페이스 구현 클래스
 - PriorityQueue
 - Deque
 - LinkedList

- Set 인터페이스

- Set 인터페이스 구현 클래스
 - HashSet
 - TreeSet

5

- Map 인터페이스
- Map.Entry 인터페이스
- Map 인터페이스 구현 클래스
 - HashMap
 - LinkedHashMap
 - HashTable
 - TreeMap
 - Properties

6

- 고급 주제
 - Sequenced Collection
 - Concurrent Collections
 - Custom Collections
- 결론

Java Collection Framework란?



정의

Java Collection Framework는 객체의 그룹을 효율적으로 관리하고 조작하기 위한 표준화된 아키텍처를 제공합니다. 컬렉션은 여러 개의 객체를 하나의 단위로 묶어 관리할 수 있는 구조로, 이 프레임워크는 이러한 컬렉션을 다루기 위한 다양한 인터페이스와 클래스를 포함합니다. 자료 구조(Data Structure) 종류의 형태들을 자바 클래스로 구현한 모음집

목적

1. 데이터 저장 및 조작: 데이터를 효율적으로 저장하고 검색 할수있는 방법 제공
2. 표준화: 컬렉션을 다루는 일관된 방법 제공
3. 재사용성: 범용적인 컬렉션 클래스를 사용하여 코드 재사용성 향상

Java Collection Framework

주요 특징 및 이점

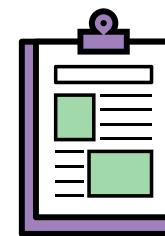
STEP 1



다양한 데이터 구조

- List, Set, Queue, Map 등의 다양한 컬렉션 타입 제공

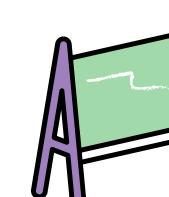
STEP 2



제네릭(Generic)

- 타입 안전성 보장
- 컴파일 타임에 타입 체크

STEP 3



표준화된 메서드

- 공통된 메서드 제공 (추가, 삭제, 검색 등)
- 인터페이스와 구현체간의 일관성

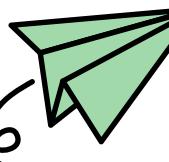
STEP 4



고성능 구현체

- 성능 최적화를 위해 설계된 클래스 제공
- 관련 없는 API 간의 상호 운용성 제공. (상위 인터페이스 타입으로 업캐스팅 후 사용)

STEP 5

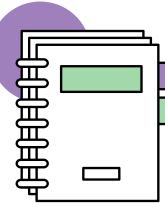


유ти리티 클래스

- 유ти리티 클래스 제공 (Collections, Arrays 등)
- 다양한 작업 지원 (정렬, 검색, 동기화)

Java Collection Framework

중요성



- **효율성:** 데이터를 효율적으로 관리하고 조작할 수 있는 구조와 알고리즘 제공
- **유연성:** 다양한 데이터 구조와 알고리즘을 제공하여 다양한 프로그래밍 요구사항을 충족
- **코드 간결성:** 표준화된 API를 통해 복잡한 데이터 구조를 쉽게 다룰 수 있어 코드의 가독성과 유지보수성 향상
- **생산성 향상:** 기본 제공되는 컬렉션 클래스를 통해 개발 시간이 단축되고, 개발자가 핵심 로직에 집중할 수 있음
- **확장성과 호환성:** 새로운 요구사항에 맞춰 쉽게 확장 가능하며, 다른 자바 라이브러리 및 API와의 호환성 보장
- **소프트웨어 재사용 촉진 :** 자바에서 지원하지 않는 새로운 자료구조가 필요하다면, 컬렉션들을 재활용하여 조합하여 새로운 알고리즘 생성 가능.

컬렉션 프레임워크에 저장할 수 있는 데이터는 오로지 객체(Object) 뿐이다.

자바의 primitive 타입은 적재가 불가능하다.

따라서 primitive 타입을 wrapper 타입으로 변환하여 즉 박싱(Boxing)하여 저장하여야 한다.

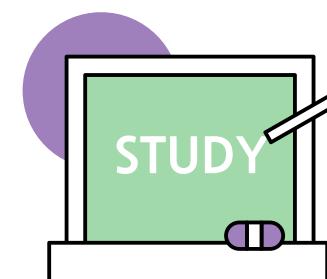
또한 객체를 컬렉션에 저장한다는 것은 그 객체의 참조를 저장하는 것을 의미하는데

객체의 참조는 메모리에 저장된 위치를 가리킨다

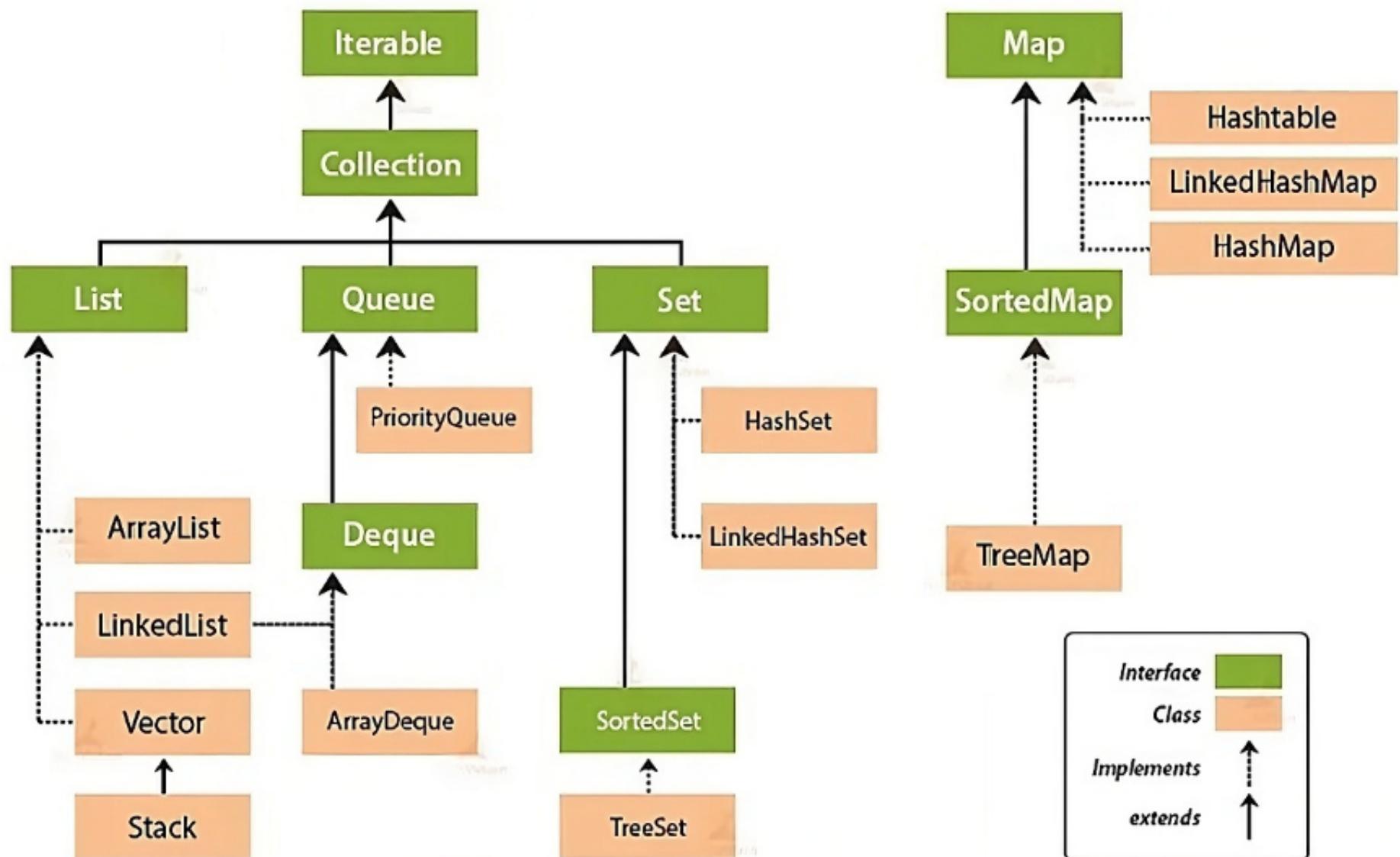
이로 인해 null 값도 저장이 가능하다



- `int` -> `Integer`
- `double` -> `Double`
- `char` -> `Character`
- `boolean` -> `Boolean`



Java Collection Framework



Java Collection Framework

- **Collection 인터페이스**

`List` 와 `Set` 인터페이스를 구현한 컬렉션 클래스들은 공통부분이 많기 때문에, 공통된 부분을 모은 `Collection` 인터페이스로 상속되어 있다.

- **Map 인터페이스**

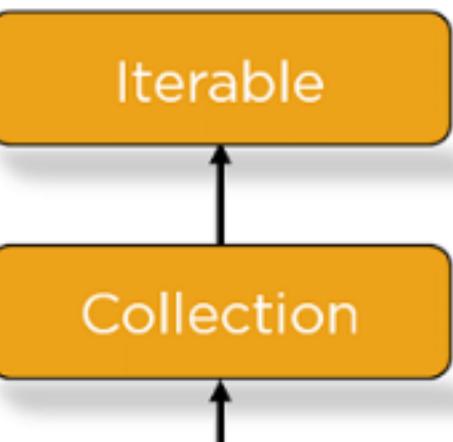
두개의 데이터를 묶어 한쌍으로 다루기 때문에 `Collection` 인터페이스와 따로 분리되어 있다.

대부분의 컬렉션 클래스들은 이 인터페이스들을 구현하고 있으며, 구현한 인터페이스의 이름을 클래스 이름에 포함하는 경향이 있습니다.

하지만 `Vector`, `Stack`, `Hashtable`, `Properties`와 같은 클래스들은 컬렉션 프레임워크가 만들어지기 이전부터 존재했던 클래스들이기 때문에, 컬렉션 프레임워크의 명명법을 따르지 않습니다.

