Java의 정석

제 11 장

컬렉션 프레임웍 (collections framework)

2016. 3. 2

남궁성 강의

castello@naver.com

1.1 컬렉션 프레임웍(collections framework)이란?

▶ 컬렉션(collection)
- 여러 객체(데이터)를 모아 놓은 것을 의미

▶ 프레임웍(framework)
- 표준화, 정형화된 체계적인 프로그래밍 방식

▶ 컬렉션 프레임웍(collections framework)
- 컬렉션(다수의 객체)을 다루기 위한 표준화된 프로그래밍 방식
- 컬렉션을 쉽고 편리하게 다룰 수 있는 다양한 클래스를 제공
- java.util패키지에 포함. JDK1.2부터 제공

▶ 컬렉션 클래스(collection class)
- 다수의 데이터를 저장할 수 있는 클래스(예, Vector, ArrayList, HashSet)

1.2 컬렉션 프레임웍의 핵심 인터페이스

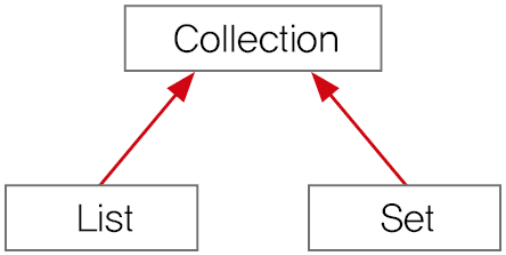
키(key)	값(value)
myId	1234
asdf	1234

List Set Map

인터페이스	특 징
List	순서가 있는 데이터의 집합. 데이터의 중복을 허용한다. 예) 대기자 명단
	구현클래스 : ArrayList, LinkedList, Stack, Vector 등
Set	순서를 유지하지 않는 데이터의 집합. 데이터의 중복을 허용하지 않는다. 예) 양의 정수집합, 소수의 집합
	구현클래스 : HashSet, TreeSet 등
Map	키(key)와 값(value)의 쌍(pair)으로 이루어진 데이터의 집합 순서는 유지되지 않으며, 키는 중복을 허용하지 않고, 값은 중복을 허용한다. 예) 우편번호, 지역번호(전화번호)
	구현클래스 : HashMap, TreeMap, Hashtable, Properties 등

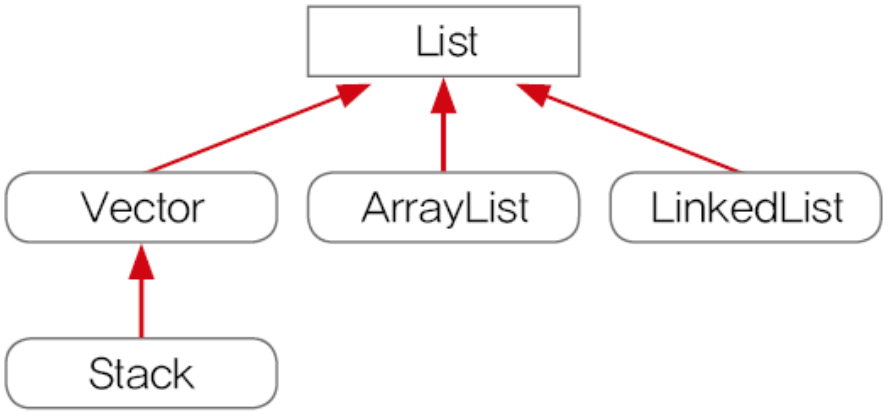
▲ 표 11-1 컬렉션 프레임웍의 핵심 인터페이스와 특징

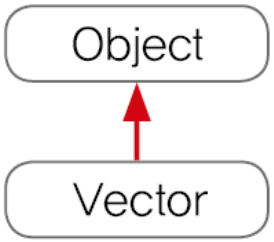
1.3 Collection인터페이스의 메서드



메서드	설 명
boolean add(Object o) boolean addAll(Collection c)	지정된 객체(o) 또는 Collection(c) 의 객 한다.
void clear( )	Collection의 모든 객체를 삭제한다.
boolean contains(Object o) boolean containsAll(Collection c)	지정된 객체(o) 또는 Collection의 객체들이 Collection에 포함되어 있는지 확인한다.
boolean equals(Object o)	동일한 Collection인지 비교한다.
int hashCode( )	Collection의 hash code를 반환한다.
boolean isEmpty( )	Collection이 비어있는지 확인한다.
Iterator iterator( )	Collection의 Iterator를 얻어서 반환한다.
boolean remove(Object o)	지정된 객체를 삭제한다.
boolean removeAll(Collection c)	지정된 Collection에 포함된 객체들을 삭제한다.
boolean retainAll(Collection c)	지정된 Collection에 포함된 객체만을 남기고 다른 객체들은 Collection 에서 삭제한다. 이 작업으로 인해 Collection에 변화가 있으면 true를 그렇지 않으면 false를 반환한다.
int size( )	Collection에 저장된 객체의 개수를 반환한다.
Object[ ] toArray( )	Collection에 저장된 객체를 객체배열(Object[ ])로 반환한다.
Object[ ] toArray(Object[ ] a)	지정된 배열에 Collection의 객체를 저장해서 반환한다.

1.4 List인터페이스의 메서드 - 순서O, 중복O

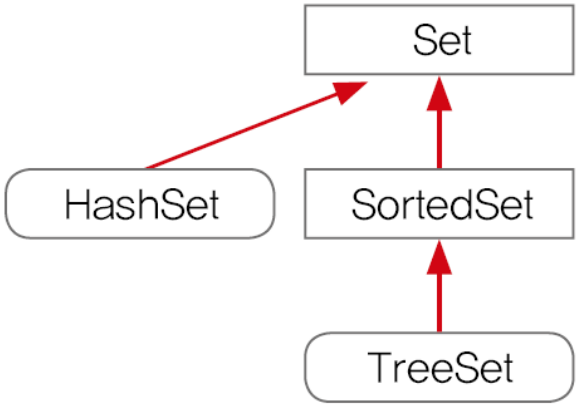




1.4 List인터페이스의 메서드 - 순서O, 중복O

메서드	설 명
void add(int index, Object element) boolean addAll(int index, Collection c)	지정된 위치(index)에 객체(element) 또는 컬렉션에 포함된 객체들을 추가한다.
Object get(int index)	지정된 위치(index)에 있는 객체를 반환한다.
int indexOf(Object o)	지정된 객체의 위치(index)를 반환한다. (List의 첫 번째 요소부터 순방향으로 찾는다.)
int lastIndexOf(Object o)	지정된 객체의 위치(index)를 반환한다. (List의 마지막 요소부터 역방향으로 찾는다.)
ListIterator listIterator( ) ListIterator listIterator(int index)	List의 객체에 접근할 수 있는 ListIterator를 반환한다.
Object remove(int index)	지정된 위치(index)에 있는 객체를 삭제하고 삭제된 객체를 반환한다.
Object set(int index, Object element)	지정된 위치(index)에 객체(element)를 저장한다
void sort(Comparator c)	지정된 비교자(comparator)로 List를 정렬한다.
List subList(int fromIndex, int toIndex)	지정된 범위(fromIndex부터 toIndex)에 있는 객체를 반환한다.

1.5 Set인터페이스 - 순서X, 중복X



* Set인터페이스의 메서드 - Collection인터페이스와 동일

메서드	설 명
boolean add(Object o)	지정된 객체(o)를 Collection에 추가한다.
void clear( )	Collection의 모든 객체를 삭제한다.
boolean contains(Object o)	지정된 객체(o)가 Collection에 포함되어 있는지 확인한다.
boolean equals(Object o)	동일한 Collection인지 비교한다.
int hashCode( )	Collection의 hash code를 반환한다.
boolean isEmpty( )	Collection이 비어있는지 확인한다.
Iterator iterator( )	Collection의 Iterator를 얻어서 반환한다.
boolean remove(Object o)	지정된 객체를 삭제한다.
int size( )	Collection에 저장된 객체의 개수를 반환한다.
Object[ ] toArray( )	Collection에 저장된 객체를 객체배열(Object[ ])로 반환한다.
Object[ ] toArray(Object[ ] a)	지정된 배열에 Collection의 객체를 저장해서 반환한다.

* 집합과 관련된 메서드(Collection에 변화가 있으면 true, 아니면 false를 반환.

메서드	설 명
boolean addAll(Collection c)	지정된 Collection(c)의 객체들을 Collection에 추가한다.(합집합)
boolean containsAll(Collection c)	지정된 Collection의 객체들이 Collection에 포함되어 있는지 확인한다.(부분집합)
boolean removeAll(Collection c)	지정된 Collection에 포함된 객체들을 삭제한다.(차집합)
boolean retainAll(Collection c)	지정된 Collection에 포함된 객체만을 남기고 나머지는 Collection에서 삭제한다.(교집합)

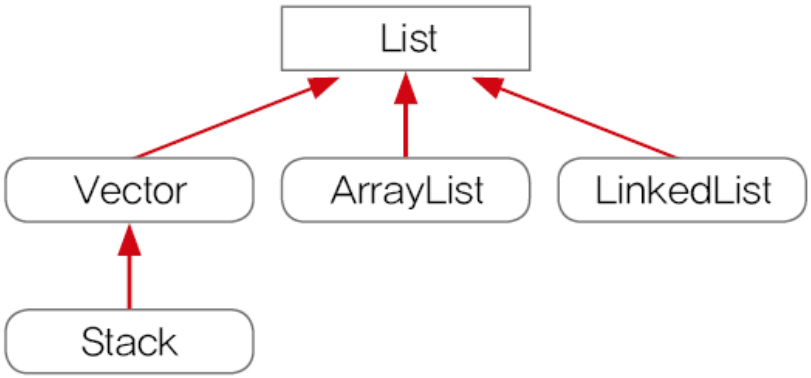
1.6 Map인터페이스의 메서드 - 순서X,중복(키X,값O)

키(key)	값(value)
myId	1234
asdf	1234

메서드	설 명
void clear( )	Map의 모든 객체를 삭제한다.
boolean containsKey(Object key)	지정된 key객체와 일치하는 Map의 key객체가 있는지 확인한다.
boolean containsValue(Object value)	지정된 value객체와 일치하는 Map의 value객체가 있는지 확인한다.
Set entrySet( )	Map에 저장되어 있는 key-value쌍을 Map.Entry타입의 객체로 저장한 Set으로 반환한다.
boolean equals(Object o)	동일한 Map인지 비교한다.
Object get(Object key)	지정한 key객체에 대응하는 value객체를 찾아서 반환한다.
int hashCode( )	해시코드를 반환한다.
boolean isEmpty( )	Map이 비어있는지 확인한다.
Set keySet( )	Map에 저장된 모든 key객체를 반환한다.
Object put(Object key, Object value)	Map에 value객체를 key객체에 연결(mapping)하여 저장한다.
void putAll(Map t)	지정된 Map의 모든 key-value쌍을 추가한다.
Object remove(Object key)	지정한 key객체와 일치하는 key-value객체를 삭제한다.
int size( )	Map에 저장된 key-value쌍의 개수를 반환한다.
Collection values( )	Map에 저장된 모든 value객체를 반환한다.

2.1 Vector와 ArrayList

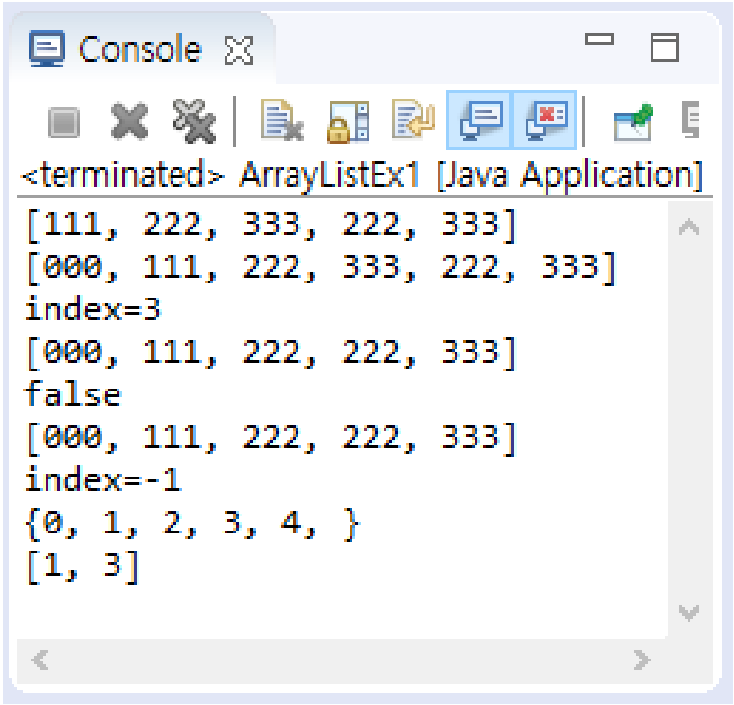
- ArrayList는 기존의 Vector를 개선한 것으로 구현원리와 기능적으로 동일
  Vector는 자체적으로 동기화처리가 되어 있으나 ArrayList는 그렇지 않다.
- List인터페이스를 구현하므로, 저장순서가 유지되고 중복을 허용한다.
- 데이터의 저장공간으로 배열을 사용한다.(배열기반)




```
public class Vector extends AbstractList
    implements List, RandomAccess, Cloneable, java.io.Serializable
{
        ...
    protected Object[] elementData;  // 객체를 담기 위한 배열
        ...
```

2.2 ArrayList의 사용예

```
ArrayList list = new ArrayList();  // JDK1.8현재 기본크기 10
list.add("111"); // void add(Object obj)
list.add("222");
list.add("333");
list.add("222"); // 중복 요소 추가가능
list.add(333);   // list.add(new Integer(333));
System.out.println(list);

list.add(0, "000");   // void add(int index, Object obj)
System.out.println(list);

System.out.println("index="+list.indexOf("333"));

list.remove("333");  // boolean remove(Object obj)
System.out.println(list);

System.out.println(list.remove("333"));
System.out.println(list);
System.out.println("index="+list.indexOf("333"));

for(int i=0;i<list.size();i++)
    list.set(i, i+"");  // Object set(int index, Object obj)

System.out.print("{");
for(int i=0;i<list.size();i++)
    System.out.print(list.get(i)+", "); // Object get(int index)
System.out.println("}");

for(int i=0;i<list.size();i++)
    list.remove(i);     // Object remove(int index)
System.out.println(list);
```



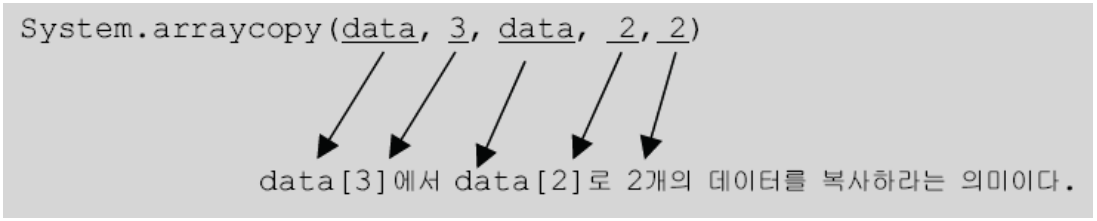

```
// 마지막 요소부터 차례대로 삭제
for(int i=list.size()-1;i>=0;i--)
    list.remove(i);
System.out.println(list);
```

2.3 ArrayList에 저장된 객체의 삭제과정(1/2)

- ArrayList에 저장된 세 번째 데이터(data[2])를 삭제하는 과정. list.remove(2);를 호출

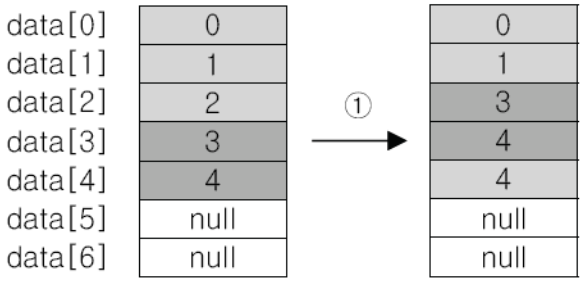
data[0]	0
data[1]	1
data[2]	2
data[3]	3
data[4]	4
data[5]	null
data[6]	null

① 삭제할 데이터 아래의 데이터를 한 칸씩 위로 복사해서 삭제할 데이터를 덮어쓴다.



