Java의 정석

제 11 장

컬렉션 프레임웍 (collection framework)

2009. 8. 29

남궁성 강의

castello@naver.com

1.1 컬렉션 프레임웍(collection framework)이란?

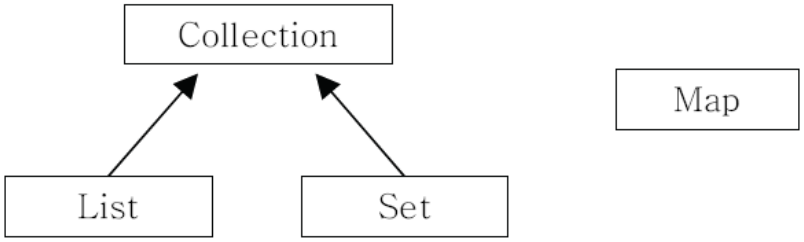
▶ 컬렉션 프레임웍(collection framework)
- 데이터 군(群)을 저장하는 클래스들을 표준화한 설계
- 다수의 데이터를 쉽게 처리할 수 있는 방법을 제공하는 클래스들로 구성
- JDK1.2부터 제공

▶ 컬렉션(collection)
- 다수의 데이터, 즉, 데이터 그룹을 의미한다.

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

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

1.2 컬렉션 프레임웍의 핵심 인터페이스



[그림11-1] 컬렉션 프레임웍의 핵심 인터페이스간의 상속계층도

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

[표11-1] 컬렉션 프레임웍의 핵심 인터페이스와 특징

1.3 컬렉션 프레임웍의 동기화(synchronization)

- 멀티쓰레드 프로그래밍에서는 컬렉션 클래스에 동기화 처리가 필요하다.
- Vector와 같은 구버젼 클래스들은 자체적으로 동기화처리가 되어 있다.
- ArrayList와 같은 신버젼 클래스들은 별도의 동기화처리가 필요하다.
- Collections클래스는 다음과 같은 동기화 처리 메서드를 제공한다.

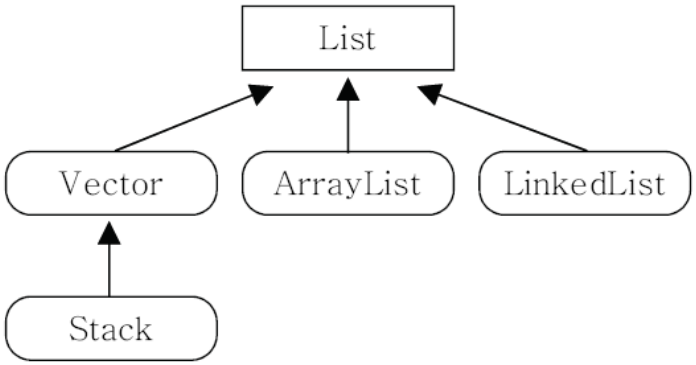
[주의] java.util.Collection은 인터페이스이고, java.util.Collections는 클래스이다.

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

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

1.4 Vector와 ArrayList

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




```
public class Vector extends AbstractList
       implements List, RandomAccess, Cloneable, java.io.Serializable
{
       ...
       protected Object elementData[];
       ...
}
```

1.4 Vector – 주요메서드(1/2)

메서드	설 명
Vector()	크기가 10인 Vector를 생성한다.
Vector(Collection c)	주어진 컬렉션을 저장할 수 있는 생성자
Vector(int initialCapacity)	Vector의 초기용량을 지정할 수 있는 생성자
Vector(int initialCapacity, int capacityIncrement)	초기용량과 용량의 증분을 지정할 수 있는 생성자
void add(int index, Object element)	지정된 위치(index)에 객체(element)를 저장한다.
boolean add(Object o)	객체(o)를 저장한다.
boolean addAll(Collection c)	주어진 컬렉션의 모든 객체를 저장한다.
boolean addAll(int index, Collection c)	지정된 위치부터 주어진 컬렉션의 모든 객체를 저장한다.
void addElement(Object obj)	객체(obj)를 저장한다. 반환값은 없다.
int capacity()	Vector의 용량을 반환한다.
void clear()	Vector를 비운다.
Object clone()	Vector를 복제한다.
boolean contains(Object elem) boolean containsAll(Collection c)	지정된 객체(elem) 또는 컬렉션(c)의 객체들이 Vector에 포함되어 있는지 확인한다.
void copyInto(Object[] anArray)	Vector에 저장된 객체들을 anArray배열에 저장한다.
Object elementAt(int index)	지정된 위치(index)에 저장된 객체를 반환한다.
Enumeration elements()	Vector에 저장된 모든 객체들을 반환한다.
void ensureCapacity(int minCapacity)	Vector의 용량이 최소한 minCapacity가 되도록 한다.
boolean equals(Object o)	주어진 객체(o)와 같은지 비교한다.
Object firstElement()	첫 번째로 저장된 객체를 반환한다.
Object get(int index)	지정된 위치(index)에 저장된 객체를 반환한다.
int hashCode()	해시코드를 반환한다.

1.4 Vector – 주요메서드(2/2)

int indexOf(Object elem)	지정된 객체가 저장된 위치를 찾아 반환한다.
int indexOf(Object elem, int index)	지정된 객체가 저장된 위치를 찾아 반환한다.(지정된 위치부터 찾는다.)
void insertElementAt(Object obj, int index)	객체(obj)를 주어진 위치(Index)에 삽입한다.
boolean isEmpty()	Vector가 비어있는지 확인한다.
Object lastElement()	Vector에 저장된 마지막 객체를 반환한다.
int lastIndexOf(Object elem)	객체(elem)가 저장된 위치를 끝에서부터 역방향으로 찾는다.
int lastIndexOf(Object elem, int index)	객체(elem)가 저장된 위치를 지정된 위치(index)부터 역방향으로 찾는다.
Object remove(int index)	지정된 위치(index)에 있는 객체를 제거한다.
boolean remove(Object o)	지정한 객체를 제거한다.
boolean removeAll(Collection c)	지정한 컬렉션에 저장된 것과 동일한 객체들을 Vector에서 제거한다.
void removeAllElements()	clear()와 동일하다.
boolean removeElement(Object obj)	지정된 객체를 삭제한다.
void removeElementAt(int index)	지정된 위치(index)에 저장된 객체를 삭제한다.
boolean retainAll(Collection c)	Vector에 저장된 객체 중에서 주어진 컬렉션과 공통된 것들만을 남기고 나머지는 삭제한다.
Object set(int index, Object element)	주어진 객체(element)를 지정된 위치(index)에 저장한다.
void setElementAt(Object obj, int index)	주어진 객체(obj)를 지정된 위치(index)에 저장한다.
void setSize(int newSize)	Vector의 크기를 지정한 크기(newSize)로 변경한다.
int size()	Vector에 저장된 객체의 개수를 반환한다.
List subList(int fromIndex, int toIndex)	fromIndex부터 toIndex사이에 저장된 객체를 반환한다.
Object[] toArray()	Vector에 저장된 모든 객체들을 객체배열로 반환한다.
Object[] toArray(Object[] a)	Vector에 저장된 모든 객체들을 객체배열 a에 담아 반환한다.
String toString()	저장된 모든 객체를 문자열로 출력한다.
void trimToSize()	용량을 크기에 맞게 줄인다.(빈 공간을 없앤다.)