`Vector`나 `Hashtable`과 같은 기존의 컬렉션 클래스들은 호환을 위해 남겨진 것 이므로 가급적 사용하지 않는 것이 좋다.

Iterable 인터페이스



```

public interface Iterable<T> {
    Returns an iterator over elements of this type.
    반환: an Iterator.
    @NotNull
    Iterator<T> iterator();
}
  
```

```

public static void iterator() {
    ArrayList<String> list = new ArrayList<>();
    list.add("Apple");
    list.add("Banana");
    list.add("Orange");

    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        String element = iterator.next();
        System.out.println(element);
    }
}
  
```

Result

```

2024.03.29
Apple
Banana
Orange
  
```

Iterable 인터페이스

iterate : (계산, 컴퓨터 처리 절차를) **반복**하다

iterator : **반복자**

- 컬렉션 인터페이스들의 **가장 최상위 인터페이스**
- 컬렉션 클래스가 **반복 가능한 객체임**을 나타내는데 사용
- **Iterable**을 구현하는 클래스는 컬렉션의 요소를 반복할 수 있는 방법을 제공해야 합니다.

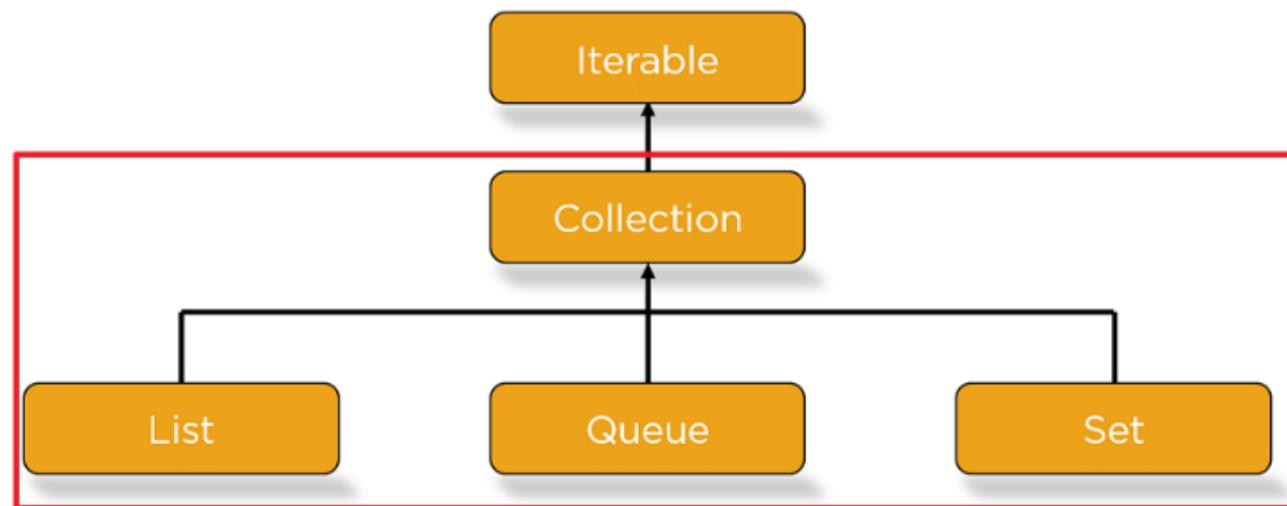
메서드

- **hasNext()**: 다음 요소가 있는지 **확인합니다**.
- **next()**: 다음 요소를 **반환합니다**.
- **remove()**: 현재 반복 중인 요소를 **삭제합니다**.



Map은 iterable 인터페이스를 상속받지 않기 때문에 Map 컬렉션에 구현되어 있지않다.
따라서 직접적으로 Map 컬렉션을 순회할수는 없고 **스트림(Stream)**을 사용하거나
간접적으로 키를 Collection으로 반환하여 루프문으로 순회하는 식을 이용한다.

Collection 인터페이스



```

public interface Collection<E> extends Iterable<E> {
    // Query Operations
    !
    Returns the number of elements in this collection. If this collection contains more than
    MAX_VALUE elements, returns Integer.MAX_VALUE.
    반환: the number of elements in this collection
  
```



Collection 인터페이스

- List, Set, Queue에 상속을 하는 **실질적인 최상위 인터페이스**
- 여러 컬렉션 타입을 위한 **기본적인 메서드를 포함하며,**
이를 확장하여 **List, Set, Queue 인터페이스**가 정의
- 업캐스팅으로 다양한 종류의 컬렉션 자료형을 받아 자료를
삽입, 삭제, 탐색을 할수있다.

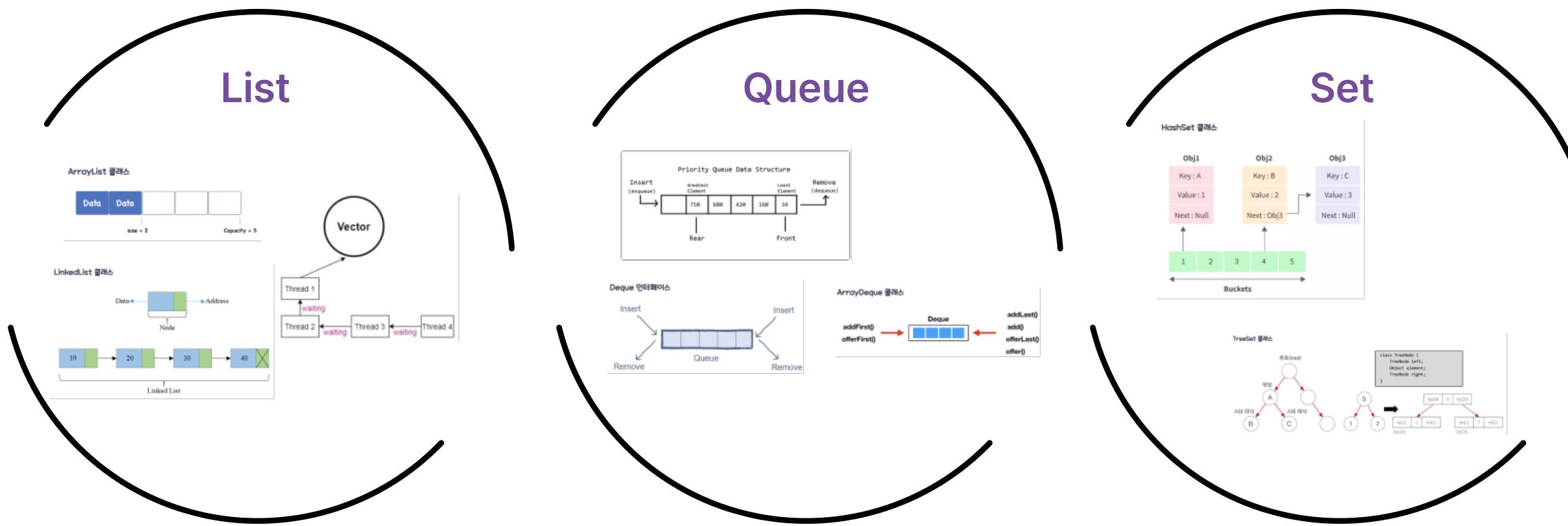
주요 메서드

- add(): 객체 Collection에 **추가**
- contains(): 객체 Collection에 **포함 유무 확인**
- remove(): Collection에 포함된 객체 **삭제**
- isEmpty() : Collection이 **비어있는지 확인**
- size(): Collection에 저장된 객체의 **개수를 반환**



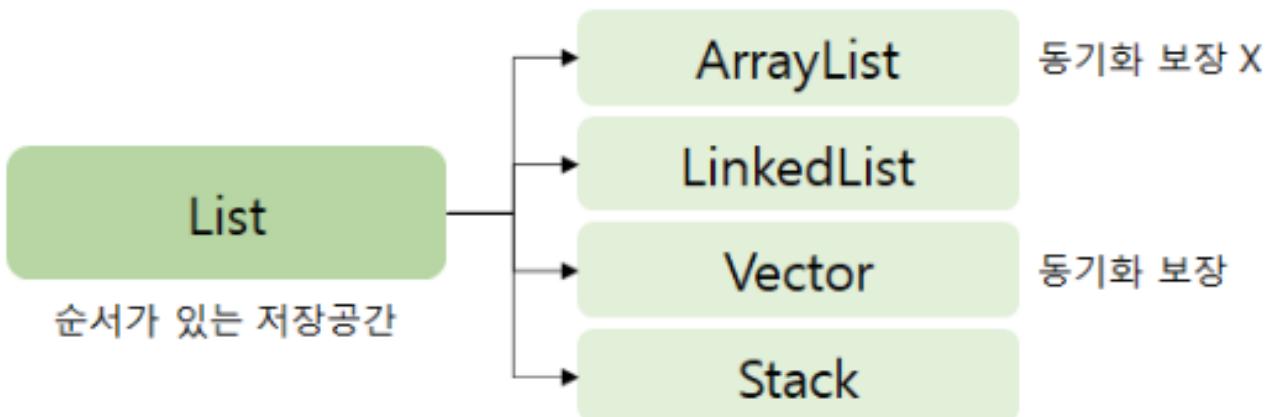
Collection 인터페이스의 메서드에는 데이터를 get 하는 메서드는 보이지 않는다.
왜냐하면 각 컬렉션 자료형마다 구현하는 **자료 구조가 제각각** 이기 때문에
최상위 타입으로 조회하기 까다롭기 때문이다.

collection 인터페이스



구분	List	Queue	Set
순서	순서보장	순서 보장 안함 (요소 처리 순서가 중요)	순서 보장 안함 (일부 구현체는 순서 보장)
중복	중복 요소 허용	중복 요소 허용	중복 허용 안함
탐색	인덱스를 통한 요소 접근	FIFO 구조 특정 상황에 맞춘 우선순위 큐	HashSet을 통한 요소 접근 (효율적)