2.3 ArrayList에 저장된 객체의 삭제과정(1/2)

- ArrayList에 저장된 세 번째 데이터(data[2])를 삭제하는 과정. list.remove(2);를 호출



① 삭제할 데이터 아래의 데이터를 한 칸씩 위로 복사해서 삭제할 데이터를 덮어쓴다.

```
System.arraycopy(data, 3, data, 2, 2)
```

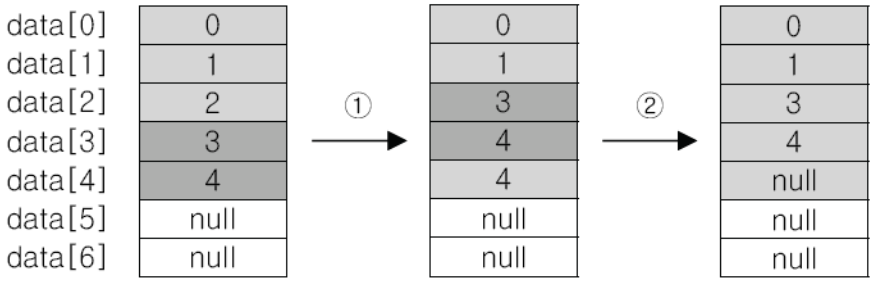
data[3]에서 data[2]로 2개의 데이터를 복사하라는 의미이다.

② 데이터가 모두 한 칸씩 이동했으므로 마지막 데이터는 null로 변경한다.

```
data[size-1] = null;
```

2.3 ArrayList에 저장된 객체의 삭제과정(1/2)

- ArrayList에 저장된 세 번째 데이터(data[2])를 삭제하는 과정. list.remove(2);를 호출



① 삭제할 데이터 아래의 데이터를 한 칸씩 위로 복사해서 삭제할 데이터를 덮어쓴다.

```
System.arraycopy(data, 3, data, 2, 2)
```

data[3]에서 data[2]로 2개의 데이터를 복사하라는 의미이다.

② 데이터가 모두 한 칸씩 이동했으므로 마지막 데이터는 null로 변경한다.

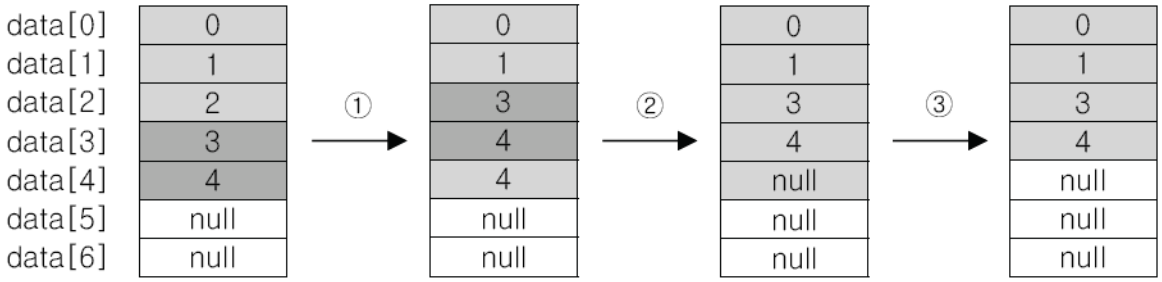
```
data[size-1] = null;
```

③ 데이터가 삭제되어 데이터의 개수가 줄었으므로 size의 값을 감소시킨다.

```
size--;
```

2.3 ArrayList에 저장된 객체의 삭제과정(1/2)

- ArrayList에 저장된 세 번째 데이터(data[2])를 삭제하는 과정. list.remove(2);를 호출



① 삭제할 데이터 아래의 데이터를 한 칸씩 위로 복사해서 삭제할 데이터를 덮어쓴다.

```
System.arraycopy(data, 3, data, 2, 2)
```

data[3]에서 data[2]로 2개의 데이터를 복사하라는 의미이다.

② 데이터가 모두 한 칸씩 이동했으므로 마지막 데이터는 null로 변경한다.

```
data[size-1] = null;
```

③ 데이터가 삭제되어 데이터의 개수가 줄었으므로 size의 값을 감소시킨다.

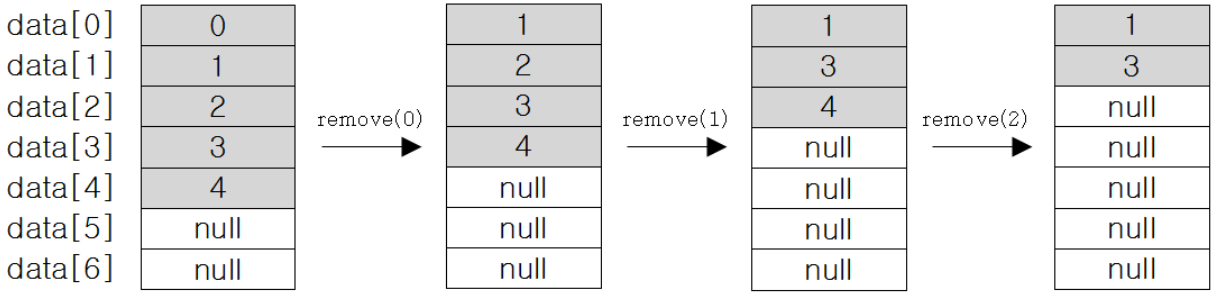
```
size--;
```

※ 마지막 데이터를 삭제하는 경우, ①의 과정(배열의 복사)은 필요없다.

2.3 ArrayList에 저장된 객체의 삭제과정(2/2)

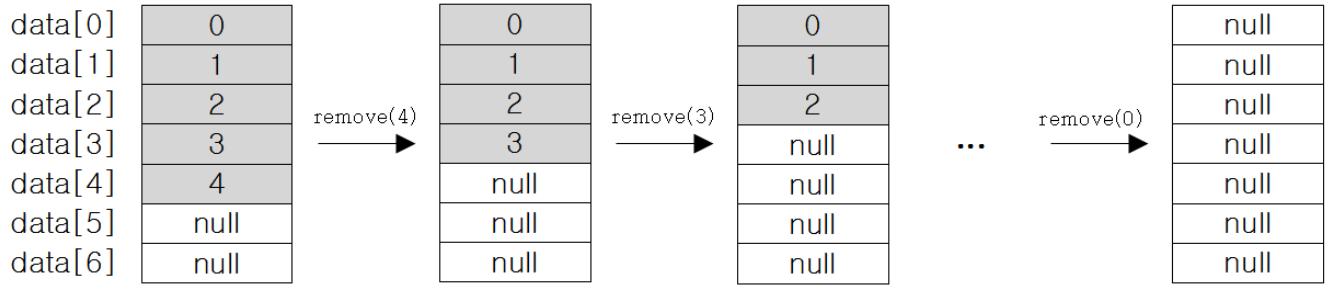
① ArrayList에 저장된 첫 번째 객체부터 삭제하는 경우(배열 복사 발생)

```
for(int i=0;i<list.size();i++)
    list.remove(i);
```



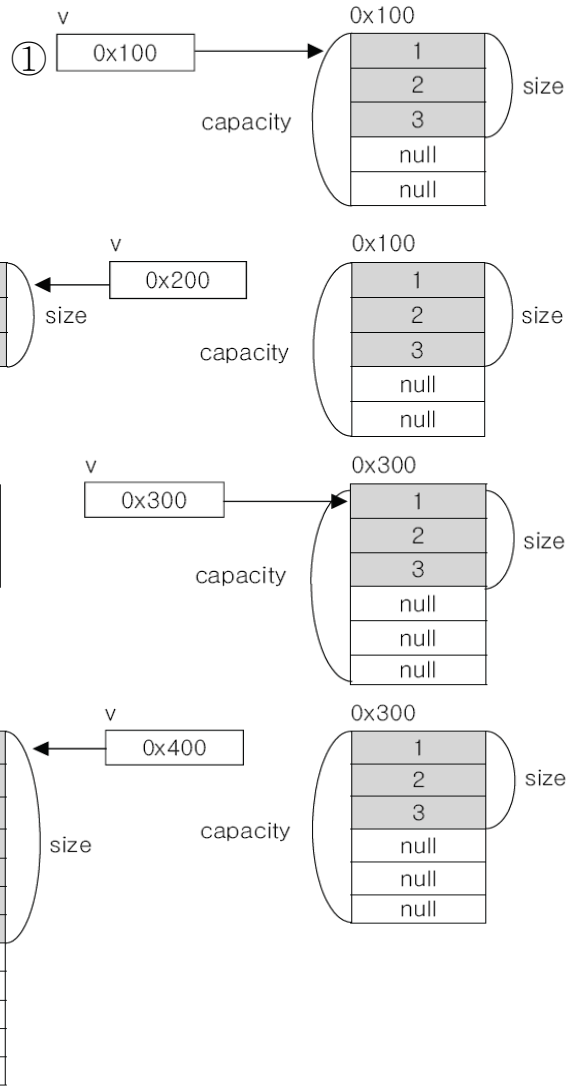
② ArrayList에 저장된 마지막 객체부터 삭제하는 경우(배열 복사 발생안함)

```
for(int i=list.size()-1;i>=0;i--)
    list.remove(i);
```



2.4 Vector의 크기(size)와 용량(capacity)

```
// 1. 용량(capacity)이 5인 Vector를 생성한다.
Vector v = new Vector(5);
v.add("1");
v.add("2");
v.add("3");

// 2. 빈 공간을 없앤다.(용량과 크기가 같아진다.)
v.trimToSize();

// 3. capacity가 6이상 되도록 한다.
v.ensureCapacity(6);

// 4. size가 7이 되게 한다.
v.setSize(7);

// 5. Vector에 저장된 모든 요소를 제거한다.
v.clear();
```



2.5 Vector를 직접 구현하기 - MyVector.java

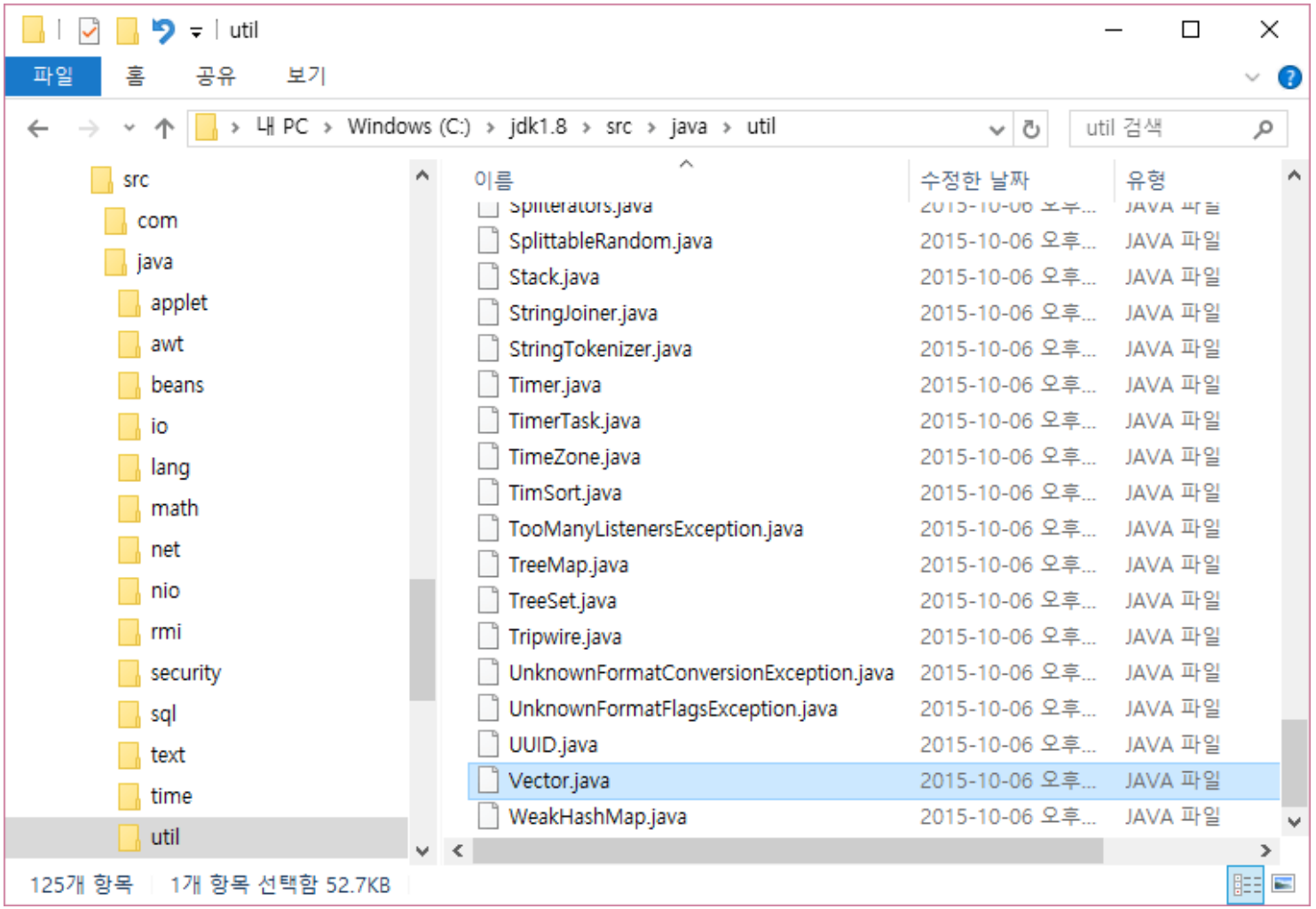
① 객체를 저장할 객체배열(objArr)과 크기(size)를 저장할 변수를 선언