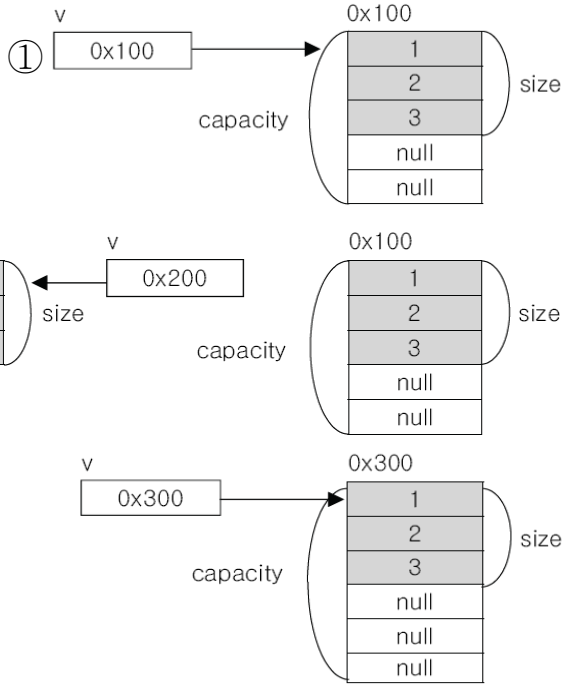
1.5 ArrayList사용예

```
ArrayList list1 = new ArrayList(10);
list1.add(new Integer(5));
list1.add(new Integer(4));
list1.add(new Integer(2));
list1.add(new Integer(0));
list1.add(new Integer(1));
list1.add(new Integer(3));

ArrayList list2 = new ArrayList(list1.subList(1,4));
print(list1, list2);

Collections.sort(list1);    // list1과 list2를 정렬한다.
Collections.sort(list2);    // Collections.sort(List l)
print(list1, list2);

System.out.println("list1.containsAll(list2):"
                           + list1.containsAll(list2));

list2.add("B");
list2.add("C");
list2.add(3, "A");
print(list1, list2);

list2.set(3, "AA");
print(list1, list2);

// list1에서 list2와 겹치는 부분만 남기고 나머지는 삭제한다.
System.out.println("list1.retainAll(list2):"
                           + list1.retainAll(list2));
print(list1, list2);

//  list2에서 list1에 포함된 객체들을 삭제한다.
for(int i= list2.size()-1; i >= 0; i--) {
    if(list1.contains(list2.get(i)))
        list2.remove(i);
}
print(list1, list2);
```

```
---------- java ----------
list1:[5, 4, 2, 0, 1, 3]
list2:[4, 2, 0]

list1:[0, 1, 2, 3, 4, 5]
list2:[0, 2, 4]

list1.containsAll(list2):true
list1:[0, 1, 2, 3, 4, 5]
list2:[0, 2, 4, A, B, C]

list1:[0, 1, 2, 3, 4, 5]
list2:[0, 2, 4, AA, B, C]

list1.retainAll(list2):true
list1:[0, 2, 4]
list2:[0, 2, 4, AA, B, C]

list1:[0, 2, 4]
list2:[AA, B, C]
```

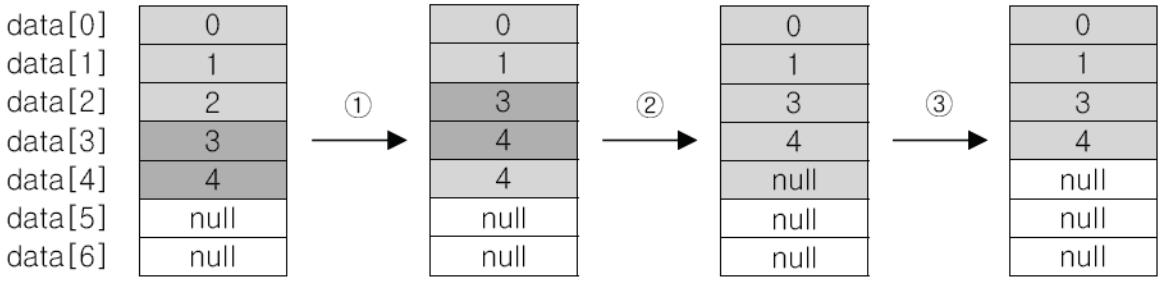
1.6 Vector의 크기(size)와 용량(capacity)

```
// 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();
```



1.7 Vector에 저장된 객체의 삭제과정

- Vector에 저장된 세 번째 데이터(data[2])를 삭제하는 과정. v.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--;
```

※ 삭제할 데이터가 마지막 데이터인 경우, ①의 과정은 필요없다.

1.8 ArrayList의 단점 - 배열의 단점

- 배열은 구조가 간단하고 데이터를 읽어오는 데 걸리는 시간(접근시간, access time)이 가장 빠르다는 장점이 있지만 단점도 있다.

▶ 단점 1 : 크기를 변경할 수 없다.

- 크기를 변경해야 하는 경우 새로운 배열을 생성하고 데이터를 복사해야 한다.(비용이 큰 작업)
- 크기 변경을 피하기 위해 충분히 큰 배열을 생성하면, 메모리 낭비가 심해진다.

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

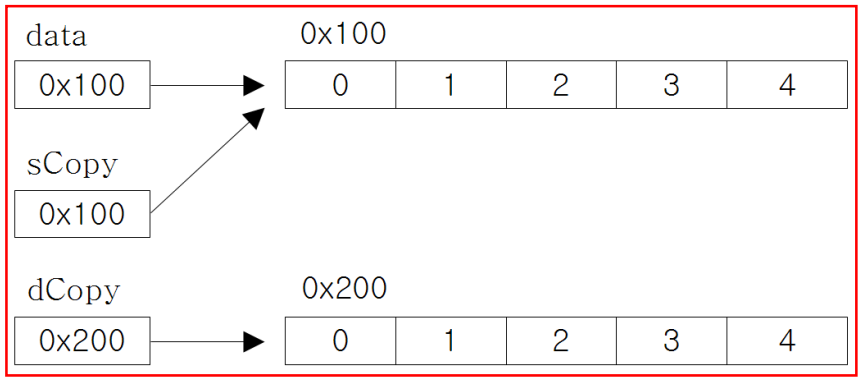
- 데이터를 추가하거나 삭제하기 위해, 많은 데이터를 옮겨야 한다.
- 그러나, 순차적인 데이터 추가(마지막에 추가)와 순차적으로 데이터를 삭제하는 것(마지막에서부터 삭제)은 빠르다.

1.9 Deep copy vs. Shallow copy

▶ Deep copy(깊은 복사) : 같은 내용의 새로운 객체를 생성.
원본의 변화에 복사본이 영향을 받지 않는다.

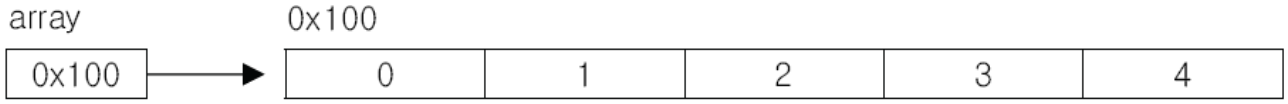
▶ Shallow copy(얕은 복사) : 참조변수만 복사. 원본의 변화에 복사본이 영향을 받는다.