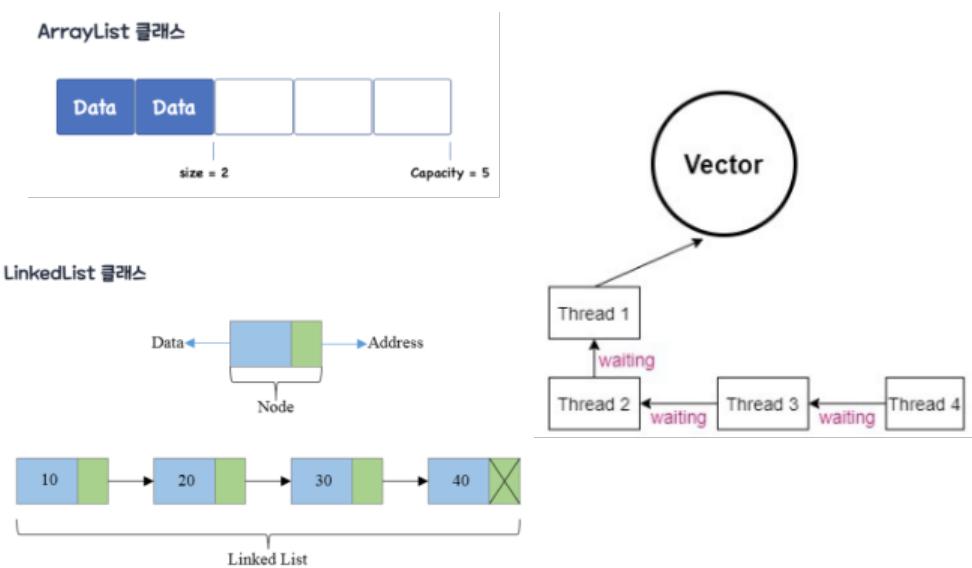
List 인터페이스



List 인터페이스

- 요소의 **저장순서가 유지**되는 컬렉션을 구현하는데 사용
- 요소의 **중복 저장을 허용**
- 요소 접근시 **index** 사용
- 자료형 크기가 **동적으로 변경된다(가변)**
- 요소사이의 **빈공간을 허용하지 않는다.**

```
public interface List<E> extends SequencedCollection<E> {
```



주요 메서드

- `add(int index, E element)`: 특정 위치에 요소 **추가**
- `get(int index)`: 특정 위치의 요소 **반환**
- `remove(int index)`: 특정 위치의 요소 **제거**
- `set(int index, E element)`: 특정 위치의 요소 **변경**

리스트는 배열과 달리 메모리에 연속적으로 나열되어있지 않고
주소로 연결되어있는 형태이기 때문에 `index`를 통한 접근 속도가 배열보다는 느리다
객체로 데이터를 다루기 때문에 적은양의 데이터만 쓸 경우 배열에 비해 차지하는 메모리가 커진다

List 구현 클래스

ArrayList 클래스



1

ArrayList

특징

- 배열을 이용하여 만든 리스트
- 단방향 포인터 구조
- 데이터의 저장순서가 유지되고 중복을 허용

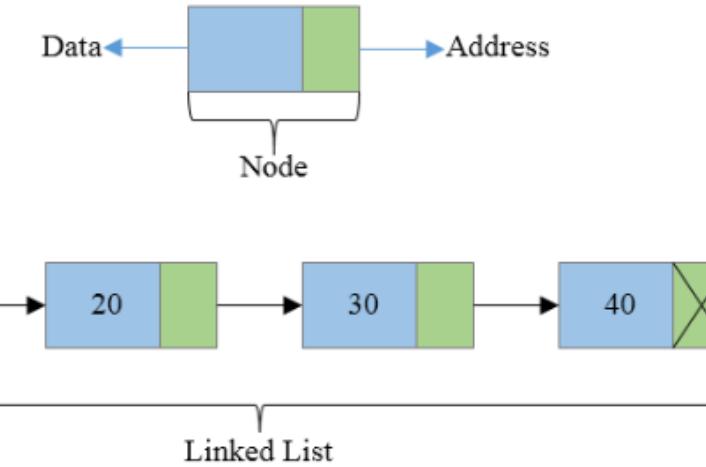
장점

- 빠른 임의 접근: ArrayList는 인덱스를 기반으로 빠른 접근이 가능합니다. 이는 배열의 특징을 그대로 갖고 있기 때문입니다.
- 저장 및 접근 속도가 빠름: 요소들은 인접한 메모리에 저장되어 있어 캐시 지역성이 높아서 접근 속도가 빠릅니다.

단점

- 삽입 및 삭제 속도가 느림: 요소를 중간에 삽입하거나 삭제할 경우 다른 요소들을 이동시켜야 하므로 속도가 느립니다.
- 크기 조절 오버헤드: 크기를 조절하기 위해 요소를 복사하고 재배열해야 하므로 크기 조절 시간이 추가됩니다.

LinkedList 클래스



2

LinkedList

특징

- 노드를 연결하여 리스트 처럼 만든 컬렉션
- 양방향 포인터 구조
- Tree 구조의 근간이 되는 자료구조

장점

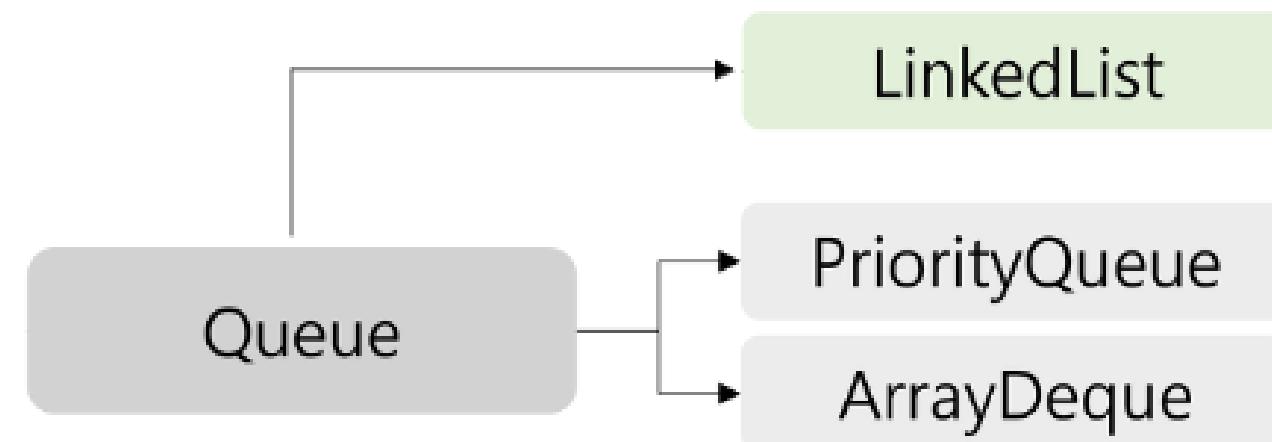
- 빠른 삽입 및 삭제: 각 요소가 이전 요소와 다음 요소를 가리키는 링크로 연결되어 있으므로, 중간에 요소를 추가하거나 삭제하는 경우에도 다른 요소를 이동시킬 필요가 없습니다.
- 메모리 동적 할당: 요소들이 포인터로 연결되어 있기 때문에 요소의 추가나 삭제에 따른 메모리 재할당이 불필요합니다.

단점

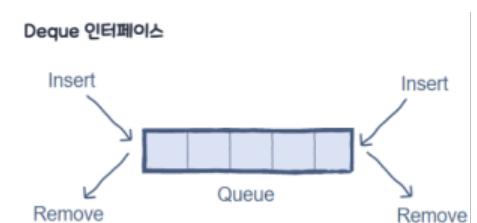
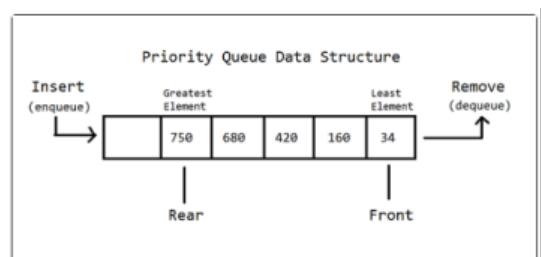
- 메모리 사용량이 높음: 각 노드가 데이터와 다음 노드에 대한 참조를 가지고 있기 때문에, 추가적인 포인터 메모리를 사용합니다.
- 순차 접근 성능이 떨어짐: 인접한 메모리에 저장되어 있지 않기 때문에 순차적인 접근 성능이 떨어집니다. 캐시 지역성이 낮아서 순차 접근 시 성능 저하가 발생할 수 있습니다.



Queue 인터페이스



```
public interface Queue<E> extends Collection<E> {
```

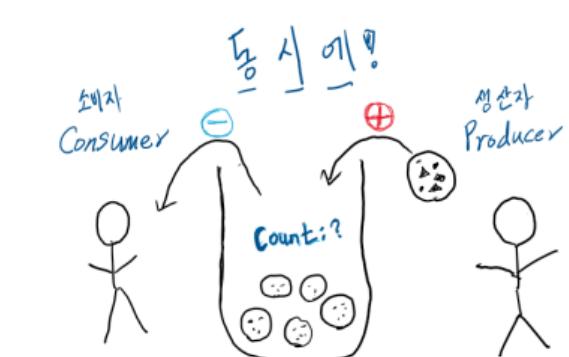


Queue 인터페이스

- 선입선출 FIFO(First-In-First-Out) 구조
- 특정 상황에 맞춘 우선순위 큐 제공 (PriorityQueue)

주요 메서드

- offer(E e):** 요소를 큐의 끝에 추가
- element():** 큐의 헤드를 반환합니다. 큐가 비어 있는 경우 예외
- poll():** 큐의 헤드 요소를 제거하고 반환
- peek():** 큐의 헤드 요소를 제거하지 않고 반환