② 생성자 MyVector(int capacity)와 기본 생성자 MyVector()를 선언

③ 메서드 구현

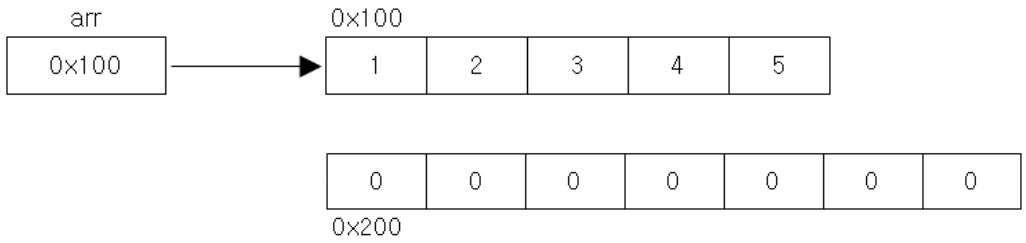
- int size() - 컬렉션의 크기(size, 객체배열에 저장된 객체의 개수)를 반환
- int capacity() - 컬렉션의 용량(capacity, 객체배열의 길이)을 반환
- boolean isEmpty() - 컬렉션이 비어있는지 확인
- void clear() - 컬렉션의 객체를 모두 제거(객체배열의 모든 요소를 null)
- Object get(int index) - 컬렉션에서 지정된 index에 저장된 객체를 반환
- int indexOf(Object obj) - 지정된 객체의 index를 반환(못찾으면 -1)
- void setCapacity(int capacity) - 컬렉션의 용량을 변경
- void ensureCapacity(int minCapacity) - 컬렉션의 용량을 확보
- Object remove (int index) - 컬렉션에서 객체를 삭제
- boolean add(Object obj) - 컬렉션에 객체를 추가

▮ 알아두면 좋아요! - Java API 소스보기(src.zip)



2.6 ArrayList의 장점과 단점

▶ 장점 : 배열은 구조가 간단하고 데이터를 읽는 데 걸리는 시간 (접근시간, access time)이 짧다.



▶ 단점 1 : 크기를 변경할 수 없다.

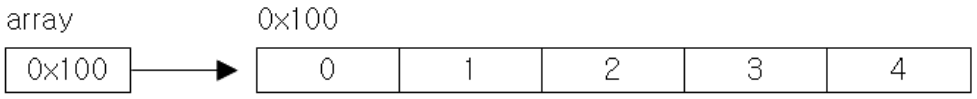
- 크기를 변경해야 하는 경우 새로운 배열을 생성 후 데이터를 복사해야함.
- 크기 변경을 피하기 위해 충분히 큰 배열을 생성하면, 메모리가 낭비됨.

▶ 단점 2 : 비순차적인 데이터의 추가, 삭제에 시간이 많이 걸린다.

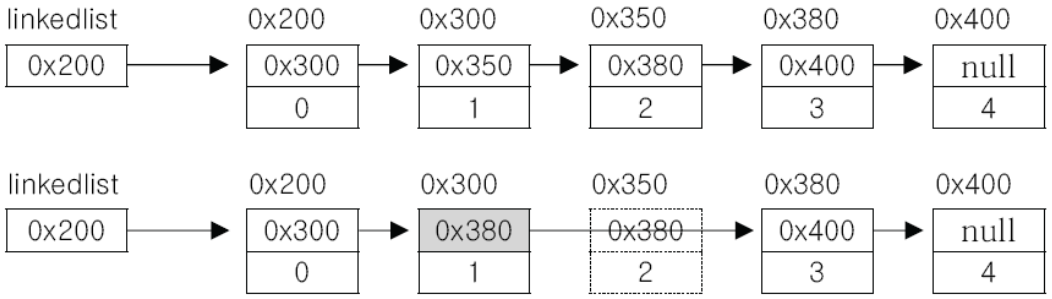
- 데이터를 추가하거나 삭제하기 위해, 다른 데이터를 옮겨야 함.
- 그러나 순차적인 데이터 추가(끝에 추가)와 삭제(끝부터 삭제)는 빠르다.

3.1 LinkedList - 배열의 단점을 보완

- 배열과 달리 링크드 리스트는 불연속적으로 존재하는 데이터를 연결(link)

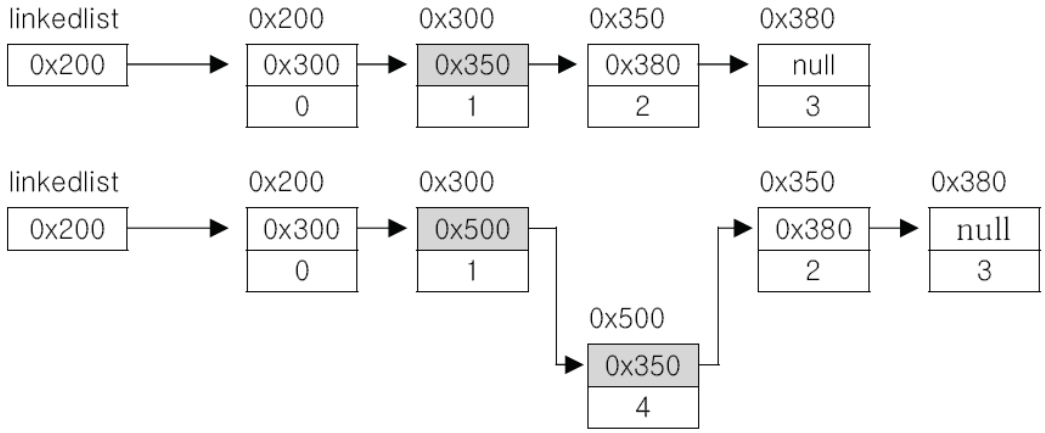


▶ 데이터의 삭제 : 단 한 번의 참조변경만으로 가능



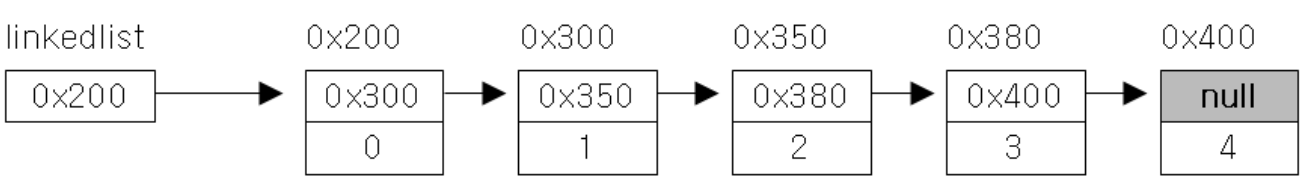

```
class Node {
    Node next;
    Object obj;
}
```

▶ 데이터의 추가 : 한번의 Node객체생성과 두 번의 참조변경만으로 가능



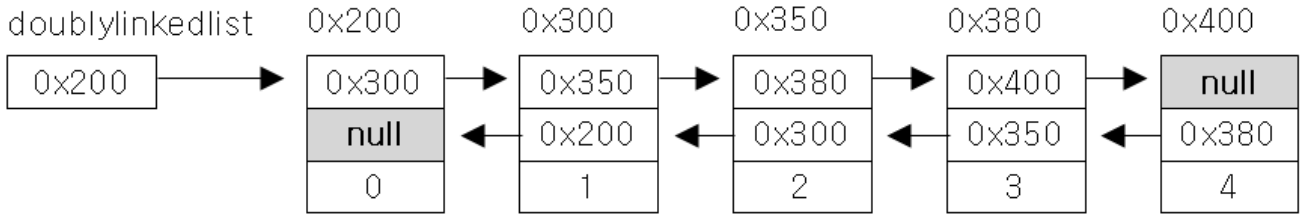
3.2 LinkedList – 이중 연결 리스트

▶ 링크드 리스트(linked list) – 연결리스트. 데이터 접근성이 나쁨



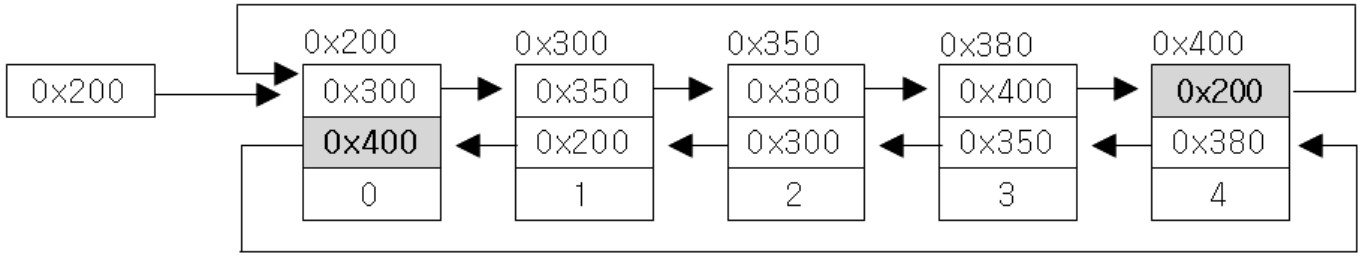

```
class Node {
    Node next;
    Object obj;
}
```

▶ 더블리 링크드 리스트(doubly linked list) – 이중 연결리스트, 접근성 향상




```
class Node {
    Node next;
    Node previous;
    Object obj;
}
```

▶ 더블리 써큘러 링크드 리스트(doubly circular linked list) – 이중 원형 연결리스트



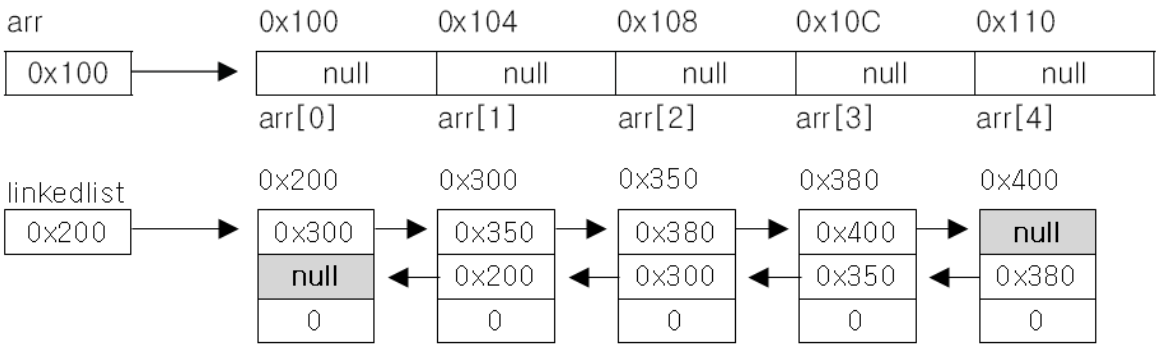
3.3 ArrayList vs. LinkedList - 성능 비교

① 순차적으로 데이터를 추가/삭제 - ArrayList가 빠름

② 비순차적으로 데이터를 추가/삭제 - LinkedList가 빠름

③ 접근시간(access time) - ArrayList가 빠름

n번째 데이터의 주소 = 배열의 주소 + (n-1) * 데이터 타입의 크기



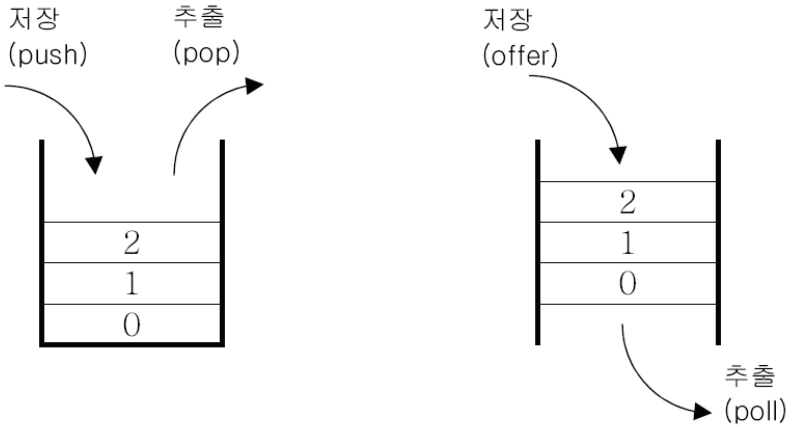

```
= 순차적으로 추가하기 =
ArrayList :406
LinkedList :606

= 순차적으로 삭제하기 =
ArrayList :11
LinkedList :46

= 중간에 추가하기 =
ArrayList :7382
LinkedList :31

= 중간에서 삭제하기 =
ArrayList :6694
LinkedList :380

= 접근시간테스트 =
ArrayList :1
LinkedList :432
```

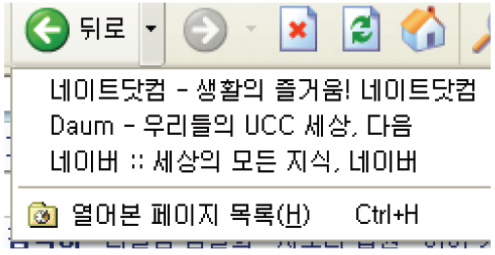
컬렉션	읽기(접근시간)	추가 / 삭제	비 고
ArrayList	빠르다	느리다	순차적인 추가삭제는 빠름. 비효율적인 메모리사용
LinkedList	느리다	빠르다	데이터가 많을수록 접근성이 떨어짐