```
public static void main(String args[]) {
    int[] data = {0,1,2,3,4};
    int[] sCopy = null;
    int[] dCopy = null;

    sCopy = shallowCopy(data);
    dCopy = deepCopy(data);
}

public static int[] shallowCopy(int[] objArray) {
    return objArray;
}

public static int[] deepCopy(int[] objArray) {
    if(objArray==null) return null;
    int[] result = new int[objArray.length];

    System.arraycopy(objArray, 0, result, 0, objArray.length);
    return result;
}
```

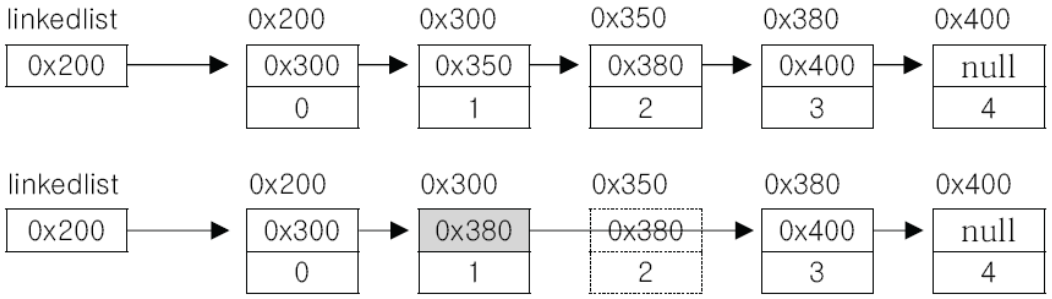


1.10 LinkedList - 배열의 단점을 보완

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

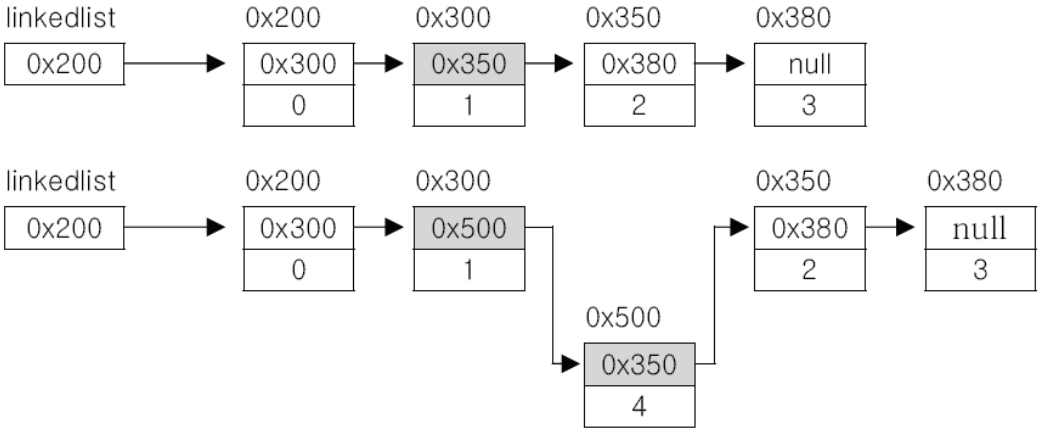


▶ 데이터의 삭제 : 단 한 번의 참조변경만으로 가능



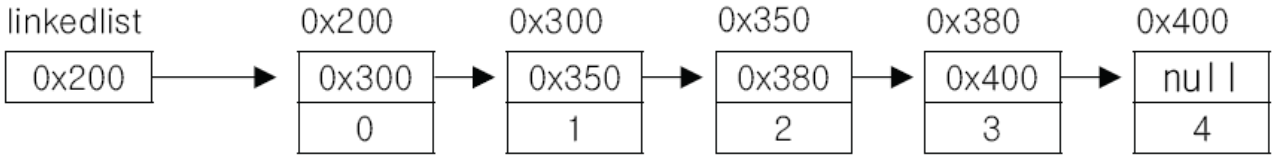

```
class Node {
    Node next;
    Object obj;
}
```

▶ 데이터의 추가 : 하나의 Node객체생성과 한 번의 참조변경만으로 가능



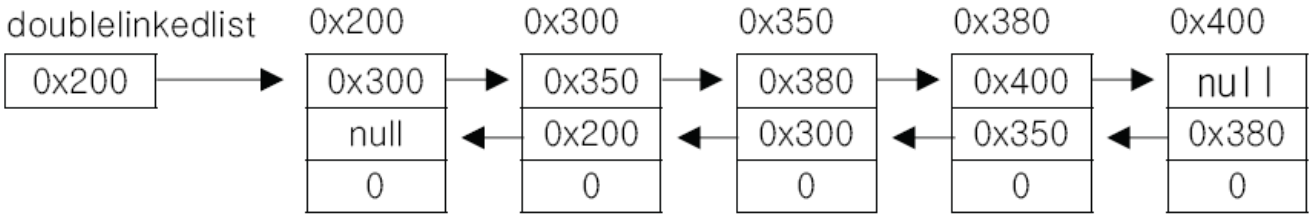
1.11 LinkedList – 이중 원형 링크드 리스트

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



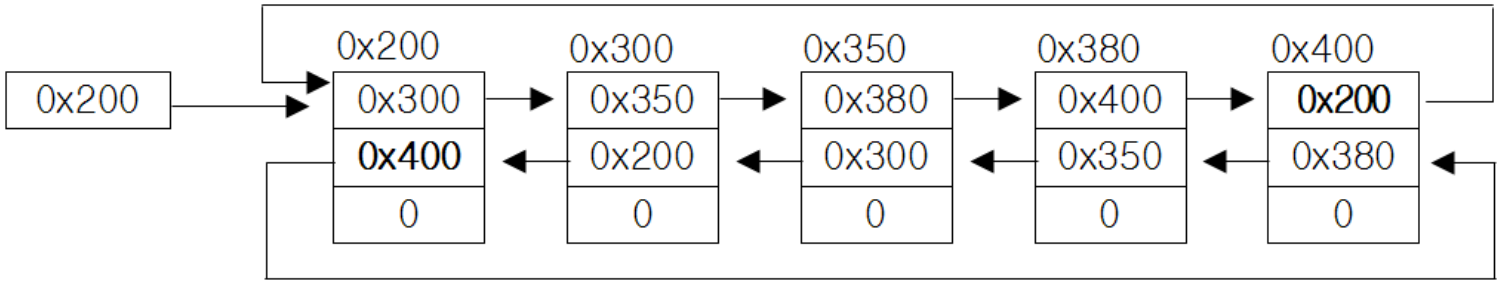