Queue는 데이터 스트림 처리 및 생산자 소비자 문제(동시성 제어) 문제 해결에 자주 사용된다

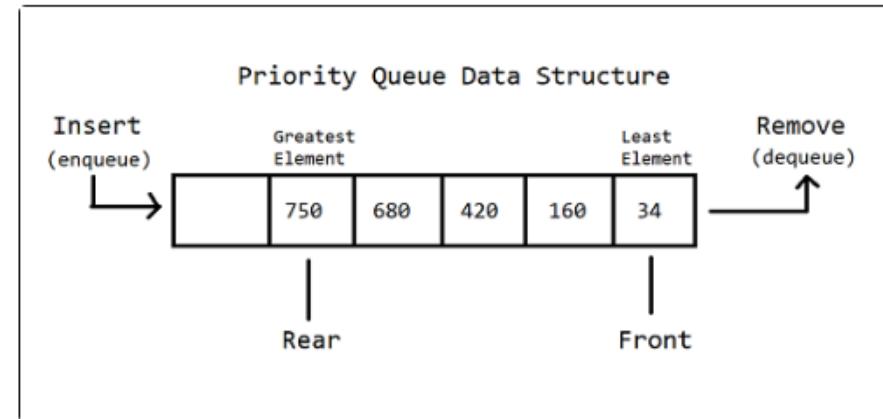
생산자 소비자 문제

생산자는 데이터를 만들어 버퍼에 저장하고(채워나가고),
소비자는 버퍼에 있는 데이터를 꺼내 소비하는(비우는)

프로세스 버퍼는 공유 자원이므로 버퍼에 대한 접근 즉, 저장하고 꺼내는 일들이 상호 배제 되어야 함

Queue 구현 클래스

PriorityQueue 클래스



1

PriorityQueue

특징

- 우선 순위를 가지는 큐
- FIFO 순서 x
- 중복 요소 o

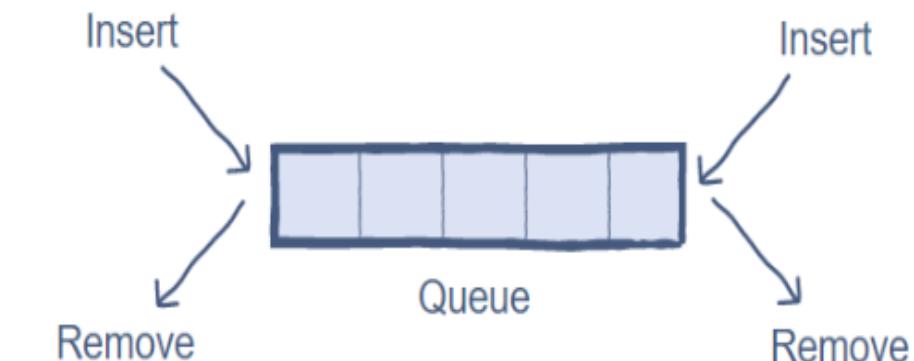
장점

- 우선순위 기반 자동 정렬 : 요소를 추가할 때마다 우선순위에 따라 자동으로 정렬
- 빠른 요소 삽입 및 삭제 : 요소를 이진 힙(binary heap)이라는 자료구조를 사용하여 저장하기 때문에, 요소의 삽입 및 삭제가 빠릅니다.

단점

- 동일한 우선순위 요소 처리의 제약 : 요소의 우선순위가 동일한 경우 처리 순서를 보장 x.
- 동기화 필요 : 스레드 안전성을 보장하지 않기 때문에 멀티스레드 환경에서 사용할 때 동기화가 필요합니다.

Deque 인터페이스



2

Deque

특징

- 양쪽으로 넣고 빼는 것이 가능한 큐
- FIFO 순서 x
- 중복 요소 o

장점

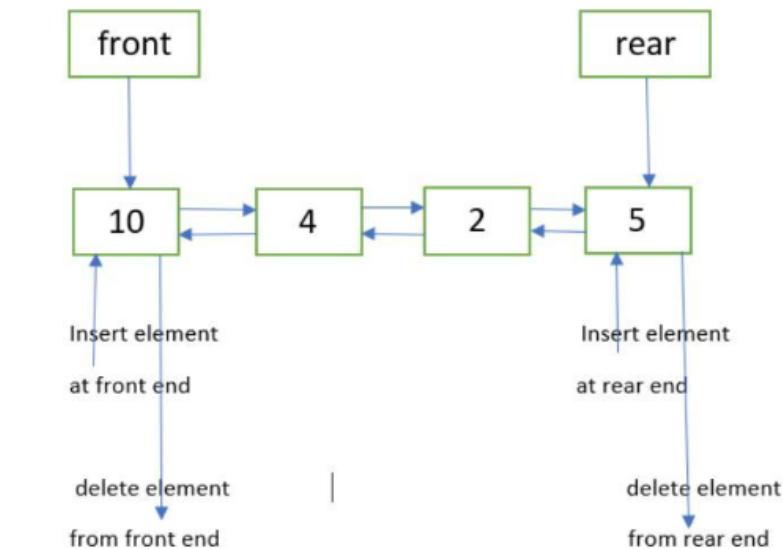
- 양쪽 끝에서의 삽입 및 삭제 : 큐(Queue)와 스택(Stack)의 기능을 모두 갖추고 있어서 다양한 용도로 사용

단점

- 동적 크기 조절의 오버헤드 : 크기를 동적으로 조절할 때 배열을 재할당하거나 요소들을 복사하는 오버헤드가 발생할 수 있습니다.

큐(queue)는 데이터를 꺼낼 때 항상 첫 번째 저장된 데이터를 삭제하므로, ArrayList와 같은 배열 기반의 컬렉션 클래스를 사용한다면, 데이터를 꺼낼 때마다 빈 공간을 채우기 위해 데이터의 이동 & 복사가 발생하므로 비효율적이다. 그래서 큐는 ArrayList보다 데이터의 추가/삭제가 용이한 LinkedList로 구현하는 것이 적합하다.

LinkedList 클래스



3

LinkedList

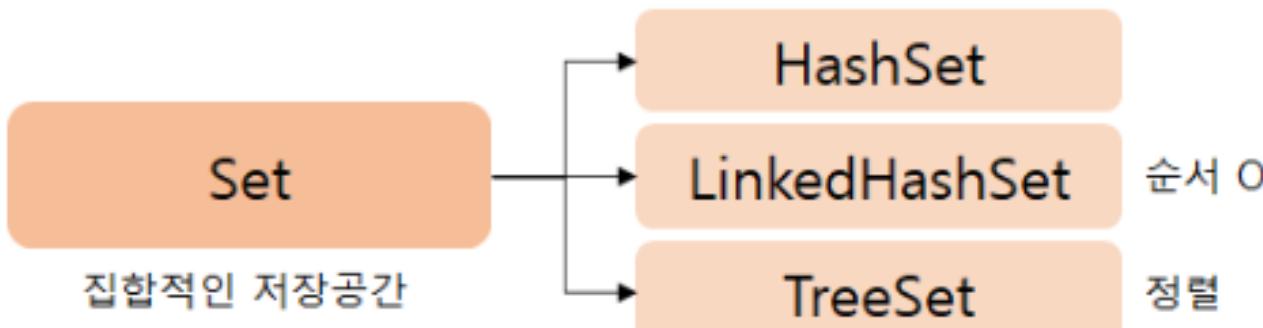
특징

- LinkedList는 List를 구현하며, Queue 도 구현합니다

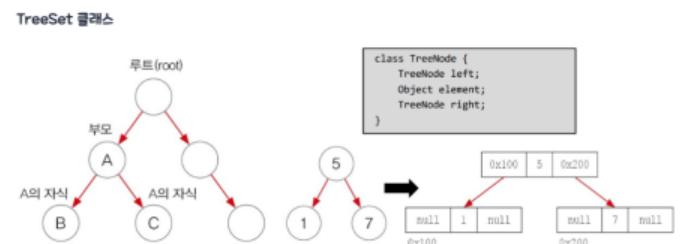
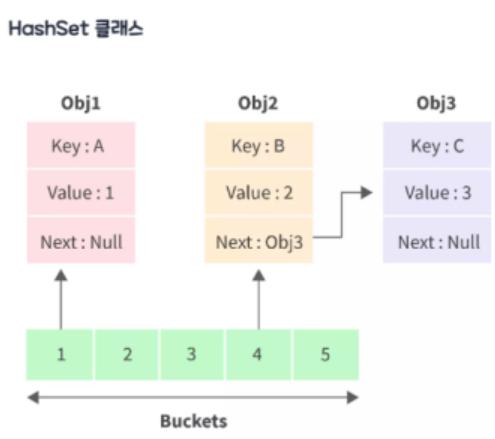
장단점은 리스트와 동일합니다.



Set 인터페이스



```
public interface Set<E> extends Collection<E> {
```



Set 인터페이스

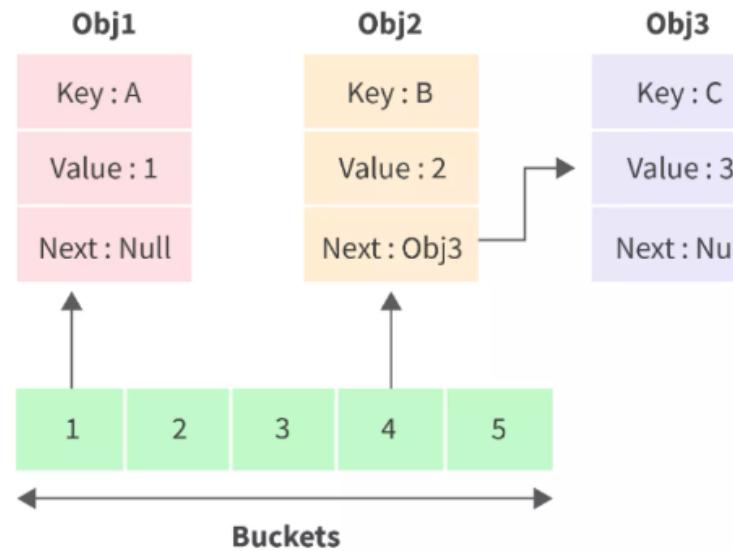
- 중복을 허용하지 않고 순서를 유지하지 않는 데이터의 집합 리스트
- 동일한 요소를 여러 번 추가해도 내부적으로는 **하나의 요소만** 유지
- 순서 자체가 없으므로 get(index) 메서드도 존재하지 않는다
- 중복 저장이 불가능하므로 **null 값 또한 하나만 저장** 가능하다.
- 효율적인 데이터 구조를 사용하여 검색 속도가 빠릅니다.