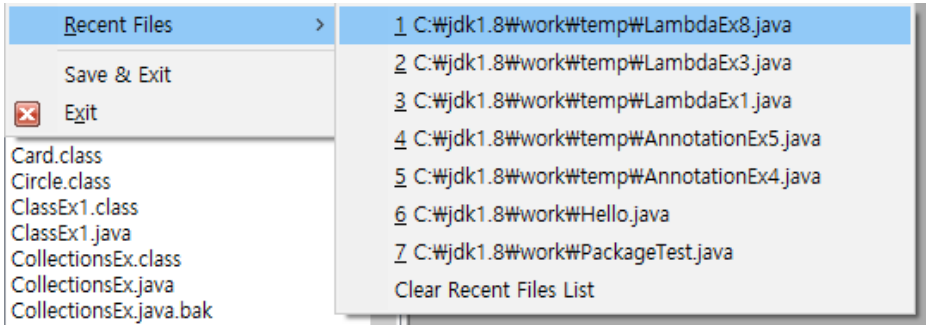
4.1 스택과 큐(Stack & Queue)

▶ 스택(Stack) : LIFO구조. 마지막에 저장된 것을 제일 먼저 꺼내게 된다.
- 수식계산, 수식괄호검사, undo/redo, 뒤로/앞으로(웹브라우저)

▶ 큐(Queue) : FIFO구조. 제일 먼저 저장한 것을 제일 먼저 꺼내게 된다.
- 최근 사용문서, 인쇄작업대기목록, 버퍼(buffer)







▌알아두면 좋아요! – 인터페이스를 구현한 클래스 찾기

java.util

Interface Queue<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

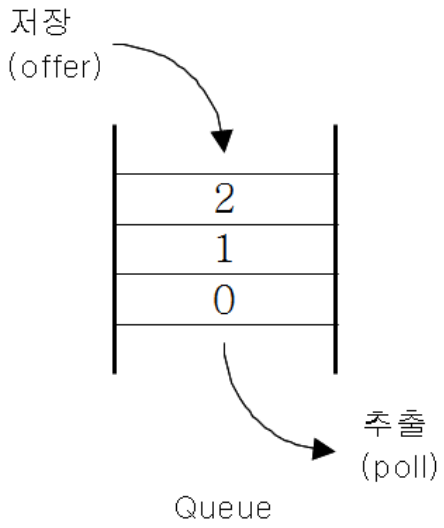
All Known Implementing Classes:

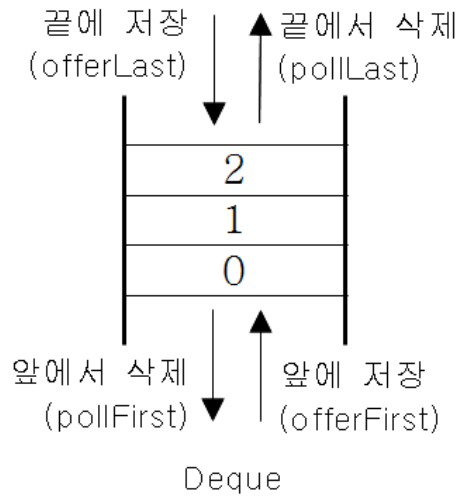
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

4.2 Queue의 변형 – Deque, PriorityQueue, BlockingQueue

▶ 덱(Deque) : Stack과 Queue의 결합. 양끝에서 저장(offer)과 삭제(poll) 가능

(구현클래스 : ArrayDeque, LinkedList)





▶ 우선순위 큐(PriorityQueue) : 우선순위가 높은 것부터 꺼냄(null 저장불가)

입력[3,1,5,2,4] -> 출력[1,2,3,4,5]

▶ 블락킹 큐(BlockingQueue) : 비어 있을 때 꺼내기와, 가득 차 있을 때 넣기를 지정된 시간동안 지연시킴(block) – 멀티쓰레드

5.1 Enumeration, Iterator, ListIterator

- 컬렉션에 저장된 데이터를 접근하는데 사용되는 인터페이스
- Enumeration은 Iterator의 구버젼
- ListIterator는 Iterator의 접근성을 향상시킨 것 (단방향 → 양방향)

메서드	설 명
boolean hasNext( )	읽어 올 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
Object next( )	다음 요소를 읽어 온다. next( )를 호출하기 전에 hasNext( )를 호출해서 읽어 올 요소가 있는지 확인하는 것이 안전하다.
void remove( )	next( )로 읽어 온 요소를 삭제한다. next( )를 호출한 다음에 remove( )를 호출해야한다.(선택적 기능)
void forEachRemaining(Consumer<? super E> action)	컬렉션에 남아있는 요소들에 대해 지정된 작업(action)을 수행한다. 람다식을 사용하는 디폴트 메서드.(JDK1.8부터 추가)

▲ 표11-12 Iterator인터페이스의 메서드

메서드	설 명
boolean hasMoreElements( )	읽어 올 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다. Iterator의 hasNext( )와 같다.
Object nextElement( )	다음 요소를 읽어 온다. nextElement( )를 호출하기 전에 hasMoreElements( )를 호출해서 읽어올 요소가 남아있는지 확인하는 것이 안전하다. Iterator의 next( )와 같다.

▲ 표11-13 Enumeration인터페이스의 메서드

5.2 Iterator

- 컬렉션에 저장된 요소들을 읽어오는 방법을 표준화한 것
- 컬렉션에 iterator()를 호출해서 Iterator를 구현한 객체를 얻어서 사용.

메서드	설 명
boolean hasNext( )	읽어 올 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
Object next( )	다음 요소를 읽어 온다. next( )를 호출하기 전에 hasNext( )를 호출해서 읽어 올 요소가 있는지 확인하는 것이 안전하다.
void remove( )	next( )로 읽어 온 요소를 삭제한다. next( )를 호출한 다음에 remove( )를 호출해야한다.(선택적 기능)
void forEachRemaining(Consumer<? super E> action)	컬렉션에 남아있는 요소들에 대해 지정된 작업(action)을 수행한다. 람다식을 사용하는 디폴트 메서드.(JDK1.8부터 추가)

```
Collection c = new ArrayList(); // 다른 컬렉션으로 변경할 때는 이 부분만 고치면 된다.
 ...
Iterator it = c.iterator();

while(it.hasNext()) {
    System.out.println(it.next());
}
```

```
public interface Collection {
    ...
    public Iterator iterator();
    ...
}
```

```
Map map = new HashMap();
   ...
Iterator list = map.entrySet().iterator();
```

```
Set eSet = map.entrySet();
Iterator list = eSet.iterator();
```

5.3 ListIterator - Iterator의 기능을 확장(상속)

- Iterator의 접근성을 향상시킨 것이 ListIterator이다.(단방향 →양방향)
- listIterator()를 통해서 얻을 수 있다.(List를 구현한 컬렉션 클래스에 존재)

```
public interface ListIterator extends Iterator {
    ...
}
```

```
public interface List extends Collection {
    ...
    ListIterator listIterator();
    ...
}
```

메서드	설 명
boolean hasNext( )	읽어 올 다음 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
boolean hasPrevious( )	읽어 올 이전 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
Object next( )	다음 요소를 읽어 온다. next( )를 호출하기 전에 hasNext( )를 호출해서 읽어 올 요소가 있는지 확인하는 것이 안전하다.
Object previous( )	이전 요소를 읽어 온다. previous( )를 호출하기 전에 hasPrevious( )를 호출해서 읽어 올 요소가 있는지 확인하는 것이 안전하다.
int nextIndex( )	다음 요소의 index를 반환한다.
int previousIndex( )	이전 요소의 index를 반환한다.
void add(Object o)	컬렉션에 새로운 객체(o)를 추가한다.(선택적 기능)
void remove( )	next( ) 또는 previous( )로 읽어 온 요소를 삭제한다. 반드시 next( )나 previous( )를 먼저 호출한 다음에 이 메서드를 호출해야한다.(선택적 기능)
void set(Object o)	next( ) 또는 previous( )로 읽어 온 요소를 지정된 객체(o)로 변경한다. 반드시 next( )나 previous( )를 먼저 호출한 다음에 이 메서드를 호출해야한다.(선택적 기능)

6.1 Arrays(1/3) - 배열을 다루기 편리한 메서드(static) 제공

1. 배열의 출력 - toString()

```
static String toString(boolean[] a)
static String toString(byte[] a)
static String toString(char[] a)
static String toString(short[] a)
static String toString(int[] a)
static String toString(long[] a)
static String toString(float[] a)
static String toString(double[] a)
static String toString(Object[] a)
```

2. 다차원 배열의 비교와 출력 - deepEquals(), deepToString(), equals()

```
int[]arr   = {0,1,2,3,4};
int[][] arr2D = {{11,12}, {21,22}};

System.out.println(Arrays.toString(arr)); // [0, 1, 2, 3, 4]
System.out.println(Arrays.deepToString(arr2D)); // [[11, 12], [21, 22]]
```

```
String[][] str2D  = new String[][]{{"aaa","bbb"},{"AAA","BBB"}};
String[][] str2D2 = new String[][]{{"aaa","bbb"},{"AAA","BBB"}};

System.out.println(Arrays.equals(str2D, str2D2));     // false
System.out.println(Arrays.deepEquals(str2D, str2D2)); // true
```

6.1 Arrays(2/3) - 배열을 다루기 편리한 메서드(static) 제공

3. 배열의 복사 - copyOf(), copyOfRange()

```
int[] arr = {0,1,2,3,4};
int[] arr2 = Arrays.copyOf(arr, arr.length); // arr2=[0,1,2,3,4]
int[] arr3 = Arrays.copyOf(arr, 3);          // arr3=[0,1,2]
int[] arr4 = Arrays.copyOf(arr, 7);          // arr4=[0,1,2,3,4,0,0]

int[] arr5 = Arrays.copyOfRange(arr, 2, 4);  // arr4=[2,3] ← 4는 포함되지 않음
int[] arr6 = Arrays.copyOfRange(arr, 0, 7);  // arr4=[0,1,2,3,4,0,0]
```

4. 배열 채우기 - fill(), setAll()

```
int[] arr = new int[5];
Arrays.fill(arr, 9);    // arr=[9,9,9,9,9]
Arrays.setAll(arr, () -> (int)(Math.random()*5)+1); // arr=[1,5,2,1,1]
```

6.1 Arrays(3/3) - 배열을 다루기 편리한 메서드(static) 제공

5. 배열을 List로 변환 - asList(Object... a)

```
List list = Arrays.asList(new Integer[]{1,2,3,4,5}); // list =[1, 2, 3, 4, 5]
List list = Arrays.asList(1,2,3,4,5);               // list =[1, 2, 3, 4, 5]
list.add(6);  // UnsupportedOperationException 예외 발생.  list의 크기를 변경할 수 없음.
```

```
List list = new ArrayList(Arrays.asList(1,2,3,4,5)); // 변경가능한 ArrayList생성
```

6. 배열의 정렬과 검색 - sort(), binarySearch()

```
int[] arr = { 3, 2, 0, 1, 4};                // 정렬되지 않은 배열
int idx = Arrays.binarySearch(arr, 2);       // idx=-5 ← 잘못된 결과

Arrays.sort(arr);  // 배열 arr을 정렬한다.
System.out.println(Arrays.toString(arr)); // [0, 1, 2, 3, 4]
int idx = Arrays.binarySearch(arr, 2);       // idx=2  ← 올바른 결과
```

▌알아두면 좋아요! – 순차 검색과 이진 검색

6.2 Comparator와 Comparable

▶ 객체를 정렬하는데 필요한 메서드를 정의한 인터페이스(정렬기준을 제공)

Comparable	기본 정렬기준을 구현하는데 사용.
Comparator	기본 정렬기준 외에 다른 기준으로 정렬하고자할 때 사용

```
public final class Integer extends Number implements Comparable {
    ...
    public in t compareTo(Integer anotherInteger) {
        int v1 = this.value;
        int v2 = anotherInteger.value;
        // 같으면 0, 오른쪽 값이 크면 -1, 작으면 1을 반환
        return (v1 < v2 ? -1 : (v1==v2? 0 : 1));
    }
    ...
}
```

▶ compare()와 compareTo()는 두 객체의 비교결과를 반환하도록 작성
같으면 0, 오른쪽이 크면 음수(-), 작으면 양수(+)

```
public interface Comparator {
    int compare(Object o1, Object o2); // o1, o2 두 객체를 비교
    boolean equals(Object obj); // equals를 오버라이딩하라는 뜻
}
public interface Comparable {
    int compareTo(Object o); // 주어진 객체(o)를 자신과 비교
}
```

6.2 Comparator와 Comparable – example

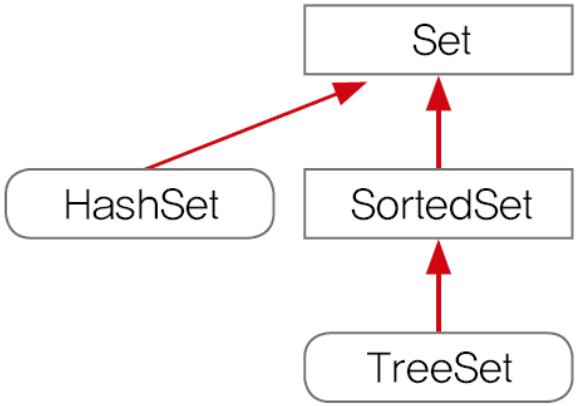
```
public static void main(String[] args) {
    Integer[] arr = { 30, 50, 10, 40, 20 };

    Arrays.sort(arr); // 기본 정렬기준(Comparable)으로 정렬
    System.out.println(Arrays.toString(arr));

    // sort(Object[] objArr, Comparator c)
    Arrays.sort(arr, new DescComp()); // DescComp에 구현된 정렬기준으로 정렬
    System.out.println(Arrays.toString(arr));
}
```

```
------- java -------
[10, 20, 30, 40, 50]
[50, 40, 30, 20, 10]
```

```
public final class Integer extends Number implements Comparable {
    ...
    public in t compareTo(Integer anotherInteger) {
        int v1 = this.value;
        int v2 = anotherInteger.value;

        // 같으면 0, 오른쪽 값이 크면 -1, 작으면 1을 반환
        return (v1 < v2 ? -1 : (v1==v2? 0 : 1));
    }
    ...
}
```

```
class DescComp implements Comparator {
    public int compare(Object o1, Object o2) {
        if(!(o1 instanceof Integer)) return -1;
        if(!(o2 instanceof Integer)) return -1;

        Integer i  = (Integer)o1;
        Integer i2 = (Integer)o2;
        return i.compareTo(i2) * -1; // 기본 정렬방식의 반대
    }
}
```

7.1 HashSet과 TreeSet - 순서X, 중복X



▶ HashSet

- Set인터페이스를 구현한 대표적인 컬렉션 클래스
- 순서를 유지하려면, LinkedHashSet클래스를 사용하면 된다.