```
class Node {
    Node next;
    Object obj;
}
```

▶ 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



1.11 LinkedList – 주요메서드(1/2)

생성자 또는 메서드	설 명
LinkedList()	LinkedList객체를 생성한다.
LinkedList(Collection c)	주어진 컬렉션을 포함하는 LinkedList객체를 생성한다.
void add(int index, Object element)	지정된 위치(index)에 객체(element)를 추가한다.
boolean add(Object o)	지정된 객체(o)를 LinkedList의 끝에 추가한다.
boolean addAll(Collection c)	주어진 컬렉션에 포함된 모든 요소를 LinkedList의 끝에 추가한다.
boolean addAll(int index, Collection c)	지정된 위치(index)에 주어진 컬렉션에 포함된 모든 요소를 추가한다.
void addFirst(Object o)	객체(o)를 LinkedList의 첫 번째 요소 앞에 추가한다.
void addLast(Object o)	객체(o)를 LinkedList의 마지막 요소 뒤에 추가한다.
void clear()	LinkedList의 모든 요소를 삭제한다.
Object clone()	LinkedList를 복제해서 반환한다.
boolean contains(Object o)	지정된 객체가 LinkedList에 포함되어 있는지 알려준다.
Object get(int index)	지정된 위치(index)의 객체를 반환한다.
Object getFirst()	LinkedList의 첫 번째 요소를 반환한다.
Object getLast()	LinkedList의 마지막 요소를 반환한다.
int indexOf(Object o)	지정된 객체가 저장된 위치(앞에서 몇 번째)를 반환한다.
int lastIndexOf(Object o)	지정된 객체가 저장된 위치(뒤에서 몇 번째)를 반환한다.

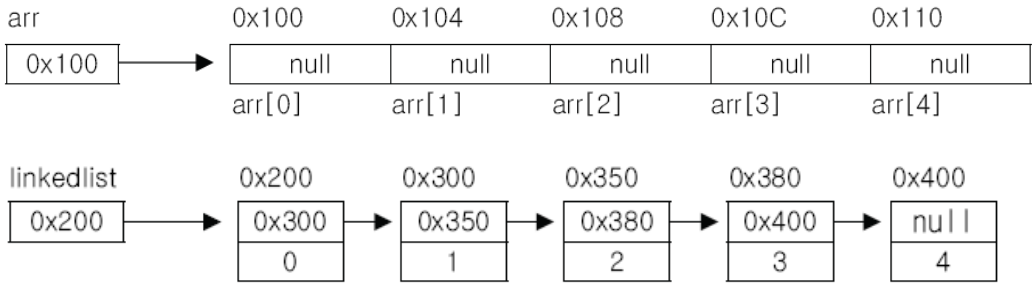
1.11 LinkedList – 주요메서드(2/2)

생성자 또는 메서드	설 명
ListIterator listIterator(int index)	지정된 위치에서부터 시작하는 ListIterator를 반환한다.
Object remove(int index)	지정된 위치(index)의 객체를 LinkedList에서 제거한다.
boolean remove(Object o)	지정된 객체를 LinkedList에서 제거한다.
Object removeFirst()	LinkedList의 첫 번째 요소를 제거한다.
Object removeLast()	LinkedList의 마지막 요소를 제거한다.
Object set(int index, Object element)	지정된 위치(index)의 객체를 주어진 객체로 바꾼다.
int size()	LinkedList에 저장된 객체의 수를 반환한다.
Object[] toArray()	LinkedList에 저장된 객체를 배열로 반환한다.
Object[] toArray(Object[] a)	LinkedList에 저장된 객체를 주어진 배열에 저장하여 반환한다.
E element()	LinkedList의 첫 번째 요소를 반환한다.
boolean offer(E o)	지정된 객체(o)를 LinkedList의 끝에 추가한다.
E peek()	LinkedList의 첫 번째 요소를 반환한다.
E poll()	LinkedList의 첫 번째 요소를 반환한다.(LinkedList에서는 제거된다.)
E remove()	LinkedList의 첫 번째 요소를 제거한다.

1.12 ArrayList vs. LinkedList

- 순차적으로 데이터를 추가/삭제하는 경우, ArrayList가 빠르다.
- 비순차적으로 데이터를 추가/삭제하는 경우, LinkedList가 빠르다.
- 접근시간(access time)은 ArrayList가 빠르다.

n번째 데이터의 주소 = 배열의 주소 + n * 데이터 타입의 크기



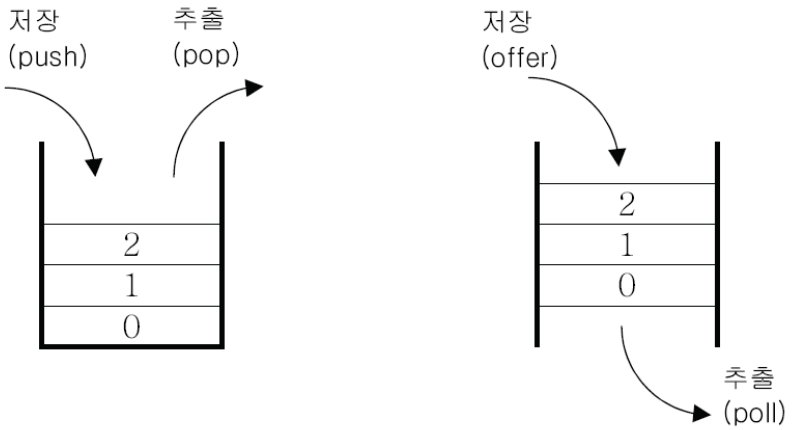

```
= 순차적으로 추가하기 =
ArrayList :591
LinkedList :721

= 순차적으로 삭제하기 =
ArrayList :10
LinkedList :50

= 중간에 추가하기 =
ArrayList :3465
LinkedList :10

= 중간에서 삭제하기 =
ArrayList :3415
LinkedList :10

= 접근시간테스트 =
ArrayList :10
LinkedList :3195
```

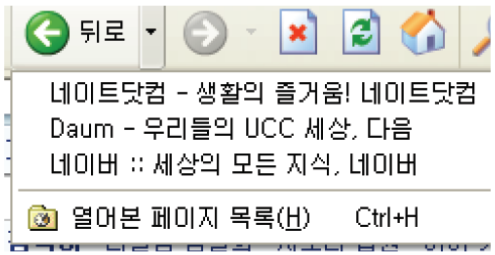
컬렉션	읽기(접근시간)	추가 / 삭제	비 고
ArrayList	빠르다	느리다	순차적인 추가삭제는 빠름. 비효율적인 메모리사용
LinkedList	느리다	빠르다	데이터가 많을수록 접근성이 떨어짐

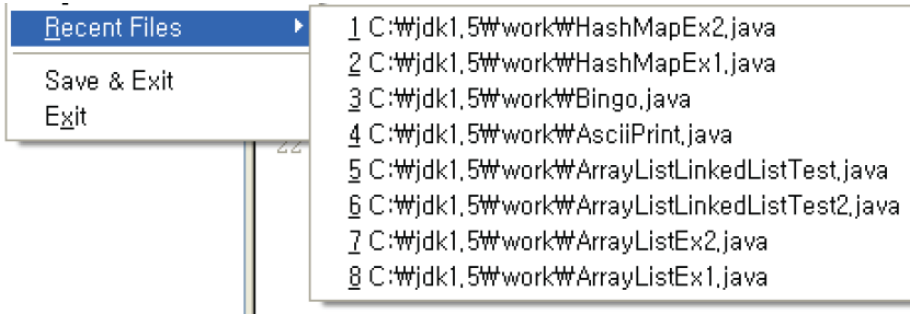
1.13 스택과 큐(Stack & Queue)

▶ 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)







1.14 Enumeration, Iterator, ListIterator

- 컬렉션 클래스에 저장된 데이터를 접근하는데 사용되는 인터페이스이다.
- Enumeration는 Iterator의 구버젼이다.
- Iterator의 접근성을 향상시킨 것이 ListIterator이다.(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)

[표11-12] Iterator인터페이스의 메서드

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

[표11-13] Enumeration인터페이스의 메서드

1.15 Iterator

- 컬렉션 클래스에 저장된 요소들을 나열하는 방법을 제공.
- 컬렉션 클래스의 iterator()를 호출해서 Iterator를 구현한 객체를 얻는다.

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)

[표11-12] Iterator인터페이스의 메서드

```
List list = new ArrayList();    // 다른 컬렉션으로 변경할 때는 이 부분만 고치면 된다.

Iterator it = list.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}

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

```
Map map = new HashMap();
...
Iterator it = map.ketSet().iterator();
```

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

1.16 ListIterator – Iterator의 기능을 확장(상속)

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

```
List list = new ArrayList();
ListIterator lit = list.listIterator();

while(lit.hasNext()) {
    Object obj = lit.next();
    System.out.println(obj);
}
```

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

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

메서드	설 명
void add(Object o)	컬렉션에 새로운 객체(o)를 추가한다.(선택적 기능)
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 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)

1.17 HashSet – 중복허용×, 순서유지×

- Set인터페이스를 구현한 대표적인 컬렉션 클래스.(중복허용×, 순서유지×)
- 순서를 유지하고자 한다면, LinkedHashSet클래스를 사용해야 한다.