주요 메서드

- add(E e): 요소를 세트에 **추가**, 요소가 이미 세트에 있으면 false 반환
- remove(Object o): 지정된 요소를 세트에서 **제거합니다**.
- contains(Object o): 세트에 지정된 요소가 있는지 **여부를 확인합니다**.

Set 구현 클래스

HashSet 클래스



1

HashSet

특징

- 해시 테이블을 사용하여 요소 저장

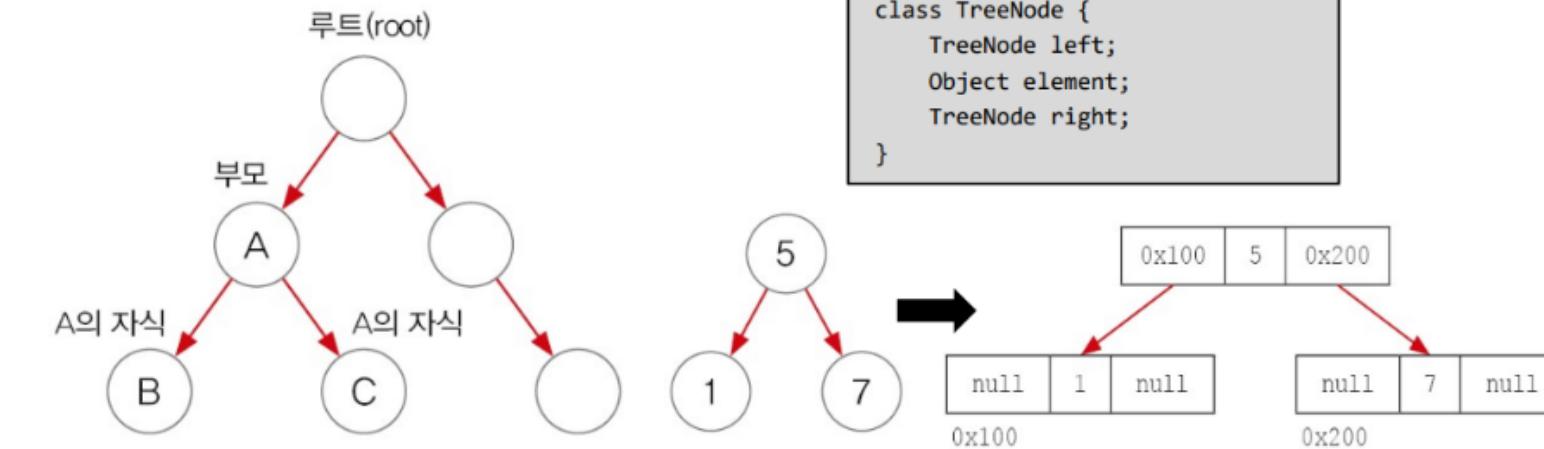
장점

- 가장 빠른 임의 검색 접근 속도 : 내부적으로 해시 테이블을 사용하여 요소를 저장하므로, 평균적으로 $O(1)$ 의 시간 복잡도로 요소를 (추가, 삭제, 검색)할 수 있습니다.
- 중복된 요소를 허용하지 않음 : 집합(Set) 자료구조의 특성을 가지므로, 중복된 요소를 자동으로 제거합니다.

단점

- 순서를 보장하지 않음 : 요소의 순서를 보장하지 않습니다.
- 복잡한 해시 함수 : 요소의 해시 코드를 기반으로 요소를 저장하므로, 잘못된 해시 코드를 구현하면 성능이 저하될 수 있습니다.

TreeSet 클래스



2

TreeSet

특징

- 이진 검색 트리를 사용하여 요소 저장

- 정렬된 순서로 요소를 유지

- 정렬, 검색, 범위 검색에 높은 성능

- 요소의 정렬 순서는 요소의 자연 순서 또는 사용자가 지정한 Comparator에 따라 결정

장점

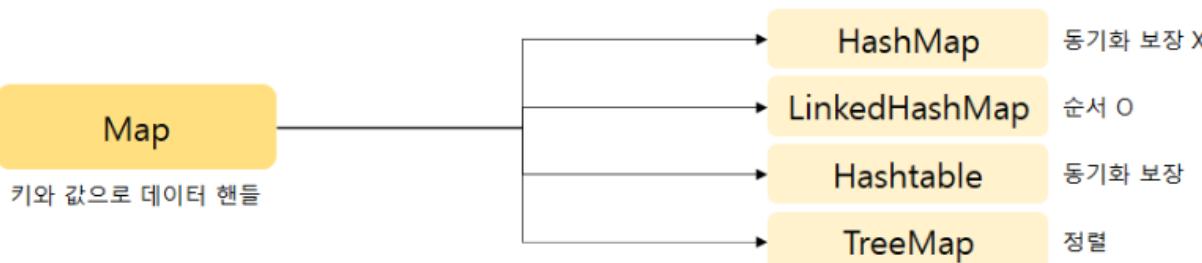
- 중복된 요소를 허용하지 않음 : 중복된 요소를 자동으로 제거합니다.

- 자동 정렬 : 내부적으로 이진 검색 트리를 사용하여 요소를 저장하므로, 요소들이 항상 정렬된 상태로 유지됩니다.

단점

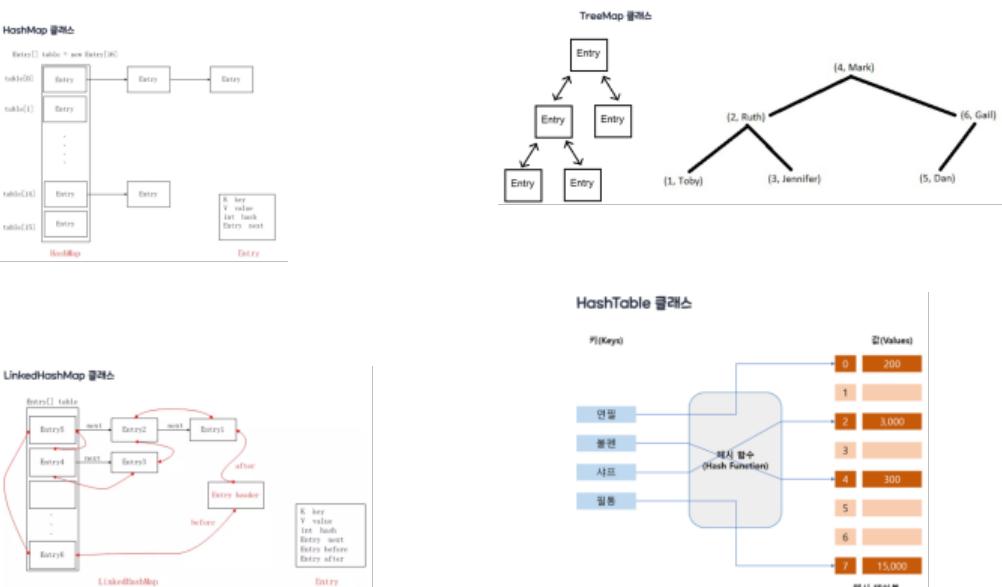
- 비교 객체 필요 : TreeSet은 요소들을 정렬하기 위해 요소의 비교가 필요합니다. 요소가 Comparable 인터페이스를 구현하거나 Comparator를 제공해야 합니다.
- 메모리 사용량: TreeSet은 이진 검색 트리 구조를 유지하기 위해 추가적인 포인터와 메모리를 사용합니다.

Map 인터페이스



```

public interface Map<K, V> {
    // Query Operations
}
  
```



Map 인터페이스

- 키(Key)와 값(value)의 쌍으로 연관지어 이루어진 데이터의 집합
- 값(value)은 중복되서 저장될수 있지만, 키(key)는 유일하다
- 키를 통해 값을 효율적으로 검색

주요 메서드

- containsKey(): 지정된 키가 맵에 존재하는지 확인
- containsValue(): 지정된 값이 맵에 하나 이상 존재하는지 확인
- get(): 지정된 키에 매핑된 값을 반환
- isEmpty(): Collection이 비어있는지 확인
- put(): 지정된 키와 값을 맵에 매핑합니다. 이전에 동일한 키에 매핑된 값이 있다면 그 값을 반환합니다
- Collection<V> values(): 맵에 있는 모든 값을 Collection 형태로 반환
- Set<K> keySet(): 맵에 있는 모든 키들을 Set 형태로 반환

Map 인터페이스의 메소드를 보면,

키(key)을 반환할때 Set 인터페이스 타입으로 반환하고,
값(value)을 반환할때 Collection 타입으로 반환하는걸 볼 수 있다.

Map 인터페이스에서

값(value)은 중복을 허용하기 때문에 Collection 타입으로 반환하고,
키(key)는 중복을 허용하지 않기 때문에 Set 타입으로 반환하는 것이다.

Map.Entry 인터페이스

```

interface Entry<K, V> {
    Returns the key corresponding to
    반환: the key corresponding to
    던지기: IllegalStateException –
            exception if the entry has

```

```

public interface Map {
    ...
    interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
        boolean equals(Object o);
        int hashCode();
        ...
    }
    ...
}

```



Map.Entry 인터페이스

- Map.Entry 인터페이스는 Map 인터페이스 안에 있는 내부 인터페이스이다.
- Map에 저장되는 key - value 쌍의 Node 내부 클래스가 이를 구현한다.
- Map 자료구조를 보다 객체지향적인 설계를 하도록 유도하기 위한 것이다.