▶ TreeSet

- 범위 검색과 정렬에 유리한 컬렉션 클래스
- HashSet보다 데이터 추가, 삭제에 시간이 더 걸림

7.2 HashSet – boolean add(Object o)

- HashSet은 객체를 저장하기전에 기존에 같은 객체가 있는지 확인한다.
  같은 객체가 없으면 저장하고, 있으면 저장하지 않는다.
- boolean add(Object o)는 저장할 객체의 equals()와 hashCode()를 호출
  equals()와 hashCode()가 오버라이딩 되어 있어야 함

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return name +":"+ age;
    }
}
```

```
public boolean equals(Object obj) {
    if(!(obj instanceof Person)) return false;

    Person tmp = (Person)obj;

    return name.equals(tmp.name) && age==tmp.age;
}

public int hashCode() {
    return (name+age).hashCode();
}
```

7.3 HashSet – hashCode()의 오버라이딩 조건

▶ 동일 객체에 대해 hashCode()를 여러 번 호출해도 동일한 값을 반환해야 한다.

```
Person2 p = new Person2("David", 10);

int hashCode1 = p.hashCode();
int hashCode2 = p.hashCode();

p.age = 20;
int hashCode3 = p.hashCode();
```

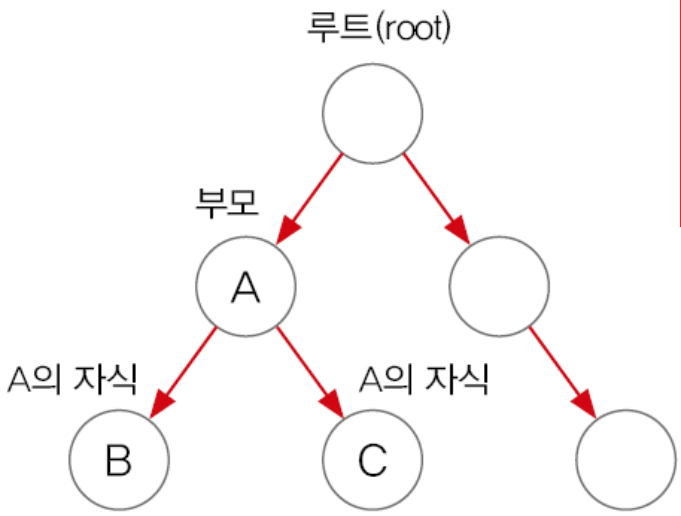
▶ equals()로 비교해서 true를 얻은 두 객체의 hashCode()값은 일치해야 한다.

```
Person2 p1 = new Person2("David", 10);
Person2 p2 = new Person2("David", 10);

boolean b = p1.equals(p2);

int hashCode1 = p1.hashCode();
int hashCode2 = p2.hashCode();
```

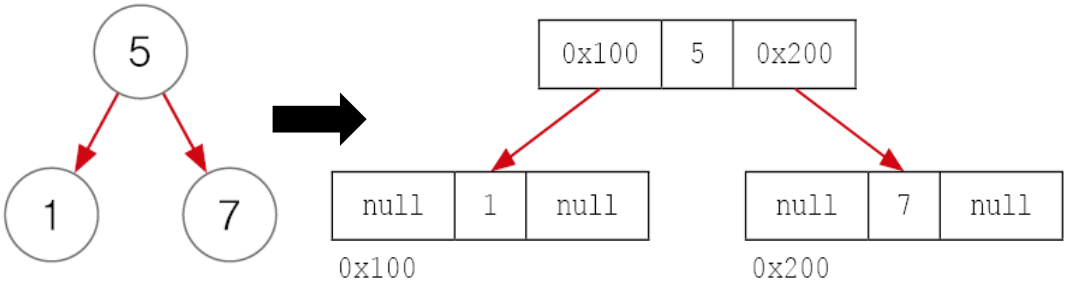
※ equals()로 비교한 결과가 false인 두 객체의 hashCode()값이 같을 수도 있다.
그러나 성능 향상을 위해 가능하면 서로 다른 값을 반환하도록 작성하자.

7.4 TreeSet – 범위 검색과 정렬에 유리

- 범위 검색과 정렬에 유리한 이진 검색 트리(binary search tree)로 구현
  링크드 리스트처럼 각 요소(node)가 나무(tree)형태로 연결된 구조
- 이진 트리는 모든 노드가 최대 두 개의 하위 노드를 갖음(부모-자식관계)
- 이진 검색 트리는 부모보다 작은 값을 왼쪽에, 큰 값은 오른쪽에 저장
- HashSet보다 데이터 추가, 삭제에 시간이 더 걸림(반복적인 비교 후 저장)




```
class TreeNode {
    TreeNode left;     // 왼쪽 자식노드
    Object   element;  // 저장할 객체
    TreeNode right;    // 오른쪽 자식노드
}
```



7.5 TreeSet – 데이터 저장과정 boolean add(Object o)

※ TreeSet에 7,4,9,1,5의 순서로 데이터를 저장하면, 아래의 과정을 거친다.

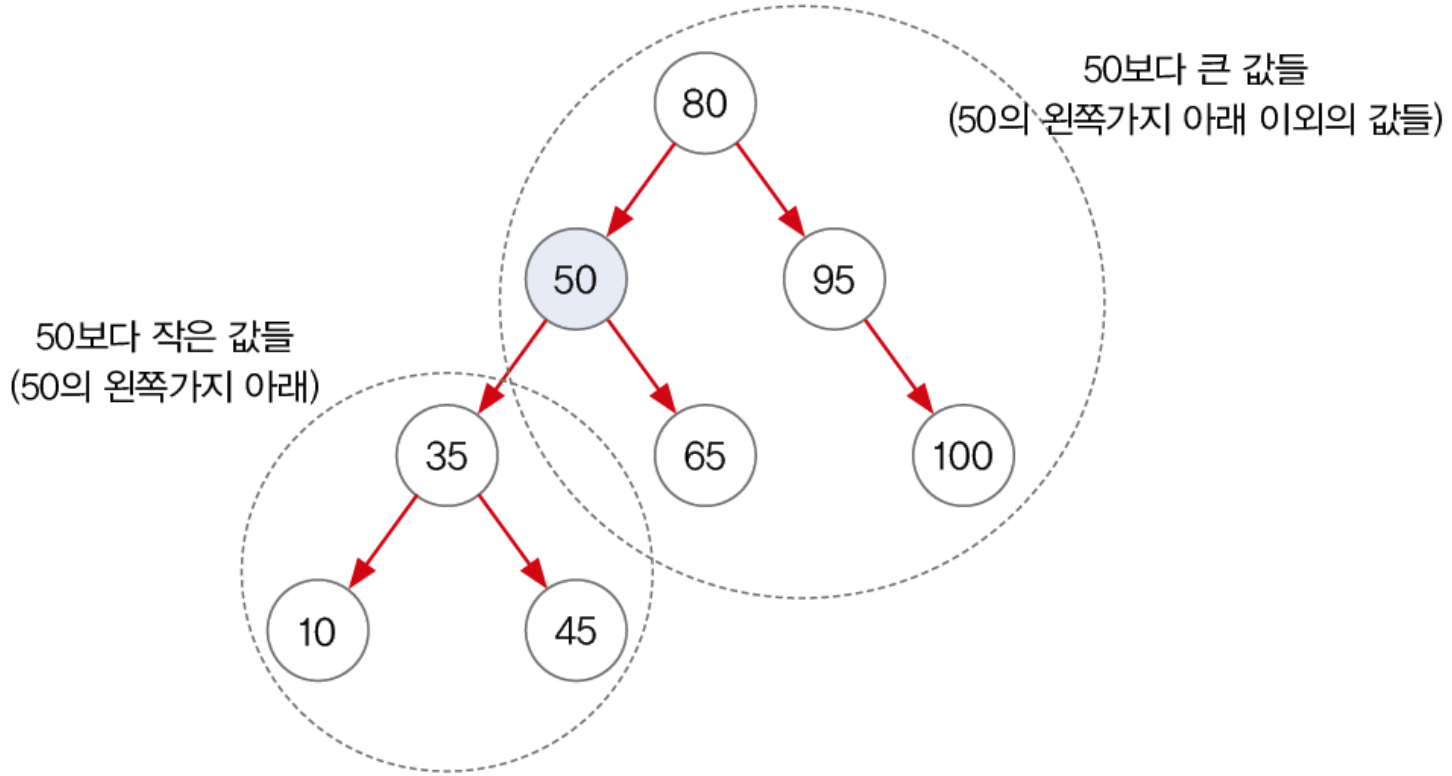
(루트부터 트리를 따라 내려가며 값을 비교. 작으면 왼쪽, 크면 오른쪽에 저장)

7.6 TreeSet – 주요 생성자와 메서드

생성자 또는 메서드	설 명
TreeSet( )	기본 생성자
TreeSet(Collection c)	주어진 컬렉션을 저장하는 TreeSet을 생성
TreeSet(Comparator comp)	주어진 정렬기준으로 정렬하는 TreeSet을 생성
Object first( )	정렬된 순서에서 첫 번째 객체를 반환한다.
Object last( )	정렬된 순서에서 마지막 객체를 반환한다.
Object ceiling(Object o)	지정된 객체와 같은 객체를 반환. 없으면 큰 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
Object floor(Object o)	지정된 객체와 같은 객체를 반환. 없으면 작은 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
Object higher(Object o)	지정된 객체보다 큰 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
Object lower(Object o)	지정된 객체보다 작은 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
SortedSet subSet(Object fromElement, Object toElement)	범위 검색(fromElement와 toElement사이)의 결과를 반환한다.(끝 범위인 toElement는 범위에 포함되지 않음)
SortedSet headSet(Object toElement)	지정된 객체보다 작은 값의 객체들을 반환한다.
SortedSet tailSet(Object fromElement)	지정된 객체보다 큰 값의 객체들을 반환한다.

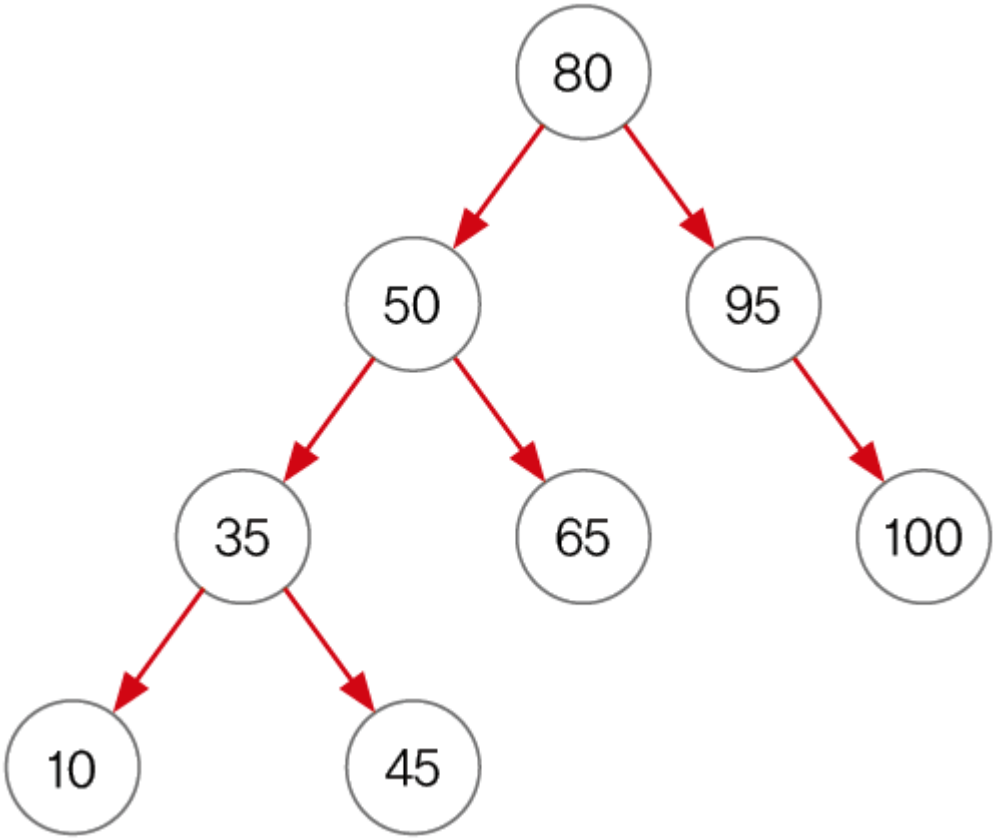
7.7 TreeSet – 범위 검색 subSet(), headSet(), tailSet()

메서드	설 명
SortedSet subSet(Object fromElement, Object toElement)	범위 검색(fromElement와 toElement사이)의 결과를 반환한다.(끝 범위인 toElement는 범위에 포함되지 않음)
SortedSet headSet(Object toElement)	지정된 객체보다 작은 값의 객체들을 반환한다.
SortedSet tailSet(Object fromElement)	지정된 객체보다 큰 값의 객체들을 반환한다.



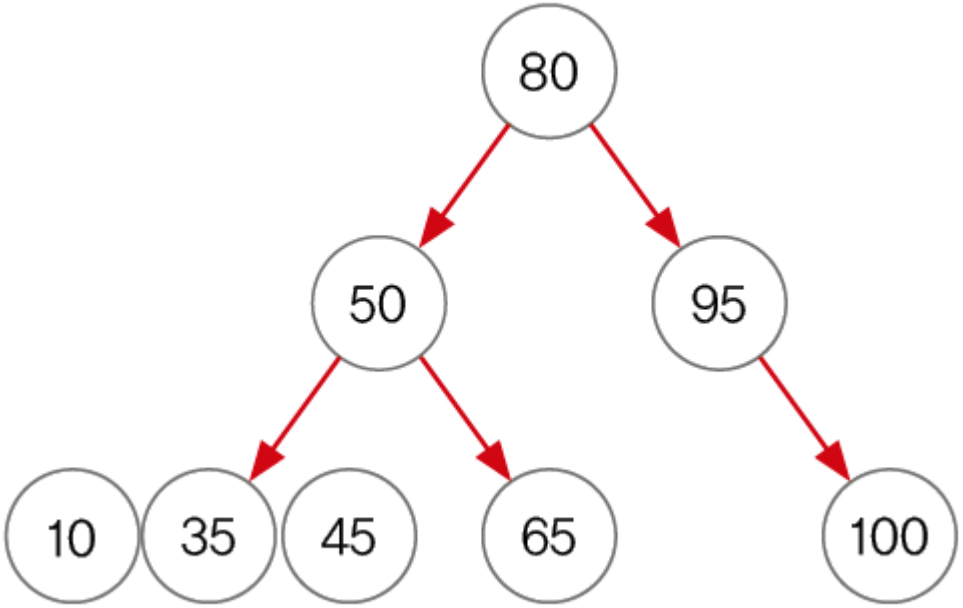
7.8 TreeSet – 트리 순회(전위, 중위, 후위)

- 이진 트리의 모든 노드를 한번씩 읽는 것을 트리 순회라고 한다.
- 전위, 중위 후위 순회법이 있으며, 중위 순회하면 오름차순으로 정렬된다.



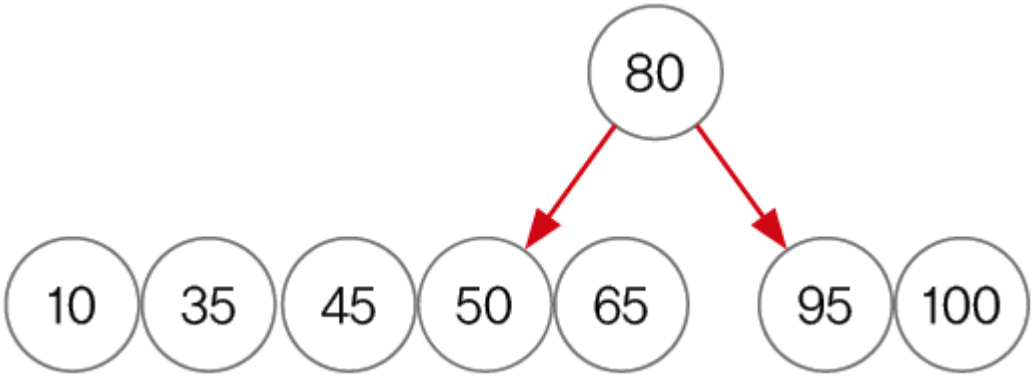
7.8 TreeSet – 트리 순회(전위, 중위, 후위)

- 이진 트리의 모든 노드를 한번씩 읽는 것을 트리 순회라고 한다.
- 전위, 중위 후위 순회법이 있으며, 중위 순회하면 오름차순으로 정렬된다.



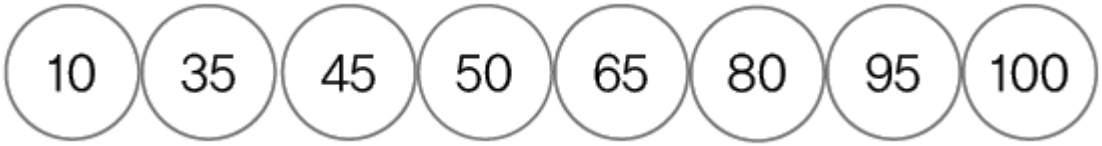
7.8 TreeSet – 트리 순회(전위, 중위, 후위)

- 이진 트리의 모든 노드를 한번씩 읽는 것을 트리 순회라고 한다.
- 전위, 중위 후위 순회법이 있으며, 중위 순회하면 오름차순으로 정렬된다.



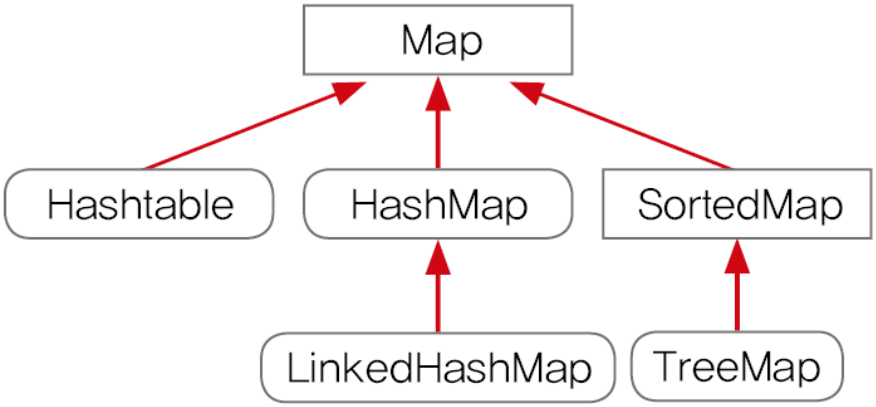
7.8 TreeSet – 트리 순회(전위, 중위, 후위)

- 이진 트리의 모든 노드를 한번씩 읽는 것을 트리 순회라고 한다.
- 전위, 중위 후위 순회법이 있으며, 중위 순회하면 오름차순으로 정렬된다.



8.1 HashMap과 TreeMap - 순서X, 중복(키X,값O)

- Map인터페이스를 구현. 데이터를 키와 값의 쌍으로 저장
- HashMap(동기화X)은 Hashtable(동기화O)의 신버젼



▶ HashMap

- Map인터페이스를 구현한 대표적인 컬렉션 클래스
- 순서를 유지하려면, LinkedHashMap클래스를 사용하면 된다.

▶ TreeMap

- 범위 검색과 정렬에 유리한 컬렉션 클래스
- HashMap보다 데이터 추가, 삭제에 시간이 더 걸림

8.2 HashMap

- 해싱(hashing)기법으로 데이터를 저장. 데이터가 많아도 검색이 빠르다.
- Map인터페이스를 구현. 데이터를 키와 값의 쌍으로 저장

키(key)	컬렉션 내의 키(key) 중에서 유일해야 한다.
값(value)	키(key)와 달리 데이터의 중복을 허용한다.