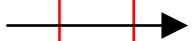
생성자 또는 메서드	설 명
HashSet()	HashSet객체를 생성한다.
HashSet(Collection c)	주어진 컬렉션을 포함하는 HashSet객체를 생성한다.
HashSet(int initialCapacity)	주어진 값을 초기용량으로하는 HashSet객체를 생성한다.
HashSet(int initialCapacity, float loadFactor)	초기용량과 load factor를 지정하는 생성자.
boolean add(Object o)	새로운 객체를 저장한다.
boolean addAll(Collection c)	주어진 컬렉션에 저장된 모든 객체들을 추가한다.(합집합)
void clear()	저장된 모든 객체를 삭제한다.
Object clone()	HashSet을 복제해서 반환한다.
boolean contains(Object o)	지정된 객체를 포함하고 있는지 알려준다.
boolean containsAll(Collection c)	주어진 컬렉션에 저장된 모든 객체들을 포함하고 있는지 알려준다.
boolean isEmpty()	HashSet이 비어있는지 알려준다.
Iterator iterator()	Iterator를 반환한다.
boolean remove(Object o)	지정된 객체를 HashSet에서 삭제한다.(성공하면 true, 실패하면 false)
boolean removeAll(Collection c)	주어진 컬렉션에 저장된 모든 객체와 동일한 것들을 HashSet에서 모두 삭제한다.(차집합)
boolean retainAll(Collection c)	주어진 컬렉션에 저장된 객체와 동일한 것만 남기고 삭제한다.(교집합)
int size()	저장된 객체의 개수를 반환한다.
Object[] toArray()	저장된 객체들을 객체배열의 형태로 반환한다.
Object[] toArray(Object[] a)	저장된 객체들을 주어진 객체배열(a)에 담는다.

1.17 HashSet – boolean add(Object o)

- HashSet에 객체를 저장할 때 boolean add(Object o)를 사용하며, 기존에 저장된 객체와 중복된 객체를 저장하면 false를 반환한다.
- boolean add(Object o)는 저장할 객체의 equals()와 hashCode()를 호출하므로 저장할 객체의 equals()와 hashCode()가 적절히 오버라이딩되어 있어야 한다. 그렇지 않으면 중복된 객체를 중복된 것으로 간주하지 않을 수 있다.

```
public static void main(String[] args) {
    HashSet set = new HashSet();

    set.add("abc");
    set.add("abc");
    set.add(new Person("David",10));
    set.add(new Person("David",10));

    System.out.println(set);
}
```

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

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

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



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

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

    public boolean equals(Object obj) {
        if(obj instanceof Person) {
            Person tmp = (Person)obj;
            return name.equals(tmp.name) && age==tmp.age;
        }

        return false;
    }

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

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

1.17 HashSet – int hashCode()

- 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.

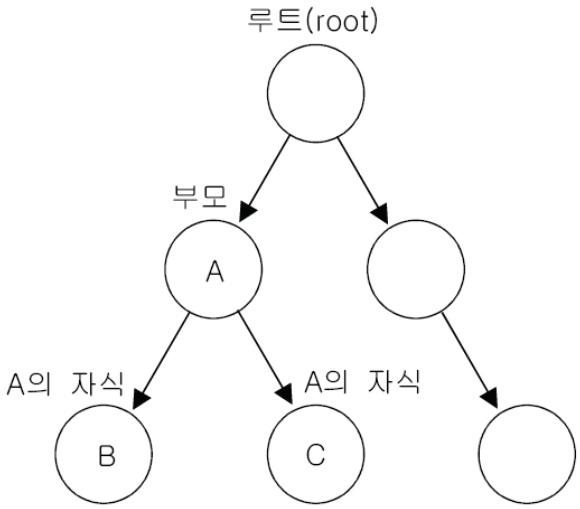
```
Person2 p1 = new Person2("David", 10);
Person2 p2 = new Person2("David", 10);

boolean b = p1.equals(p2);

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

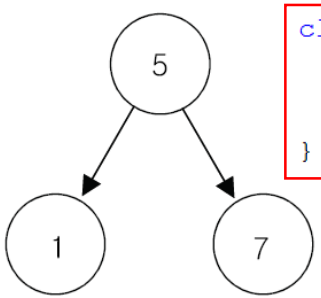
- equals()로 비교해서 false를 얻은 두 객체의 hashCode()값은 서로 다른 것이 보통이지만, 같아도 문제없다.

  그러나 성능향상을 위해 가능하면 서로 다른 값을 반환하도록 오버라이딩하자.

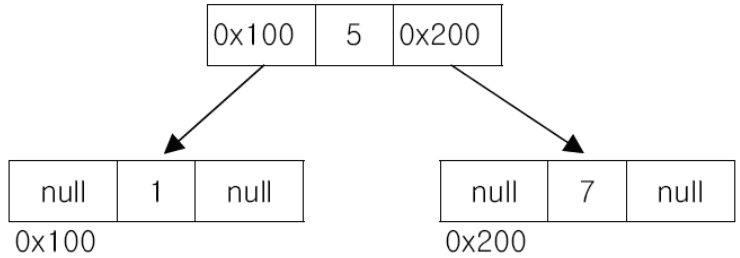
1.18 TreeSet – 검색과 정렬에 유리

- Set인터페이스를 구현한 컬렉션 클래스.(중복허용×, 순서유지×, 정렬저장○)
- 이진검색트리(binary search tree - 정렬, 검색에 유리)의 구조로 되어있다.
- 링크드리스트와 같이 각 요소(node)가 나무(tree)형태로 연결된 구조
- 모든 트리는 하나의 루트(root node)를 가지며, 서로 연결된 두 요소를 '부모-자식관계'에 있다 하고, 하나의 부모에 최대 두 개의 자식을 갖는다.
- 왼쪽 자식의 값은 부모의 값보다 작은 값을, 오른쪽 자식의 값은 부모보다 큰 값을 저장한다.
- 검색과 정렬에 유리하지만, HashSet보다 데이터 추가, 삭제시간이 더 걸린다.






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

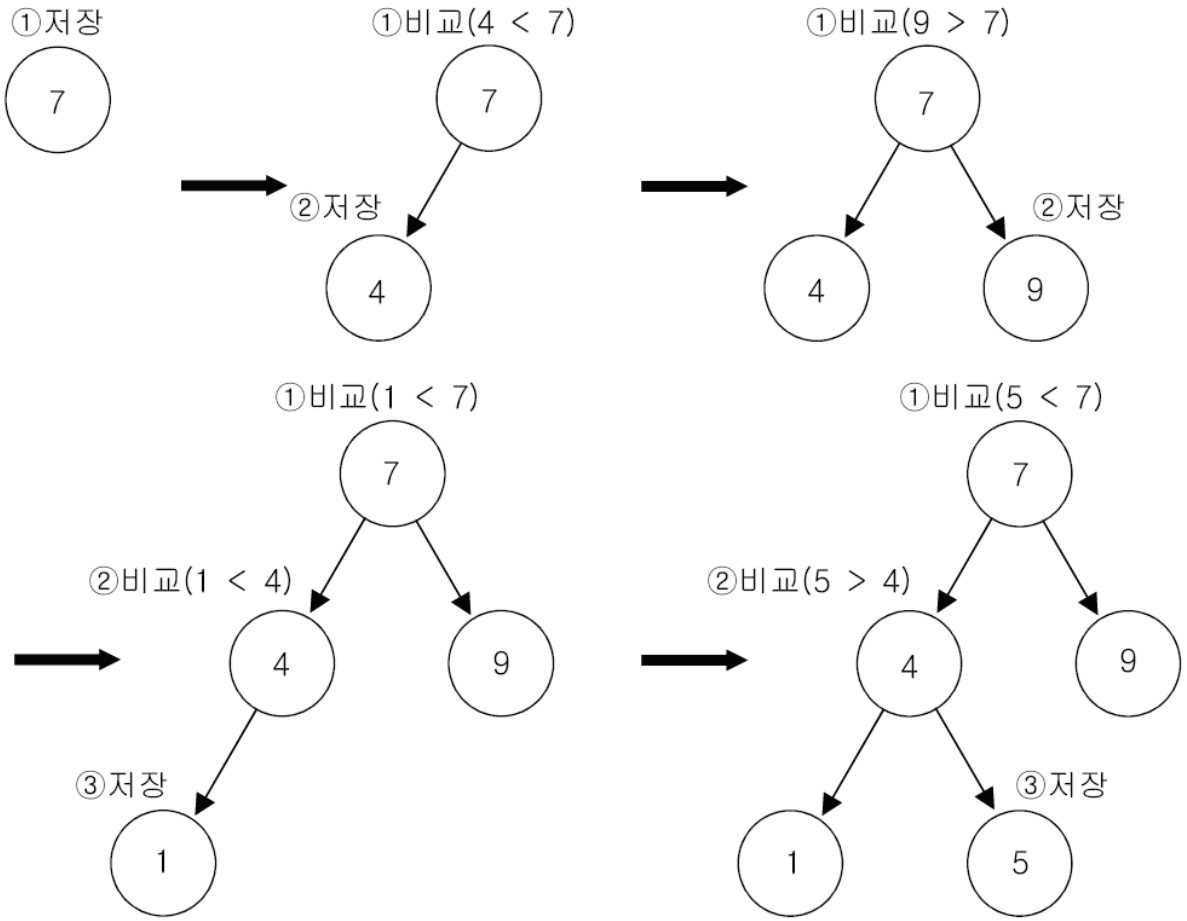


1.18 TreeSet – 주요메서드

생성자 또는 메서드	설 명
TreeSet()	기본생성자
TreeSet(Collection c)	주어진 컬렉션을 저장하는 TreeSet을 생성
TreeSet(Comparator c)	주어진 정렬조건으로 정렬하는 TreeSet을 생성
TreeSet(SortedSet s)	주어진 SortedSet을 구현한 컬렉션을 저장하는 TreeSet을 생성
boolean add(Object o) boolean addAll(Collection c)	지정된 객체(o) 또는 Collection(c) 의 객체들을 Collection에 추가한다.
void clear()	저장된 모든 객체를 삭제한다.
Object clone()	TreeSet을 복제하여 반환한다.
Comparator comparator()	TreeSet의 정렬기준(Comparator)를 반환한다.
boolean contains(Object o) boolean containsAll(Collection c)	지정된 객체(o) 또는 Collection의 객체들이 포함되어 있는지 확인한다.
Object first()	정렬된 순서에서 첫 번째 객체를 반환한다.
SortedSet headSet(Object toElement)	지정된 객체보다 작은 값의 객체들을 반환한다.
boolean isEmpty()	TreeSet이 비어있는지 확인한다.
Iterator iterator()	TreeSet의 Iterator를 반환한다.
Object last()	정렬된 순서에서 마지막 객체를 반환한다.
boolean remove(Object o)	지정된 객체를 삭제한다.
boolean retainAll(Collection c)	주어진 컬렉션과 공통된 요소만을 남기고 삭제한다.(교집합)
int size()	저장된 객체의 개수를 반환한다.
SortedSet subSet(Object fromElement, Object toElement)	범위 검색(fromElement와 toElement사이)의 결과를 반환한다.(끝 범위인 toElement는 범위에 포함되지 않음)
SortedSet tailSet(Object fromElement)	지정된 객체보다 큰 값의 객체들을 반환한다.
Object[] toArray()	저장된 객체를 객체배열로 반환한다.
Object[] toArray(Object[] a)	저장된 객체를 주어진 객체배열에 저장하여 반환한다.

1.18 TreeSet – 데이터 저장과정

※만일 TreeSet에 7,4,9,1,5의 순서로 데이터를 저장한다면, 다음과 같은 과정을 거치게 된다.



1.19 Comparator와 Comparable

▶ 객체를 정렬하는데 필요한 메서드를 정의한 인터페이스이다.

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

▶ 이 들에 정의된 compare(), compareTo()를 구현함으로써 정렬이 필요한 경우, 예를 들면 TreeSet이나 sort()를 사용할 때, 정렬기준을 제시하게 된다.