주요 메서드

- containsKey():** 지정된 키가 맵에 존재하는지 확인
- containsValue():** 지정된 값이 맵에 하나 이상 존재하는지 확인
- get():** 지정된 키에 매핑된 값을 반환

활용

```

// Map 생성
Map<String, Integer> map = new HashMap<>();
map.put("apple", 10);
map.put("banana", 20);
map.put("cherry", 30);

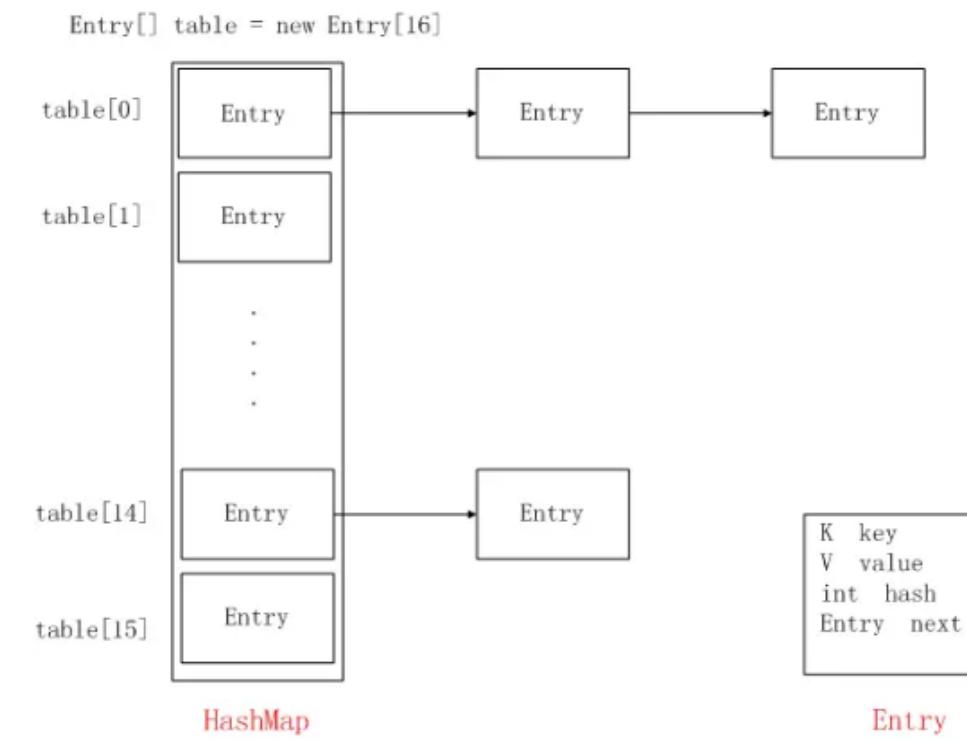
// Map의 모든 Entry 출력
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
}

// 특정 키의 값을 변경
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    if (entry.getKey().equals("banana")) {
        entry.setValue(25);
    }
}

```

Map 구현 클래스_1

HashMap 클래스



1

HashMap

특징

- 배열과 연결이 결합된 Hashing 형태 (**연관 배열 구조**)
- **키와 값을 하나의 데이터로 저장**

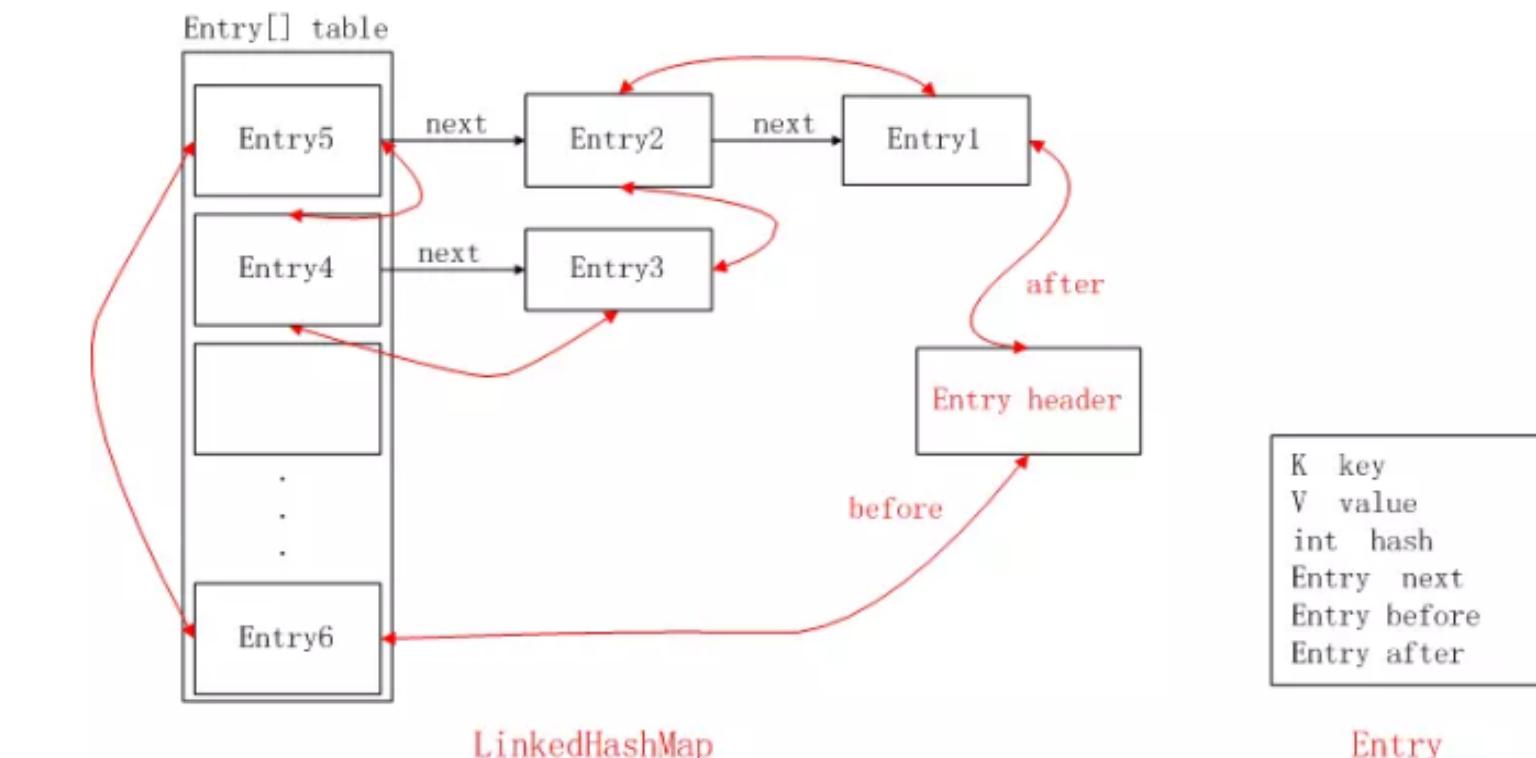
장점

- 유연한 키 타입 : 키로 사용할 객체의 타입에 제약이 없습니다.
- 빠른 데이터 접근: HashMap은 내부적으로 **해시 테이블을 사용**하여 데이터를 저장하므로, 평균적으로 **O(1)의 시간 복잡도**로 데이터를 삽입, 삭제, 검색할 수 있습니다.

단점

- 순서 보장 안 됨: **HashMap은 삽입된 순서를 보장하지 않습니다.**
- 메모리 사용량: 해시 테이블을 유지하기 위해 **추가적인 메모리 오버헤드**가 발생

LinkedHashMap 클래스



2

LinkedHashMap

특징

- **HashMap을 상속하기 때문에 흡사, Entry들이 연결 리스트를 구성 데이터의 순서를 보장**
- **들어온 순서대로 순서를 가진다.**

장점

- 삽입 순서 유지: **LinkedHashMap은 요소가 추가된 순서를 유지합니다.**

단점

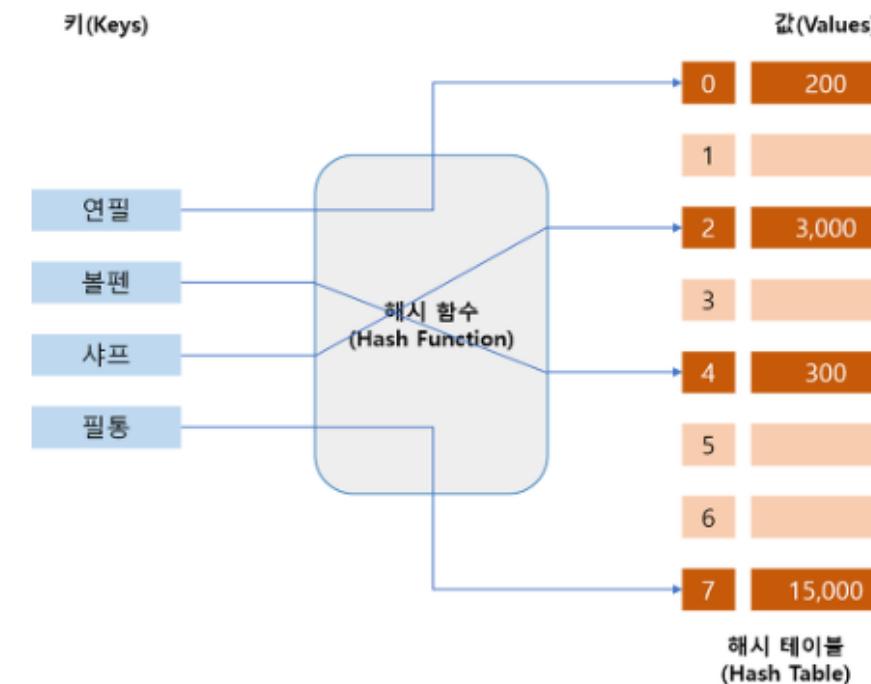
- 동기화되지 않음: 기본적으로 **LinkedHashMap은 스레드 안전하지 않습니다**



O(1) 시간 복잡도는 알고리즘의 실행 시간이 입력 크기와 무관하게 일정한 경우를 의미합니다.