```
HashMap map = new HashMap();
map.put("myId", "1234");
map.put("asdf", "1111");
map.put("asdf", "1234");
```

키(key)	값(value)
myId	1234
asdf	1234

```
public class HashMap extends AbstractMap
                     implements Map, Cloneable,Serializable {
    transient Entry[] table;
    ...
    static class Entry implements Map.Entry {
        final Object key;
        Object value;
        ...
    }
}
```

비객체지향적인 코드	객체지향적인 코드

Object[ ] key;
Object[ ] value;
	Entry[ ] table;
	class Entry {
	  Object key;
	  Object value;
	}

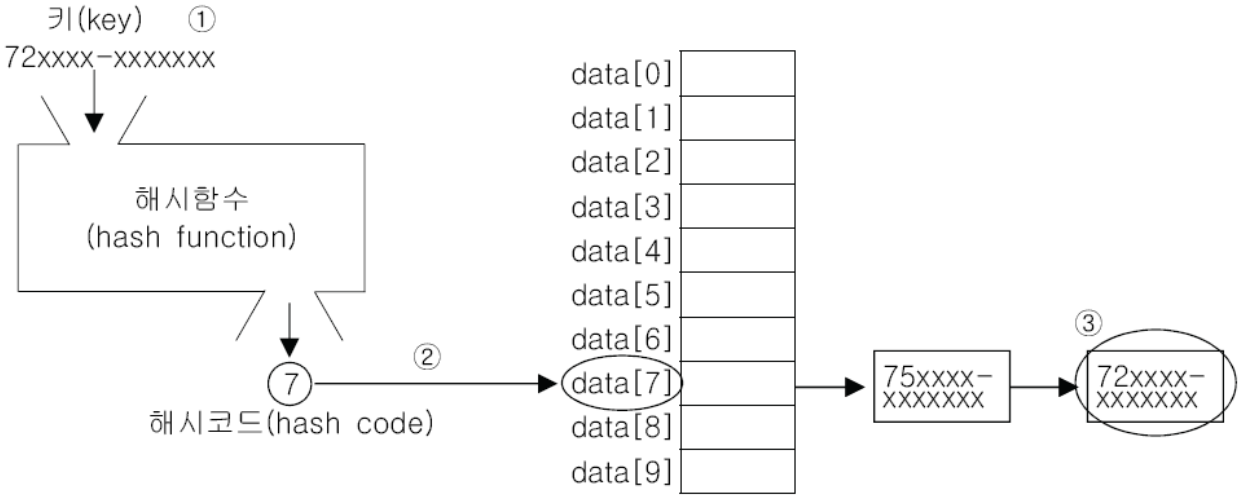
8.2 HashMap - 주요 메서드

키(key)	값(value)
myId	1234
asdf	1234

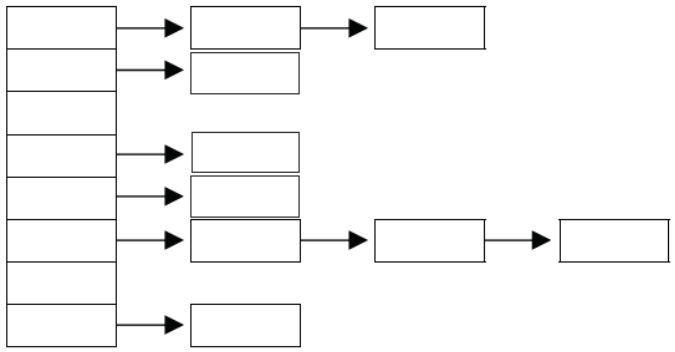
생성자 / 메서드	설명
HashMap( )	HashMap객체를 생성
HashMap(int initialCapacity)	지정된 값을 초기용량으로 하는 HashMap객체를 생성
HashMap(int initialCapacity, float loadFactor)	지정된 초기용량과 load factor의 HashMap객체를 생성
HashMap(Map m)	지정된 Map의 모든 요소를 포함하는 HashMap을 생성
Object put(Object key, Object value)	지정된 키와 값을 HashMap에 저장
void putAll(Map m)	Map에 저장된 모든 요소를 HashMap에 저장
Object remove(Object key)	HashMap에서 지정된 키로 저장된 값(객체)를 제거
Object replace(Object key, Object value)	지정된 키의 값을 지정된 객체(value)로 대체
boolean replace(Object key, Object oldVal, Object newVal)	지정된 키와 객체(oldVal)가 모두 일치하는 경우에만 새로운 객체(newVal)로 대체
boolean containsKey(Object key)	HashMap에 지정된 키(key)가 포함되어있는지 알려준다.(포함되어 있으면 true)
boolean containsValue(Object value)	HashMap에 지정된 값(value)가 포함되어있는지 알려준다.(포함되어 있으면 true)
Object get(Object key)	지정된 키(key)의 값(객체)을 반환. 못찾으면 null 반환
Object getOrDefalt(Object key, Object defaultValue)	지정된 키(key)의 값(객체)을 반환한다. 키를 못찾으면, 기본값(defaultValue)로 지정된 객체를 반환
Set entrySet( )	HashMap에 저장된 키와 값을 엔트리(키와 값의 결합)의 형태로 Set에 저장해서 반환
Set keySet( )	HashMap에 저장된 모든 키가 저장된 Set을 반환
Collection values( )	HashMap에 저장된 모든 값을 컬렉션의 형태로 반환
void clear( )	HashMap에 저장된 모든 객체를 제거
boolean isEmpty( )	HashMap이 비어있는지 알려준다.
int size( )	HashMap에 저장된 요소의 개수를 반환

8.3 해싱(hashing) - (1/3)

- 해시함수(hash function)로 해시테이블(hash table)에 데이터를 저장, 검색

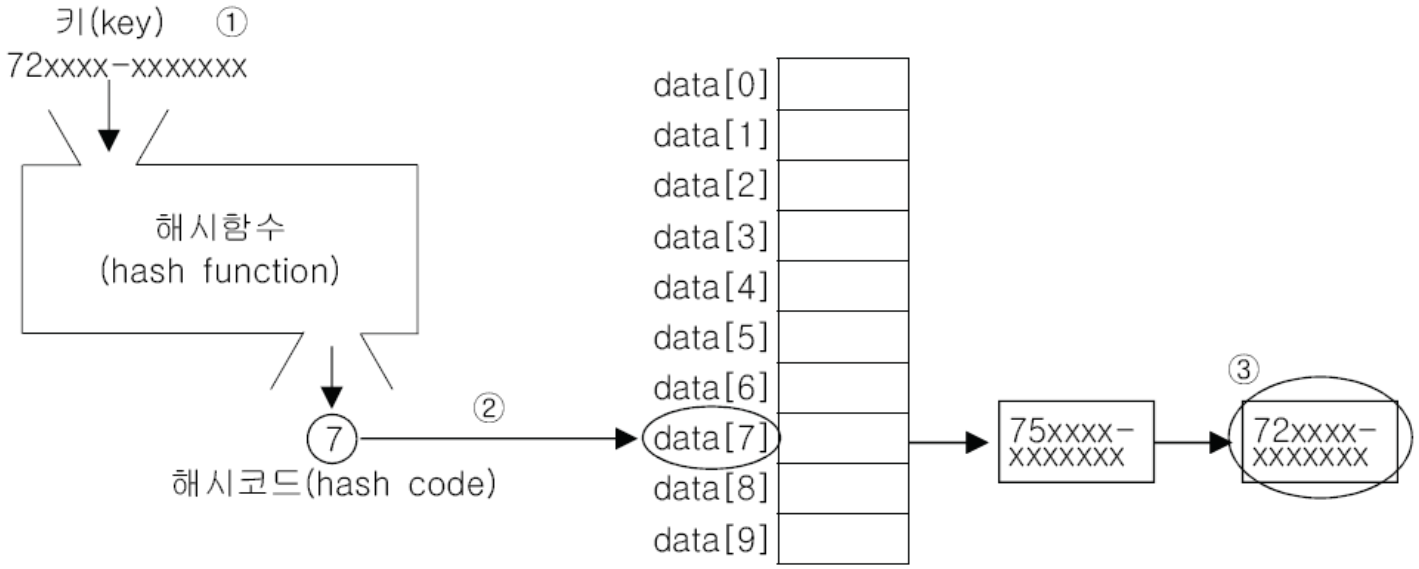


- 해시테이블은 배열과 링크드 리스트가 조합된 형태



8.3 해싱(hashing) - (3/3)

▶ 해시테이블에 저장된 데이터를 가져오는 과정



① 키로 해시함수를 호출해서 해시코드를 얻는다.

② 해시코드(해시함수의 반환값)에 대응하는 링크드리스트를 배열에서 찾는다.

③ 링크드리스트에서 키와 일치하는 데이터를 찾는다.

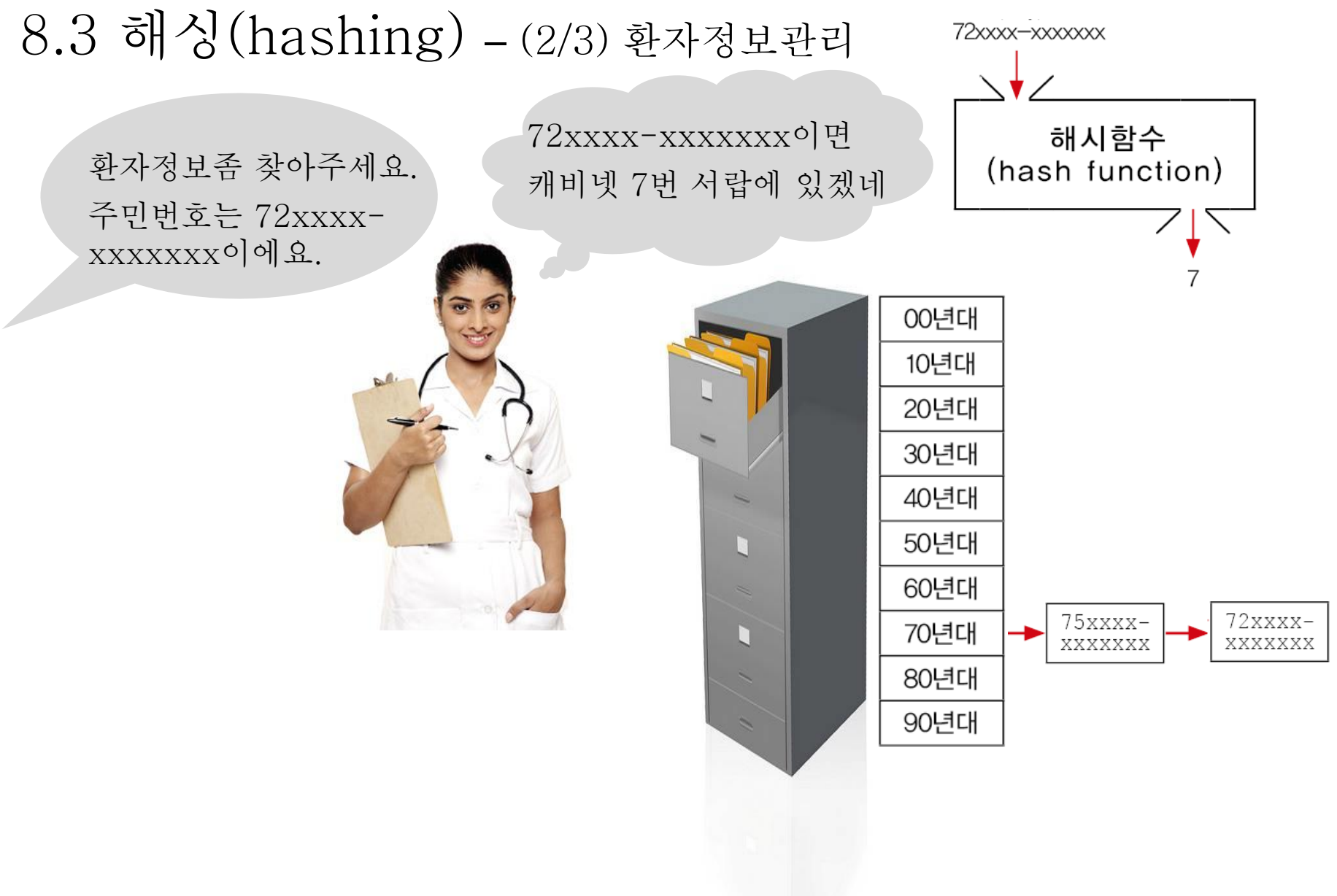
※ 해시함수는 같은 키에 대해 항상 같은 해시코드를 반환해야 한다.

서로 다른 키일지라도 같은 값의 해시코드를 반환할 수도 있다.

8.3 해싱(hashing) – (2/3) 환자정보관리

8.3 해싱(hashing) – (2/3) 환자정보관리

8.3 해싱(hashing) – (2/3) 환자정보관리



8.4 TreeMap

- 이진 검색 트리의 구조로 키와 값의 쌍으로 이루어진 데이터를 저장
- TreeSet처럼, 데이터를 정렬(키)해서 저장하기 때문에 저장시간이 길다. (TreeSet은 TreeMap을 이용해서 구현되어 있음)
- 다수의 데이터에서 개별적인 검색은 TreeMap보다 HashMap이 빠르다.
- Map이 필요할 때 주로 HashMap을 사용하고, 정렬이나 범위검색이 필요한 경우에 TreeMap을 사용

9.1 Properties

- 내부적으로 Hashtable을 사용하며, key와 value를 (String, String)로 저장
- 주로 어플리케이션의 환경설정에 관련된 속성을 저장하는데 사용되며 파일로부터 편리하게 값을 읽고 쓸 수 있는 메서드를 제공한다.

메서드	설명
Properties()	Properties()객체를 생성한다.
Properties(Properties defaults)	지정된 Properties에 저장된 목록을 가진Properties()객체를 생성한다.
String getProperty(String key)	지정된 키(key)의 값(value)을 반환한다.
String getProperty(String key, String defaultValue)	지정된 키(key)의 값(value)을 반환한다. 키를 못찾으면 defaultValue를 반환한다.
void list(PrintStream out)	지정된 PrintStream에 저장된 목록을 출력한다.
void list(PrintWriter out)	지정된 PrintWriter에 저장된 목록을 출력한다.
void load(InputStream inStream)	지정된 InputStream으로부터 목록을 읽어서 저장한다.
void loadFromXML(InputStream in) *	지정된 InputStream으로부터 XML문서를 읽어서, XML문서에 저장된 목록을 읽어다 담는다.(load & store)
Enumeration propertyNames()	목록의 모든 키(key)가 담긴 Enumeration을 반환한다.
void save(OutputStream out, String header)	deprecated되었으므로 store()를 사용하자.
Object setProperty(String key, String value)	지정된 키와 값을 저장한다. 이미 존재하는 키(key)면 새로운 값(value)로 바뀐다.
void store(OutputStream out, String header)	저장된 목록을 지정된 출력스트림에 출력(저장)한다. header는 목록에 대한 설명(주석)으로 저장된다.
void storeToXML(OutputStream os, String comment)*	저장된 목록을 지정된 출력스트림에 XML문서로 출력(저장)한다. comment는 목록에 대한 설명(주석)으로 저장된다.
void storeToXML(OutputStream os, String comment, String encoding) *	저장된 목록을 지정된 출력스트림에 해당 인코딩의 XML문서로 출력(저장)한다. comment는 목록에 대한 설명(주석)으로 저장된다.

9.2 Properties – 예제(example)