```
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
}
```

▶ compare()와 compareTo()는 이름과 매개변수의 수만 다를 뿐, 두 객체를 비교해서 같으면 0을 작으면 음수, 크면 양수를 반환한다는 것은 같다.

그리고 이 반환값을 통해 두 객체의 정렬순서가 결정된다.

※ equals메서드는 모든 클래스가 가지고 있지만, Comparator를 구현하는 클래스는 equals메서드의 오버라이딩이 필요할 수도 있다는 것을 알리기 위해 정의한 것일 뿐, 대부분의 경우 compare(Object o1, Object o2)만 구현하면 된다.

1.19 Comparator와 Comparable – 구현예(examples)

```
public static void main(String[] args) {
    TreeSet set1 = new TreeSet(); // 기본정렬(Comparable) 사용
    TreeSet set2 = new TreeSet(new Descending()); // TreeSet(Comparator c)

    int[] score = {30, 50, 10, 20, 40};

    for(int i=0; i < score.length; i++) {
        set1.add(new Integer(score[i]));
        set2.add(new Integer(score[i]));
    }
}
```

```
---------- java ----------
set1 :[10, 20, 30, 40, 50]
set2 :[50, 40, 30, 20, 10]
```

```
public final class Integer extends Number implements Comparable {
    ...
    public int compareTo(Integer anotherInteger) {
        int thisVal = this.value;
        int anotherVal = anotherInteger.value;
        // 비교하는 값이 크면 -1, 같으면 0, 작으면 1을 반환한다.
        return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 : 1));
    }
    ...
}
```

