Esercizio su iteratore per IntSet

In riferimento agli esempi visti in aula, completare e modificare la specifica e l'implementazione di un iteratore per il tipo di dato astratto `IntSet` sotto riportato.

In particolare occorre modificare l'iteratore in modo da fargli produrre gli elementi dell'insieme in ordine crescente (la prima chiamata a `it.next()` dovrà produrre l'elemento minimo del set, la seconda il successivo in ordine al minimo, etc).

```
public class IntSet {
    /** MISSION
     * This class provides an ADT for sets of int.
        IntSet is mutable, unbounded. */

    /**
     * ABSTRACTION FUNCTION: the set is represented by the items in this.elements
     *
     * INVARIANT elements != null & elements contains no duplicates &
     * elements contains boxed int (Integer).
     * Elements is not sorted.
     */
    private Vector<Integer> elements;

    /** EFFECT; initialize this to a new set, empty. */
    public IntSet(){
        this.elements = new Vector<Integer>();
    }

    /** @param: elts. REQUIRE be not null.
        EFFECT initialize this to  a new set, which contains each element
        of elts; duplicated elements are not considered. */
    public IntSet(int [] elts){
        this.elements = new Vector<Integer>();
        Objects.requireNonNull(elts);
        for (int x:elts){
            Integer y = new Integer(x);
            if (!this.elements.contains(y)){
                this.elements.addElement(y);
            }
        }
    }

    /**
     * Copy constructor.
     * EFFECT initialize this to a new set that contains all and only
```

```
     * the elements of s.
     * @param s: a set to be duplicated. REQUIRE not null.
     */
    @SuppressWarnings("unchecked")
    public IntSet(IntSet s){
        Objects.requireNonNull(s);
        this.elements = (Vector<Integer>) s.elements.clone();
    }

    /** insert x in this
     *  MODIFY this: x is added to this set if x is not present */
    public void insert(int x){
        Integer y = new Integer(x);
        if (!this.elements.contains(y)){
            this.elements.addElement(y);
        }
        assert (this.elements.contains(y));
    }

    /** remove x from this
     *  MODIFY this: x is removed to this set if x is present
        @return: true if x was removed */
    public boolean remove(int x){
        Integer y = new Integer(x);
        boolean res = this.elements.remove(y);
        assert (!this.elements.contains(y));
        return(res);
    }

    /** check if x belongs to this
     *  @return: true if x is present in this */
    public boolean isIn(int x){
        Integer y = new Integer(x);
        int i = this.indexOf(y);
        boolean res = (i>=0);
        assert (!res || this.elements.contains(y)): "res implies (y in elements)";
        return(res);
    }

    /** @return: the index of x if it is present in this ; return -1 if not present */
    private int indexOf(Integer x){
        assert (this.elements!=null);
        for (int i=0; i< this.elements.size();i++){
            if (this.elements.get(i).equals(x)){
                return(i);
            }
```

```
    }
    return(-1);
}

/** cardinality of this
 *  @return: the number of elements in this */
public long size(){
    return (this.elements.size());

}

/** choose an element of this
 *  @return: a random element in this
    @throws: EmptyIntSetException if this is empty */
public int choose() throws EmptyIntSetException {
    if (this.elements.isEmpty()){
        throw new EmptyIntSetException();
    }
    Random randomGenerator = new Random();
    int x = randomGenerator.nextInt(this.elements.size());
    return (this.elements.elementAt(x));
}

/**
 * @param s2: REQUIRE not null
 * @return true if this and s2 contain the same set of int
 */
public boolean sameValues(IntSet s2){
    Objects.requireNonNull(s2);
    Collections.sort(this.elements);// BEWARE integers are moved
    Collections.sort(s2.elements);
    boolean res = this.elements.equals(s2.elements);
    return(res);
}

/**
 * @return a standard iterator over the Integers of this set. The iterator
 * is not sensible to mutations of this set.
 */
public Iterator<Integer> iterator(){
    assert (this.elements!=null);
    IntSetIterator result = new IntSetIterator(this);
    return ((Iterator<Integer>) result);
}
/** ============================================
 *            INNER CLASS for generic iterator
```

```
 *  ==========================================
 */
/**
 * MISSION is to provide an iterator over the elements
 * of an IntSet.
 * Once the iterator is created, if the original set changes
 * the iterator continues to work on the original copy.
 */
private class IntSetIterator implements Iterator<Integer> {

    /** INVARIANT
     * elements contains a copy of the elements of the IntSet when
     * this iterator is created.
     * current is the integer i such that elements[i] is the first of the elements
     * not yet returned. current==elements.size if we explored all of them.
     */
    private int current;
    final private Vector<Integer> elements;

    /**
     * @param s an IntSet REQUIRE not null
     * Initialize the iterator with the size of the vector and
     * with current index=0, and store a copy of the elements.
     */
    @SuppressWarnings("unchecked")
    IntSetIterator(IntSet s){
        Objects.requireNonNull(s);
        this.elements = (Vector<Integer>) s.elements.clone();
        this.current = 0;
    }

    @Override
    public boolean hasNext() {
        return (this.current < this.elements.size());
    }

    @Override
    public Integer next() {
        if (this.current < this.elements.size()){
            Integer res = this.elements.get((int) this.current);
            this.current++;
            return (res);
        } else {
            throw new NoSuchElementException("Went beyond the available values");
        }
    }
```

```
        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```