```
import java.util.*;
import java.io.*;

class PropertiesEx3
{
    public static void main(String[] args)
    {
        Properties prop = new Properties();

        prop.setProperty("timeout","30");
        prop.setProperty("language","한글");
        prop.setProperty("size","10");
        prop.setProperty("capacity","10");

        try {
            prop.store(new FileOutputStream("output.txt"), "Properties Example");
            prop.storeToXML(new FileOutputStream("output.xml"), "Properties Example");
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

[output.txt]

```
#Properties Example
#Sat Aug 29 10:58:41 KST 2009
capacity=10
size=10
timeout=30
language=\uD55C\uAE00
```

[output.xml]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Properties Example</comment>
<entry key="capacity">10</entry>
<entry key="size">10</entry>
<entry key="timeout">30</entry>
<entry key="language">한글</entry>
</properties>
```

9.3 Collections(1/2) - 컬렉션을 위한 메서드(static)를 제공

1. 컬렉션 채우기, 복사, 정렬, 검색 - fill(), copy(), sort(), binarySearch() 등
2. 컬렉션의 동기화 - synchronizedXXX()

```
static Collection synchronizedCollection(Collection c)
static List       synchronizedList(List list)
static Set        synchronizedSet(Set s)
static Map        synchronizedMap(Map m)
static SortedSet  synchronizedSortedSet(SortedSet s)
static SortedMap  synchronizedSortedMap(SortedMap m)
```

```
List syncList = Collections.synchronizedList(new ArrayList(...));
```

3. 변경불가(readOnly) 컬렉션 만들기 - unmodifiableXXX()

```
static  Collection   unmodifiableCollection(Collection c)
static  List         unmodifiableList(List list)
static  Set          unmodifiableSet(Set s)
static  Map          unmodifiableMap(Map m)
static  NavigableSet unmodifiableNavigableSet(NavigableSet s)
static  SortedSet    unmodifiableSortedSet(SortedSet s)
static  NavigableMap unmodifiableNavigableMap(NavigableMap m)
static  SortedMap    unmodifiableSortedMap(SortedMap m)
```

9.3 Collections(2/2) - 컬렉션을 위한 메서드(static)를 제공

4. 싱글톤 컬렉션 만들기 - singletonXXX()

```
static  List singletonList(Object o)
static  Set  singleton(Object o)      // singletonSet이 아님
static  Map  singletonMap(Object key, Object value)
```

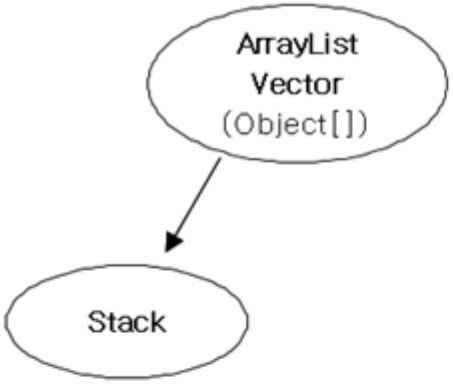
5. 한 종류의 객체만 저장하는 컬렉션 만들기 - checkedXXX()

```
static  Collection    checkedCollection(Collection c, Class type)
static  List          checkedList(List list, Class type)
static  Set           checkedSet(Set s, Class type)
static  Map           checkedMap(Map m, Class keyType, Class valueType)
static  Queue         checkedQueue(Queue queue, Class type)
static  NavigableSet  checkedNavigableSet(NavigableSet s, Class type)
static  SortedSet     checkedSortedSet(SortedSet s, Class type)
static  NavigableMap  checkedNavigableMap(NavigableMap m, Class keyType, Class valueType)
static  SortedMap     checkedSortedMap(SortedMap m, Class keyType, Class valueType)
```