```
class Descending implements Comparator {
    public int compare(Object o1, Object o2){
        if( o1 instanceof Comparable && o2 instanceof Comparable) {
            Comparable c1 = (Comparable)o1;
            Comparable c2 = (Comparable)o2;
            return c1.compareTo(c2) * -1 ; // 기본 정렬방식의 반대로 변경.
        }
        return -1;
    }
}
```

1.20 Hashtable과 HashMap

- HashMap은 Hashtable의 신버젼이며, Hashtable과 달리 HashMap은 동기화처리가 되어 있지 않다. 가능하면 Hashtable보다는 HashMap을 사용하자.
- HashMap은 해싱(hashing)기법을 사용해서 데이터를 저장하기 때문에 많은 양의 데이터를 검색할 때 성능이 뛰어나다.
- HashMap은 Map인터페이스를 구현하였으며, 데이터를 키와 값의 쌍으로 저장한다.

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

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

키(key)	값(value)
castello	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

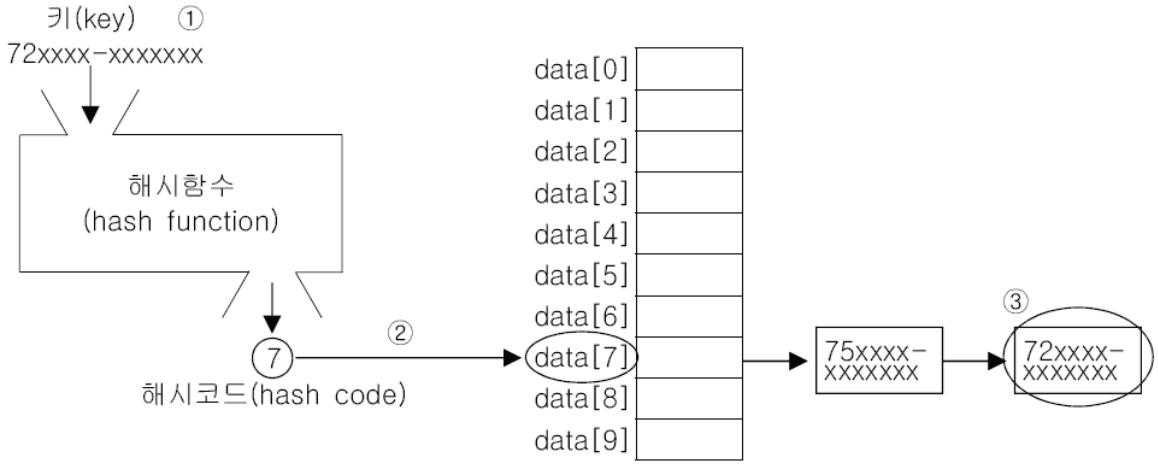
Object[] key;
Object[] value;
	Entry[] table;
	class Entry {
	  Object key;
	  Object value;
	}

1.21 HashMap - 주요메서드

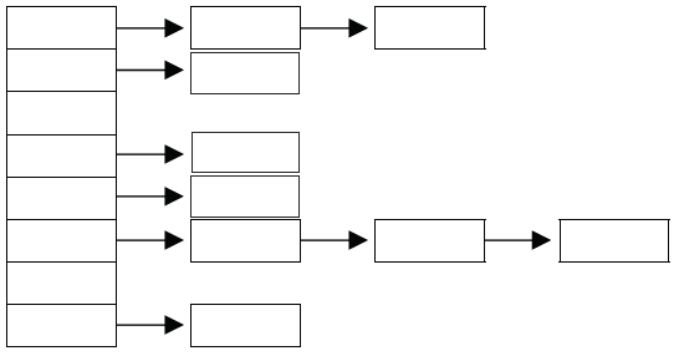
생성자 / 메서드	설명
HashMap()	HashMap객체를 생성한다.
HashMap(int initialCapacity)	지정된 값을 초기용량으로 하는 HashMap객체를 생성한다.
HashMap(int initialCapacity, float loadFactor)	지정된 초기용량과 load factor의 HashMap객체를 생성한다.
HashMap(Map m)	주어진 Map에 저장된 모든 요소를 포함하는 HashMap을 생성한다.
void clear()	HashMap에 저장된 모든 객체를 제거한다.
Object clone()	현재 HashMap을 복제해서 반환한다.
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)
Set entrySet()	HashMap에 저장된 키와 값을 엔트리(키와 값의 결합)의 형태로 Set에 저장해서 반환한다.
Object get(Object key)	지정된 키(key)의 값(객체)을 반환한다.
boolean isEmpty()	HashMap이 비어있는지 알려준다.
Set keySet()	HashMap에 저장된 모든 키가 저장된 Set을 반환한다.
Object put(Object key, Object value)	지정된 키와 값을 HashMap에 저장한다.
void putAll(Map m)	Map에 저장된 모든 요소를 HashMap에 저장한다.
Object remove(Object key)	HashMap에서 지정된 키로 저장된 값(객체)를 제거한다.
int size()	HashMap에 저장된 요소의 개수를 반환한다.
Collection values()	HashMap에 저장된 모든 값을 컬렉션의 형태로 반환한다.

1.22 해싱(hashing) - (1/2)

- 해시함수(hash function)를 이용해서 해시테이블(hash table)에 저장하고 검색하는 기법

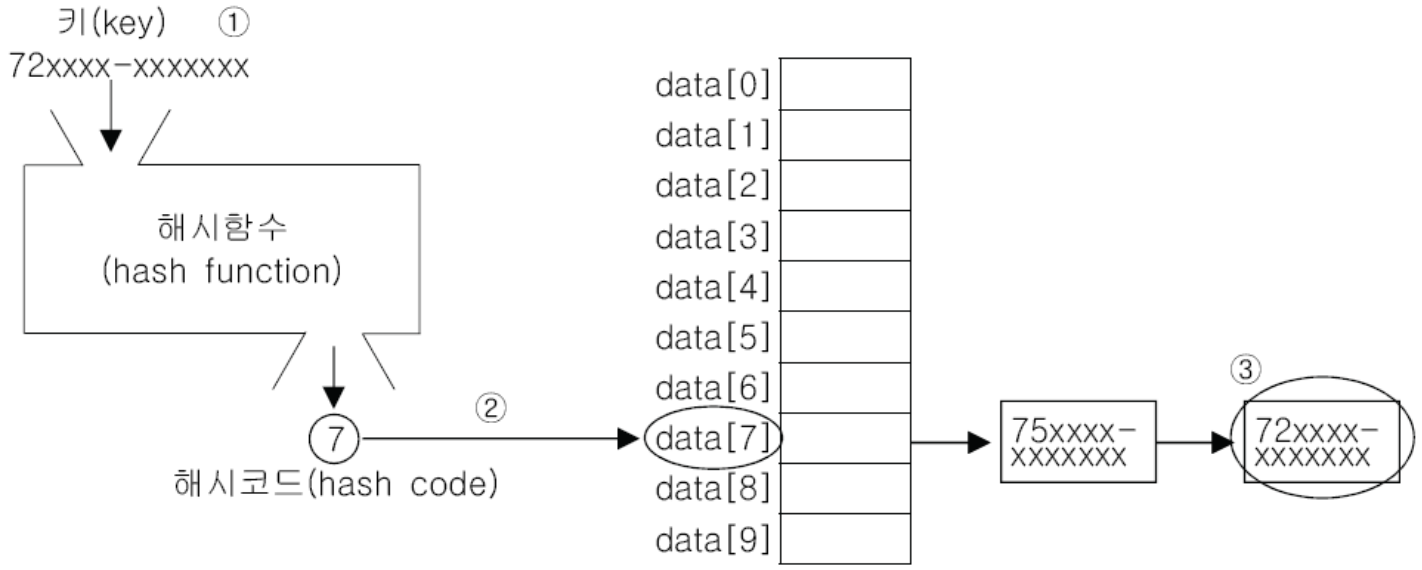


- 해싱에 사용되는 자료구조는 배열과 링크드리스트가 조합된 형태이다.



1.22 해싱(hashing) - (2/2)

▶ 키를 이용해서 해시테이블로부터 데이터를 가져오는 과정



① 키를 이용해서 해시함수를 호출한다.

② 해시함수의 호출결과인 해시코드에 대응하는 배열에 저장된 링크드리스트를 찾는다.

③ 링크드리스트에서 키와 일치하는 데이터를 찾는다.

※ 해시함수는 같은 키값에 대해 항상 같은 해시코드를 반환해야한다.

서로 다른 키값일지라도 같은 값의 해시코드를 반환할 수 있다.

1.23 TreeMap

- 이진검색트리의 형태로 키와 값의 쌍으로 이루어진 데이터를 저장한다.
- Map의 장점인 빠른 검색과 Tree의 장점인 정렬과 범위검색의 장점을 모두 갖고 있다.
- 이진검색트리처럼, 데이터를 저장할 때 정렬하기 때문에 저장시간이 길다는 단점을 가지고 있다.
- 정렬된 상태로 데이터를 조회하는 경우가 빈번하다면, 데이터를 조회할 때 정렬해야 하는 HashMap보다는 이미 정렬된 상태로 저장되어 있는 TreeMap이 빠른 조회결과를 얻을 수 있다.
- 주로 HashMap을 사용하고, 정렬이나 범위검색이 필요한 경우에만 TreeMap을 사용하는 것이 좋다.

1.23 TreeMap – 주요메서드

메서드	설명
TreeMap()	TreeMap객체를 생성한다.
TreeMap(Comparator c)	지정된 Comparator를 기준으로 정렬하는 TreeMap객체를 생성한다.
TreeMap(Map m)	주어진 Map에 저장된 모든 요소를 포함하는 TreeMap을 생성한다.
TreeMap(SortedMap m)	주어진 SortedMap에 저장된 모든 요소를 포함하는 TreeMap을 생성한다.
void clear()	TreeMap에 저장된 모든 객체를 제거한다.
Object clone()	현재 TreeMap을 복제해서 반환한다.
Comparator comparator()	TreeMap을 정렬기준이 되는 Comparator를 반환한다. Comparator가 지정되지 않았다면 null이 반환된다.
boolean containsKey(Object key)	TreeMap에 지정된 키(key)가 포함되어있는지 알려준다.
boolean containsValue(Object value)	TreeMap에 지정된 값(value)가 포함되어있는지 알려준다.
Set entrySet()	TreeMap에 저장된 키와 값을 엔트리(키와 값의 결합)의 형태로 Set에 저장해서 반환한다.
Object firstKey()	TreeMap에 저장된 첫번째 요소의 키를 반환한다.
Object get(Object key)	지정된 키(key)의 값(객체)을 반환한다.
SortedMap headMap(Object toKey)	TreeMap에 저장된 첫번째 요소부터 지정된 요소까지의 범위에 속한 모든 요소가 담긴 SortedMap을 반환한다.(toKey는 포함되지 않는다.)
Set keySet()	TreeMap에 저장된 모든 키가 저장된 Set을 반환한다.
boolean isEmpty()	TreeMap이 비어있는지 알려준다.
Object lastKey()	TreeMap에 저장된 마지막 요소의 키를 반환한다.
Object put(Object key, Object value)	지정된 키와 값을 TreeMap에 저장한다.
void putAll(Map map)	Map에 저장된 모든 요소를 TreeMap에 저장한다.
Object remove(Object key)	TreeMap에서 지정된 키로 저장된 값(객체)를 제거한다.
int size()	TreeMap에 저장된 요소의 개수를 반환한다.
SortedMap subMap(Object fromKey, Object toKey)	지정된 두 개의 키 사이에 있는 요소가 담긴 SortedMap을 반환한다.(toKey는 포함되지 않는다.)
SortedMap tailMap(Object fromKey)	지정된 키부터 마지막 요소의 범위에 속한 요소가 담긴 SortedMap을 반환한다.
Collection values()	TreeMap에 저장된 모든 값을 컬렉션의 형태로 반환한다.

1.24 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.

1.24 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();
        }
    }
}
```