Map 구현 클래스_2

HashTable 클래스



3 HashTable

특징

- Map 인터페이스의 오래된 구현체 중 하나
- 키와 값의 쌍을 저장

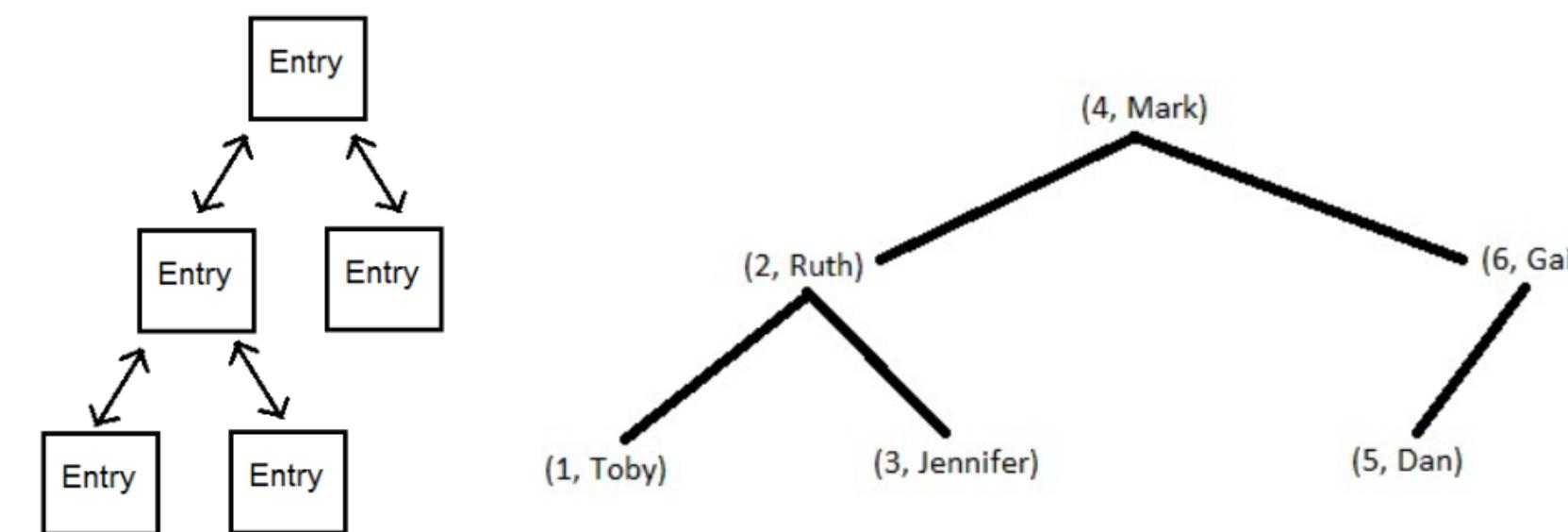
장점

- 스레드 안전성: 모든 메서드가 동기화되어 있어 여러 스레드가 동시에 접근할 때 안전합니다.

단점

- NULL 키와 값 허용 안 됨: NULL 키와 값을 허용하지 않음

TreeMap 클래스



4 TreeMap

특징

- 이진 검색 트리의 형태로 키와 값의 쌍으로 이루어진 데이터를 저장
- Key 값을 기준으로 정렬

장점

- 자동 정렬: TreeMap은 내부적으로 이진 검색 트리를 사용하여 요소를 저장하므로, 키가 항상 정렬된 상태로 유지됩니다. (숫자 → 알파벳 대문자 → 알파벳 소문자 → 한글)

단점

- 동기화 부족: 기본적으로 TreeMap은 스레드 안전하지 않습니다.

이진 검색 트리는 각 노드가 최대 두 개의 자식 노드를 가지며,
각 노드의 왼쪽 자식 노드는 해당 노드의 키보다 작고
오른쪽 자식 노드는 해당 노드의 키보다 큰 이진 트리입니다.



Map 구현 클래스_3

5 Properties

- Properties(String, String)의 형태로 저장하는 단순화된 key-value 컬렉션
- 주로 애플리케이션의 환경 설정과 관련된 속성 파일인 .properties 를 설정하는데 사용

root-context.xml

```
<!-- 설정한 property 파일을 스프링 설정(xml) 파일과 java, jsp 소스 코드 상에서 properties
<context:property-placeholder location="classpath:config.properties" />
| You, 2023-11-18 오전 11:45 • Uncommitted changes
<!-- DB 세팅 -->
<bean id="dataSource" class="org.apache.ibatis.datasource.pooled.PooledDataSource">
    <property name="driver" value="${db.driver}"></property>
    <property name="url" value="${db.url}"></property>
    <property name="username" value="${db.username}"></property>
    <property name="password" value="${db.password}"></property>
```

config.properties

```
#####
#DB Connection#####
#db.url=jdbc:log4jdbc:postgresql://127.0.0.1:31215/HN-Smar
#db.url=jdbc:log4jdbc:postgresql://192.168.0.204:31215/HN-
db.url=jdbc:log4jdbc:postgresql://112.216.98.130:31204/HN-
#db.url=jdbc:log4jdbc:postgresql://110.16.5.237:5432/HN-Sma
db.username=postgres
db.password=daeho1990!
db.driver=net.sf.log4jdbc.sql.jdbcapi.DriverSpy
#####
```

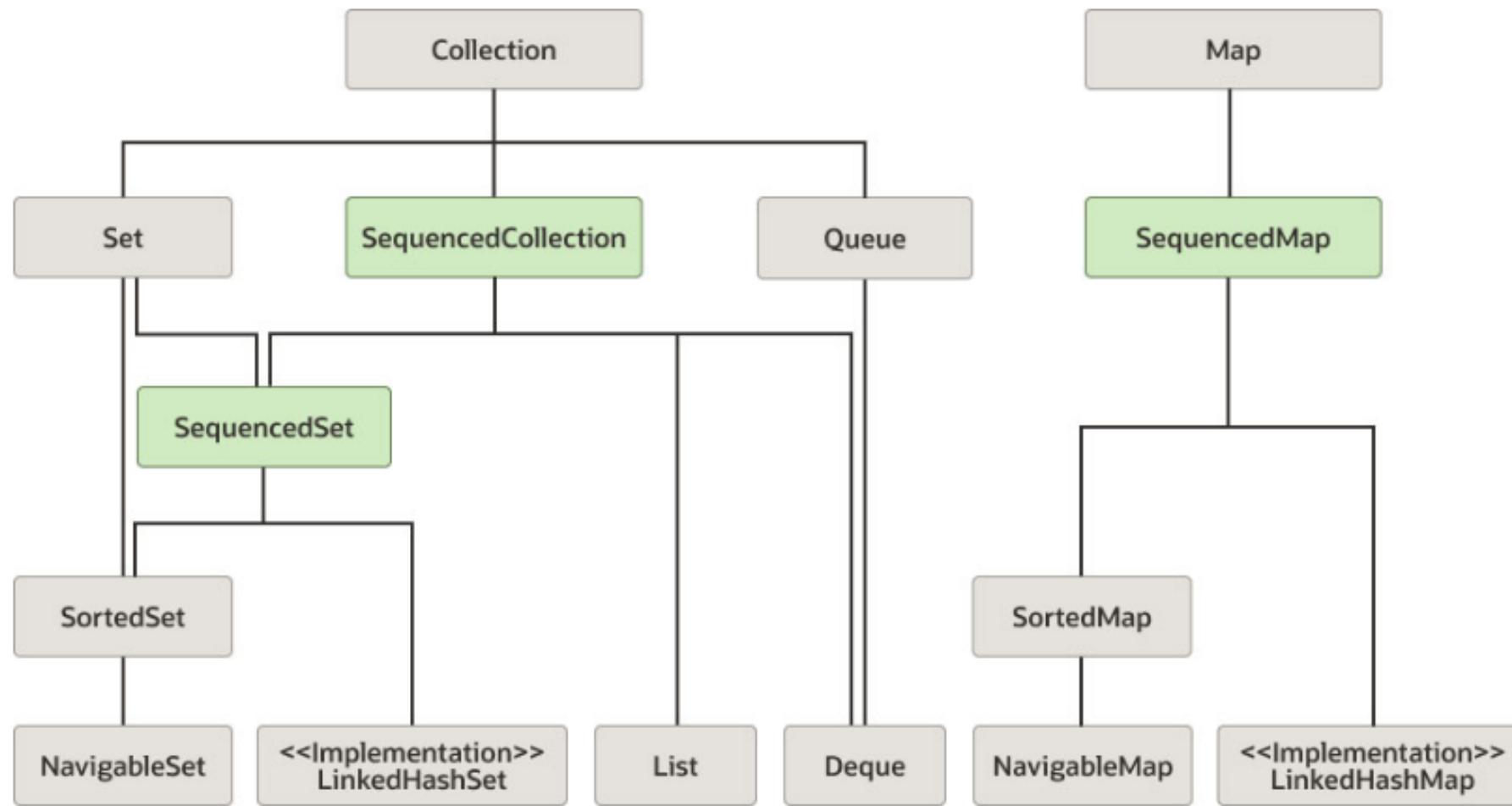


root-context.xml은 스프링 프레임워크에서 사용되는 XML 설정 파일 중 하나입니다.
일반적으로 스프링 애플리케이션의 루트 컨텍스트(root context)에 대한 설정을 담당합니다.
루트 컨텍스트는 모든 서블릿 컨텍스트(servlet context)가 공유하는 빈(bean) 및 설정을 포함합니다.



Sequenced Collection

```
public interface SequencedCollection<E> extends Collection<E> {
```



Sequenced Collection (JDK21 기준)

정렬을 제공하는 Collection 자료구조들의
(첫, 마지막) 데이터로의 접근, 정렬 및 역정렬에 대한
표준화된 API를 제공하여
기존 Collection의 문제점을 보완하기 위해 도입

특징

순서 보장: 요소들이 **특정한 순서**를 가지고 저장됩니다.
양방향 접근: 요소를 **앞쪽과 뒤쪽에서 모두 접근**

주요 메서드

- **getFirst():** 첫 번째 요소를 반환합니다.
- **getLast():** 마지막 요소를 반환합니다.
- **removeFirst():** 첫 번째 요소를 제거하고 반환
- **removeLast():** 마지막 요소를 제거하고 반환

List는 순서가 보장되는 선형 자료구조이기 때문에 어떤 구현체이든 순서가 존재한다.