```
List list = new ArrayList();
List checkedList = checkedList(list, String.class); // String만 저장가능
checkedList.add("abc");                 // OK.
checkedList.add(new Integer(3));        // 에러. ClassCastException발생
```

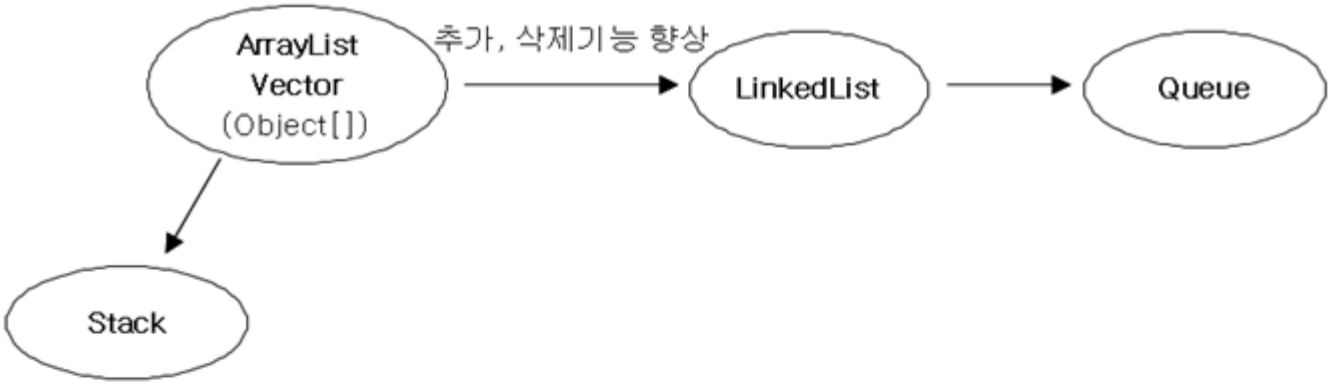
9.4 컬렉션 클래스 정리 & 요약 (1/2)

9.4 컬렉션 클래스 정리 & 요약 (1/2)

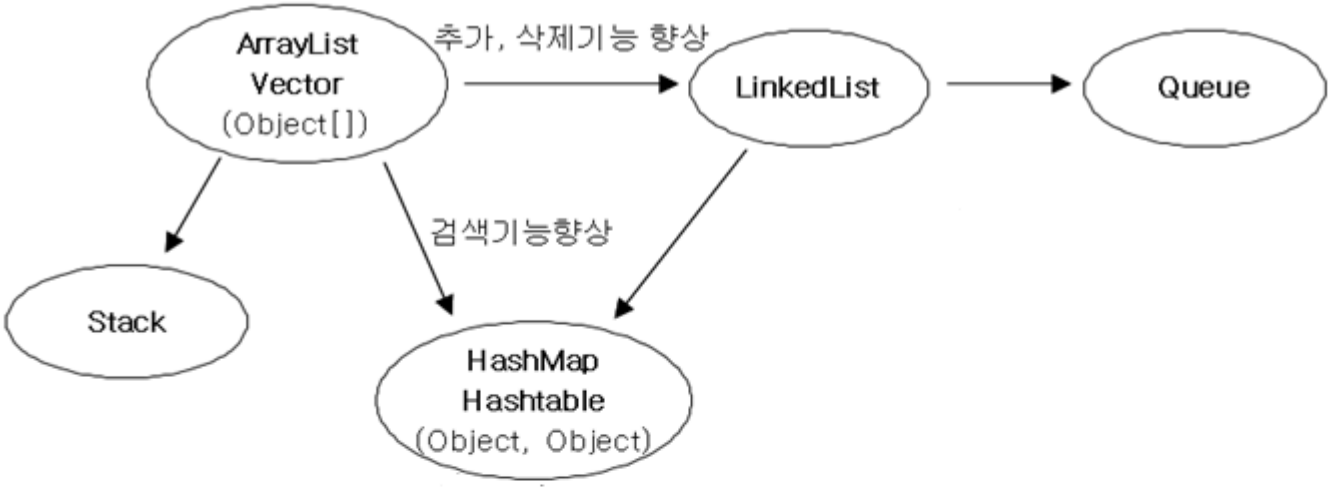


9.4 컬렉션 클래스 정리 & 요약 (1/2)

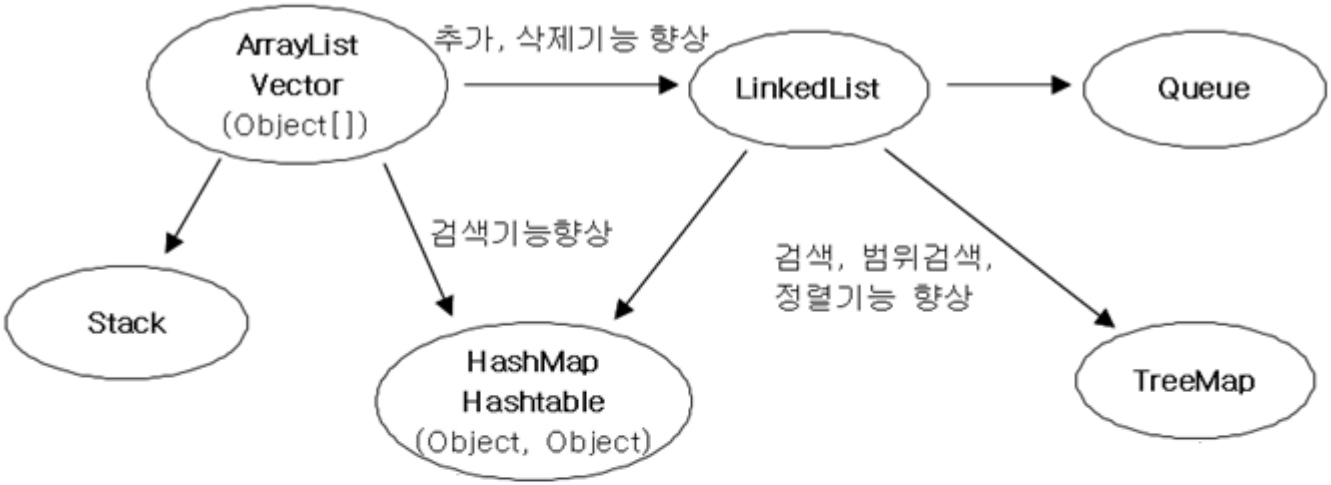
9.4 컬렉션 클래스 정리 & 요약 (1/2)



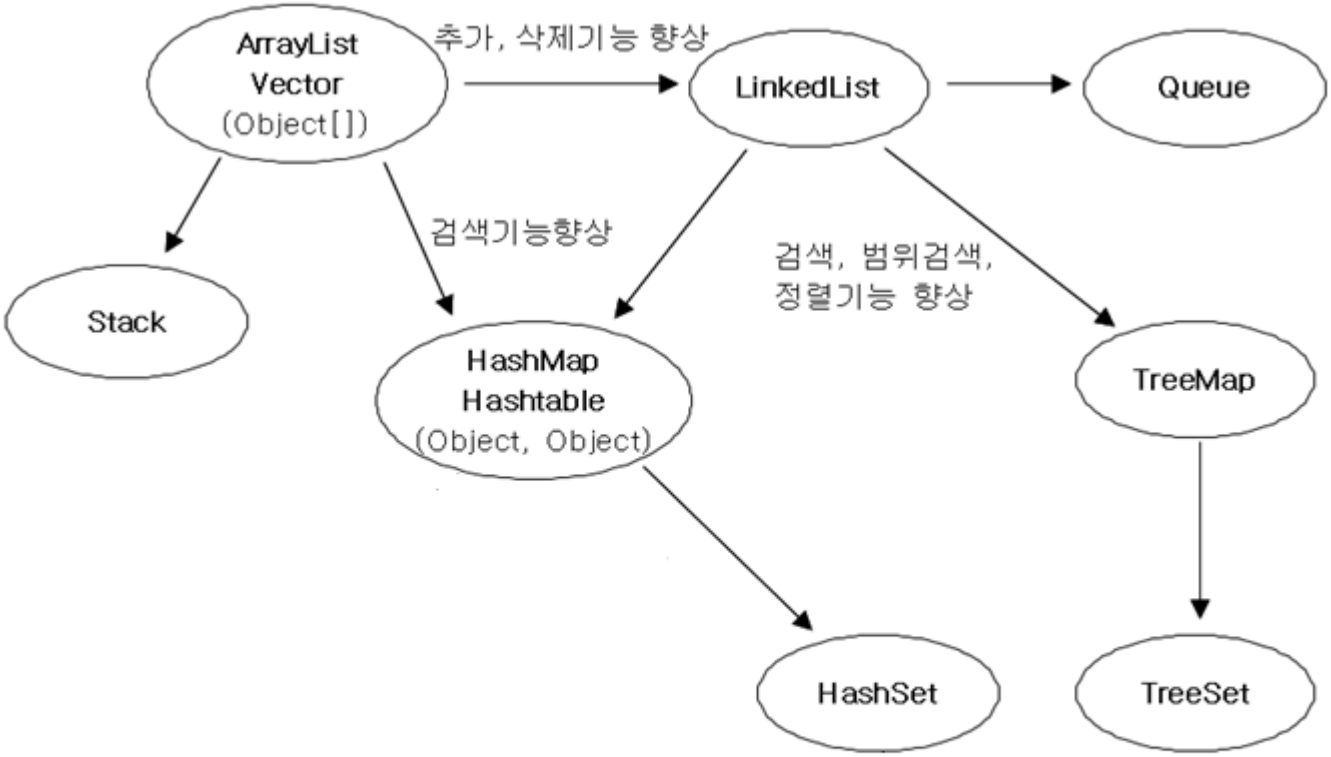
9.4 컬렉션 클래스 정리 & 요약 (1/2)



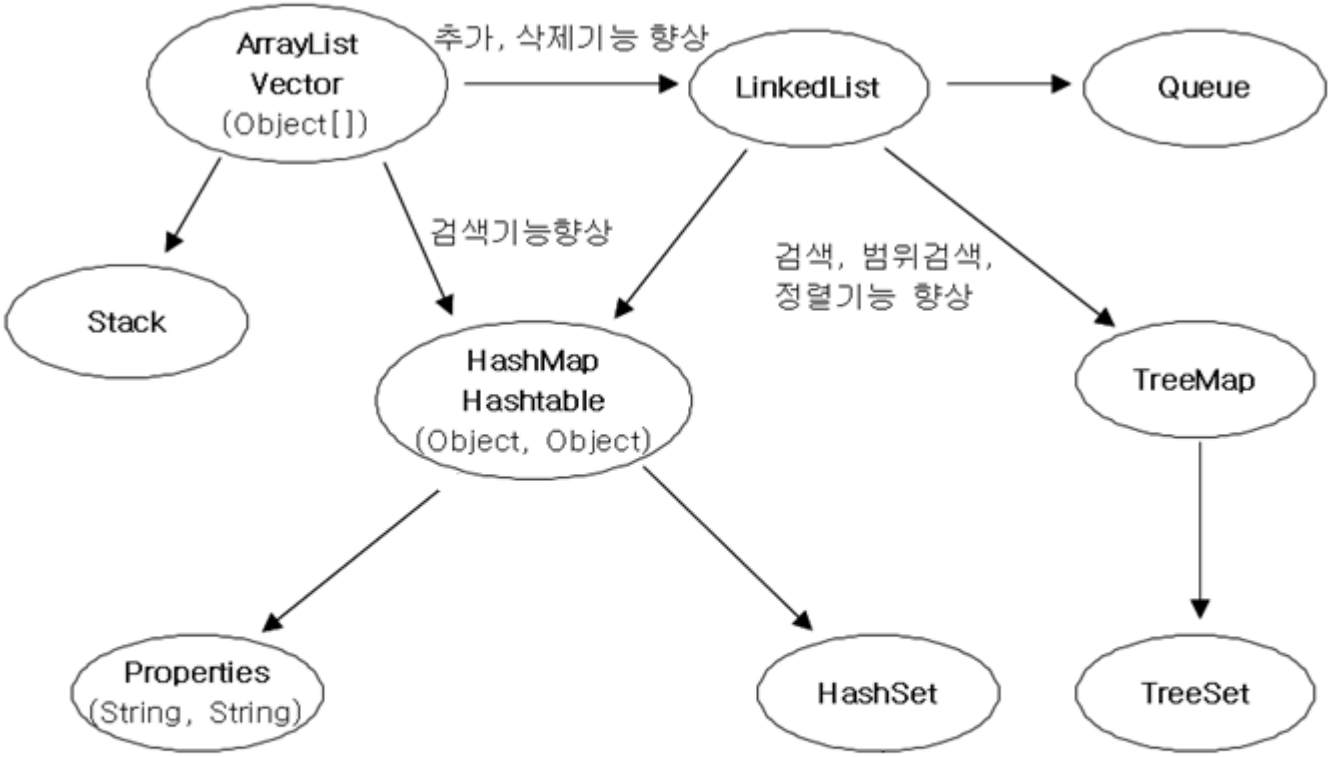
9.4 컬렉션 클래스 정리 & 요약 (1/2)



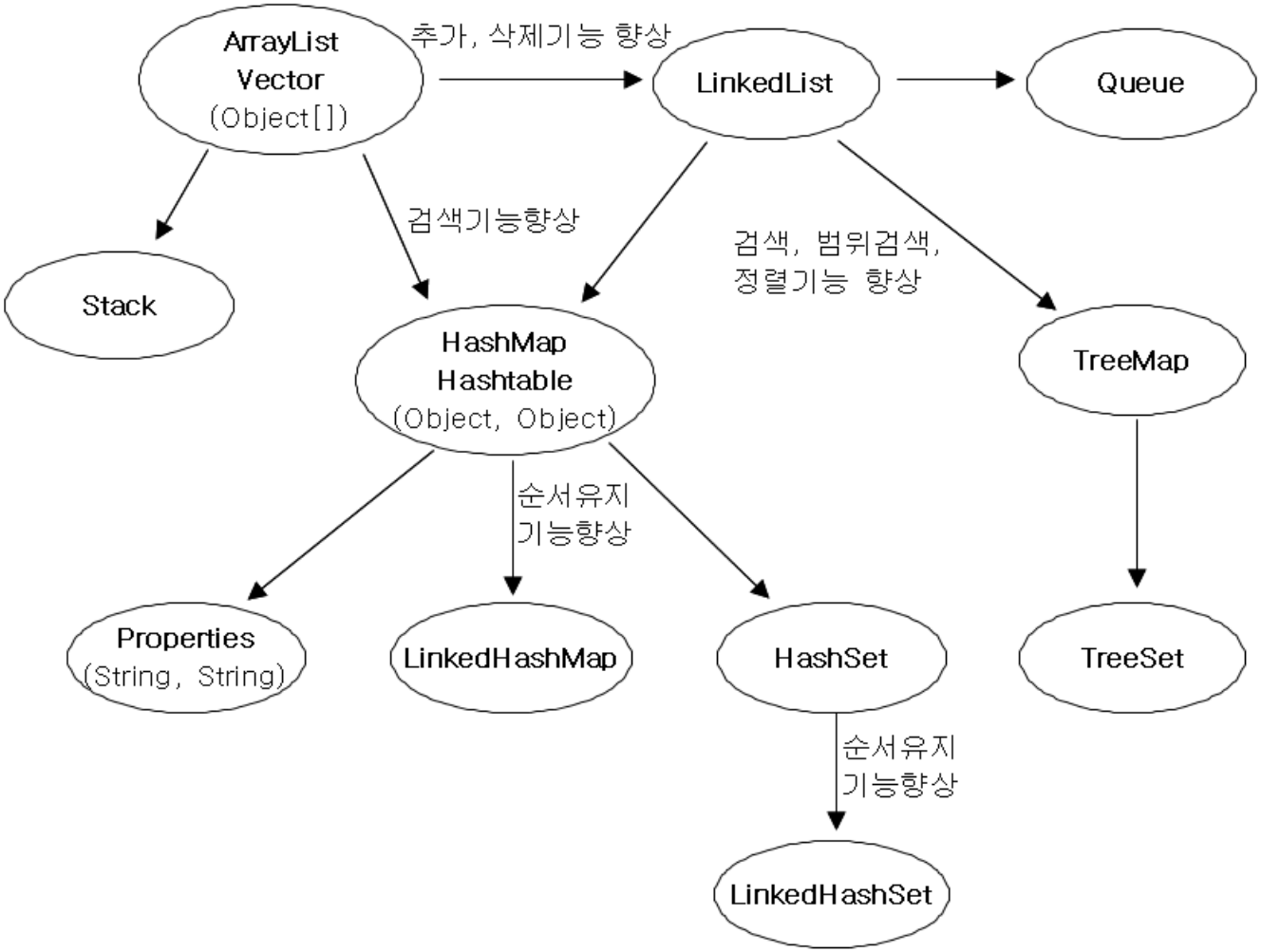
9.4 컬렉션 클래스 정리 & 요약 (1/2)



9.4 컬렉션 클래스 정리 & 요약 (1/2)



9.4 컬렉션 클래스 정리 & 요약 (1/2)



9.4 컬렉션 클래스 정리 & 요약 (2/2)

컬렉션	특 징
ArrayList	배열기반, 데이터의 추가와 삭제에 불리, 순차적인 추가/삭제는 제일 빠름. 임의의 요소에 대한 접근성(accessibility)이 뛰어남.
LinkedList	연결기반. 데이터의 추가와 삭제에 유리. 임의의 요소에 대한 접근성이 좋지 않다.
HashMap	배열과 연결이 결합된 형태. 추가, 삭제, 검색, 접근성이 모두 뛰어남. 검색에는 최고성능을 보인다.
TreeMap	연결기반. 정렬과 검색(특히 범위검색)에 적합. 검색성능은 HashMap보다 떨어짐.
Stack	Vector를 상속받아 구현(LIFO)
Queue	LinkedList가 Queue인터페이스를 구현(FIFO)
Properties	Hashtable을 상속받아 구현(String, String)
HashSet	HashMap을 이용해서 구현
TreeSet	TreeMap을 이용해서 구현
LinkedHashMap LinkedHashSet	HashMap과 HashSet에 저장순서유지기능을 추가하였음.