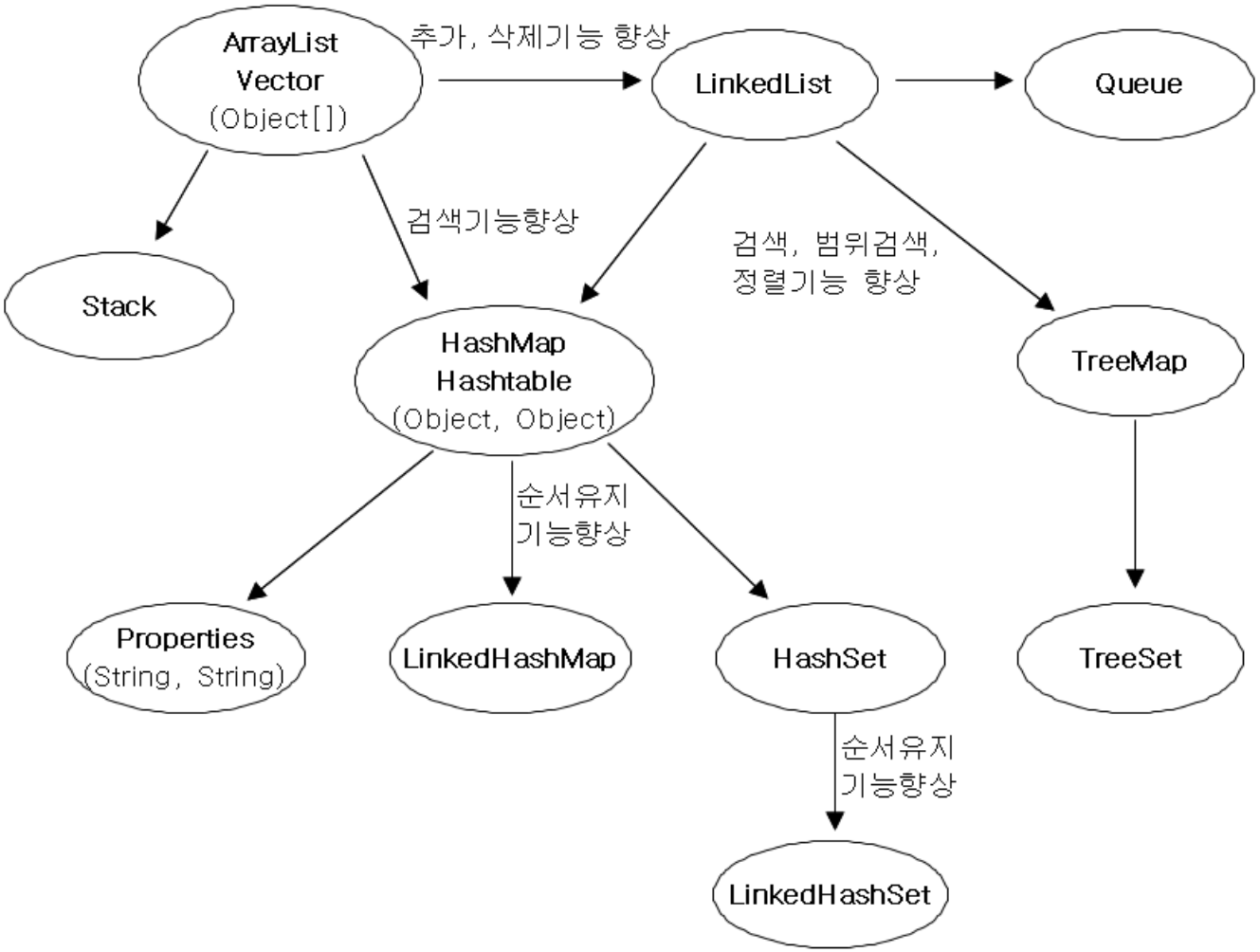
[output.txt]

```
#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>
```

1.25 컬렉션 클래스 정리 & 요약 (1/2)



1.25 컬렉션 클래스 정리 & 요약 (2/2)

컬렉션	특 징
ArrayList	배열기반, 데이터의 추가와 삭제에 불리, 순차적인 추가삭제는 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
Queue	LinkedList가 Queue인터페이스를 구현
Properties	Hashtable을 상속받아 구현
HashSet	HashMap을 이용해서 구현
TreeSet	TreeMap을 이용해서 구현
LinkedHashMap LinkedHashSet	HashMap과 HashSet에 저장순서유지기능을 추가하였음.

감사합니다.

더 많은 동영상강좌를 아래의 사이트에서 구하실 수 있습니다.

http://www.javachobo.com

관련문의 : 남궁성 castello@naver.com