따라서 SequencedCollection이 List보다 상위의 인터페이스에 해당하는 것이다.

Jdk 21 기준 ArrayList와LinkedList 에서 Sequenced Collection 메서드 사용 가능하다

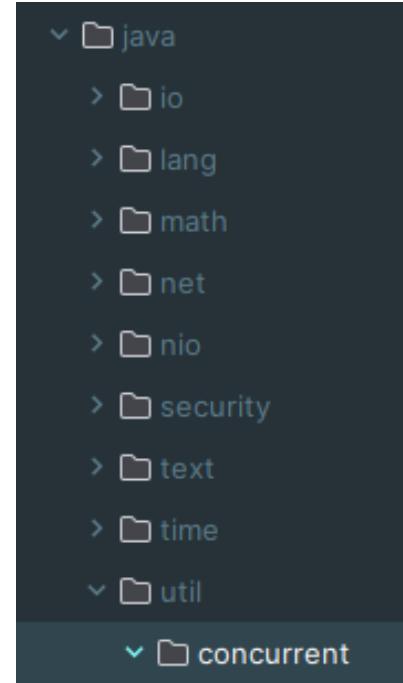


Concurrent Collections

동시성 문제 해결을 위한 컬렉션

- 멀티스레드 환경에서 동시 접근 문제를 해결하기 위한 컬렉션들
- 기본 컬렉션들의 한계와 **동기화 문제**

```
package java.util.concurrent;
```



Concurrent Collections 주요 클래스와 사용법

1

ConcurrentHashMap

고성능 동시 접근을 지원하는 해시맵

```
ConcurrentHashMap<String, Integer> conMap = new ConcurrentHashMap<>();
conMap.put("Apple", 10);
conMap.put("Banana", 20);
int value = conMap.get("Apple");
```

2

CopyOnWriteArrayList

읽기 작업이 많은 환경에서 유용한 리스트

```
CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>();
cowList.add("Apple");
cowList.add("Banana");
for (String fruit : cowList) {
    System.out.println(fruit);
}
```

3

BlockingQueue

생산자-소비자 패턴 구현에 유용한 큐

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();
try {
    queue.put("Apple");
    String item = queue.take();
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

Custom Collections

컬렉션 인터페이스(Collection, List, Set, Map)를 구현하여 **커스텀 컬렉션**을 만드는 방법
AbstractCollection 및 AbstractList와 같은 추상 클래스를 사용하여 구현 간소화

CustomOrderedSetList

List와 Set의 특성을 결합, 중복된 요소를 허용하지 않으며 요소가 추가된 순서를 유지

```
public class CustomOrderedSetList<E> extends AbstractList<E> {
    6개 사용 위치
    private final List<E> list;
    4개 사용 위치
    private final Set<E> set;

    0개의 사용 위치
    public CustomOrderedSetList() {
        list = new ArrayList<>();
        set = new HashSet<>();
    }

    @Override
    public boolean add(E e) {
        if (set.add(e)) { // HashSet을 이용하여 중복을 체크
            list.add(e);
            return true;
        }
        return false;
    }
}
```

```
CustomOrderedSetList<String> customList = new CustomOrderedSetList<>();
customList.add("Apple");
customList.add("Banana");
customList.add("Cherry");
customList.add("Apple"); // 중복된 요소 추가 시도

System.out.println("CustomOrderedSetList elements:");
for (String fruit : customList) {
    System.out.println(fruit);
}

customList.add(index: 1, element: "Date"); // 특정 위치에 요소 추가
System.out.println("index 1번에 Date 를 넣은 후");
for (String fruit : customList) {
    System.out.println(fruit);
}

customList.remove(index: 2); // 요소 제거
System.out.println("index 2번을 제거 한 후");
for (String fruit : customList) {
    System.out.println(fruit);
}
```

결과

```
CustomOrderedSetList elements:
Apple
Banana
Cherry
index 1번에 Date 를 넣은 후
Apple
Date
Banana
Cherry
index 2번을 제거 한 후
Apple
Date
Cherry
```

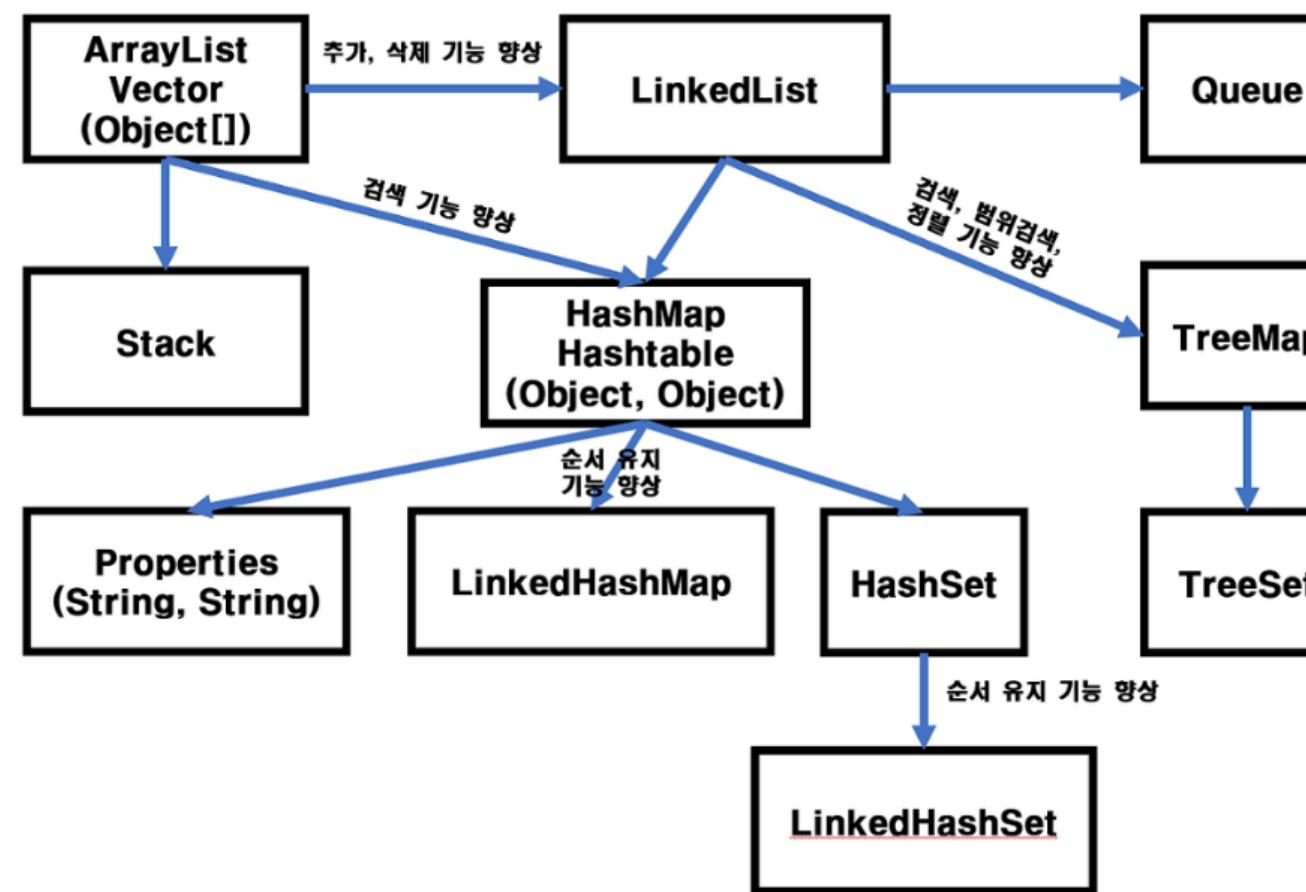
CustomOrderedSetList는
중복된 요소를 허용하지 않으며,
요소가 추가된 순서를 유지합니다.
이는 List와 Set의 장점을 결합한 커스텀 컬렉션입니다.

결론

자바 컬렉션 프레임워크는 자바 개발자에게 매우 중요한 도구이지만, 그만큼 다양하고 복잡합니다.

모든 컬렉션을 외우는 것은 현실적이지 않습니다.

대신, 어떤 상황에서 어떤 컬렉션을 사용해야 하는지를 이해하는 것이 중요합니다.



컬렉션 선택

- List:

- 순서가 있는 요소의 집합을 유지하고자 할 때 사용합니다.
- 중복된 요소를 허용하고, 인덱스로 요소에 접근할 수 있습니다.
- 데이터를 자주 추가하거나 삭제해야 하는 경우 사용합니다.
- ArrayList 또는 LinkedList 등을 사용할 수 있습니다.

- Set:

- 중복된 요소를 허용하지 않고, 요소의 순서를 보장하지 않습니다.
- 고유한 요소를 유지하고자 할 때 사용합니다.
- HashSet 또는 TreeSet 등을 사용할 수 있습니다.

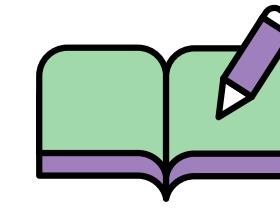
- Map:

- 키-값 쌍을 저장하는 데 사용됩니다.
- 키는 중복될 수 없고, 각 키는 하나의 값과 연결됩니다.
- 데이터베이스의 테이블과 유사한 구조를 갖습니다.
- HashMap, TreeMap, LinkedHashMap 등을 사용할 수 있습니다.

동시성 고려

멀티스레드 환경에서 작업할 때는 **동시성** 문제를 고려해야 합니다.

이 경우 **Concurrent** 접두사가 있는 컬렉션을 사용하거나,
동기화 메커니즘을 사용하여 스레드 안전성을 확보해야 합니다.



THANK YOU!

자료를 만들면서 참고한 사이트

- <https://gangnam-americano.tistory.com/41>
 - <https://kadosholy.tistory.com/117>
 - <https://inpa.tistory.com/entry/JCF-%F0%9F%A7%B1-Collections-Framework-%EC%A2%85%EB%A5%98-%EC%B4%9D%EC%A0%95%EB%A6%AC>
- ++ gpt-4o