```java
package adt;

import entity.Tutor;
import java.io.Serializable;
import java.util.Comparator;
import java.util.Iterator;

/**
 *
 * @author Chew Lip Sin
 * @author Lim Yi Leong
 * @author Eugene Teoh
 * @param <T> type of elements stored in the stack.
 */
public class ArrList<T> implements ListInterface<T>, Serializable {

    private T[] arr;
    private int numberOfEntries;
    private static final int DEFAULT_CAPACITY = 20;

    /**
     * Constructs a new list with the default capacity.
     */
    public ArrList() {
        this(DEFAULT_CAPACITY);
    }

    /**
     * Constructs a new list with the specified initial capacity.
     *
     * @param initialCapacity The initial capacity of the list.
     */
    public ArrList(int initialCapacity) {
        numberOfEntries = 0;
        arr = (T[]) new Object[initialCapacity];
    }

    /**
     * Adds the specified element to the end of the list.
     *
     * @param newEntry The element to add.
     * @return true if the addition is successful, or false if the list is full
     * Description: Adds a new entry to the end of the list. Entries currently
     * in the list are unaffected. The lists size is increased by 1.
     * Precondition: newEntry is not null. Post-condition:The entry has been
     * added to the list.
     *
     */
    @Override
    public boolean add(T newEntry) {
        if (isArrayFull()) {
            //double the arry list if array list is full
            doubleArray();
        }
        arr[numberOfEntries] = newEntry;
        numberOfEntries++;
        return true;
    }

    /**
     * Adds the specified element to the list at the specified position.
     *
     * @param newPosition The position to add the element at.
     * @param newEntry The element to add.
     * @return true if the element was added successfully, false otherwise. *
     * Description: Adds a new entry at a specified position within the list.
     * Entries originally at and above the specified position are at the next
     * higher position within the list. The list size is increased by 1.
     * Precondition: newPosition >= 1 and newPosition smaller equal than
     * getLength()+1newEntry is not null. Post-condition:newEntry is added to
     * the list in the given position. The old entries have been shifted up one
     * position.
     */
    @Override
    public boolean add(int newPosition, T newEntry) {
        boolean isSuccessful = true;

        if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1)) {
            if (isArrayFull()) {
                //double the arry list if array list is full
                doubleArray();
            }
            makeRoom(newPosition);
            arr[newPosition - 1] = newEntry;
            numberOfEntries++;
        } else {
            isSuccessful = false;
        }

        return isSuccessful;
    }

```

```java
    /**
     * Adds all of the elements in the specified array to the end of the list.
     *
     * @param newElements The array of elements to add.
     * @return true if all of the elements were added successfully, false
     * otherwise. Precondition:newElements must not be null. Post-condition:
     */
    @Override
    public boolean addAll(T... newElements) {
        if (newElements != null) {
            if (isElementsValid(newElements)) {
                for (T element : newElements) {
                    add(element);
                }
                return true;
            }
        }
        return false;
    }

    /**
     * Post-condition:The list is empty. Description:Removes all entries from
     * the list.
     */
    @Override
    public void clear() {
        numberOfEntries = 0;
    }

    /**
     * Checks whether the list contains the specified element.
     *
     *
     * @param anEntry The element to check for.
     * @return true if the list contains the element, false otherwise.
     * Description: This method finds whether the new Entry exists or not.
     * Precondition: The array must exist. Post-condition:The array remains
     * unchanged
     *
     *
     */
    @Override
    public boolean contains(T anEntry) {
        boolean found = false;
        //!found = true and index should smaller than the length of array list
        for (int index = 0; !found && (index < numberOfEntries); index++) {
            if (anEntry.equals(arr[index])) {
                found = true;
            }
        }
        return found;
    }

    /**
     * This method is used to retrieve the entry at a given position in the
     * list.
     *
     * @param givenPosition The position of the element to get.
     * @return a reference to the indicated entry or null, if either the list is
     * empty, givenPosition smaller 1, or givenPosition bigger getLength()
     * Precondition:The array must exist. Post-condition:The array remains
     * unchanged.
     */
    @Override
    public T getEntry(int givenPosition) {
        T result = null;

        if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
            result = arr[givenPosition - 1];
        }
        return result;
    }

    /**
     * Gets the number of entries currently in the list.
     *
     *
     * @return The number of entries currently in the list. Precondition:The
     * array must exist. Post-condition:The array remains unchanged.
     */
    @Override
    public int size() {
        return numberOfEntries;
    }

    /**
     * This method check if the array is empty
     *
     * @return true if the list is empty, false otherwise. * Post-condition:The
     * array remains unchanged.
     *
     */
    @Override
    public boolean isEmpty() {
        return numberOfEntries == 0;
    }

    /**
     * Removes the element at the specified position in the list.
     *
     * @param givenPosition The position of the element to remove.
     * @return The element that was removed, or null if the position is invalid.
     */
    @Override
    public T remove(int givenPosition) {
        T result = null;

        //the number enter by user must between 1 and the length of the array list
        if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
            result = arr[givenPosition - 1];

            //shift the existing entries if the entry removed is not located at the last
entry
            if (givenPosition < numberOfEntries) {
                removeGap(givenPosition);
            }

            //length should minus 1 after removing a entry
            numberOfEntries--;
        }
        return result;
    }

    /**
     * Removes all occurrences of the specified elements from the list.
     *
     * @param elements The elements to be removed.
     * @return {@code true} if removal is successful, {@code false} if the list
     * is empty or elements are invalid.
     */
    @Override
    public boolean removeAll(T... elements) {
        if (isEmpty() || !isElementsValid(elements)) {
            return false;
        } else {
            for (T element : elements) {
                remove(element);
            }
```

```java
            return true;
        }
    }

    /**
     * Replaces the entry at the specified position with the new entry.
     *
     * @param givenPosition The position of the entry to be replaced.
     * @param newEntry The new entry to replace the existing entry.
     * @return {@code true} if replacement is successful, {@code false} if the
     * list is empty, or position is invalid.
     */
    @Override
    public boolean replace(int givenPosition, T newEntry) {
        boolean isSuccessful = true;

        if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
            //-1 because the givenPosition will only start with 1, does not like
//             index of array list that start with 0.
            arr[givenPosition - 1] = newEntry;
        } else {
            isSuccessful = false;
        }
        return isSuccessful;
    }

    /**
     * Checks if the array is full.
     *
     * @return {@code true} if the array is full, {@code false} otherwise.
     */
    @Override
    public boolean isFull() {
        return false;
    }

    /**
     * Removes the first occurrence of the specified entry from the list.
     *
     * @param anEntry The entry to be removed.
     * @return {@code true} if removal is successful, {@code false} if the entry
     * is not found.
     */
    @Override
    public boolean remove(T anEntry) {
        boolean isSuccessful = false;

        //if the array list is not empty
        if (!isEmpty()) {
            for (int i = 0; i < numberOfEntries; i++) {
                if (arr[i].equals(anEntry)) { //compare the given entry and
//                    every entry in the array list,
                    // if true then go in and remove the given entry
                    removeGap(i + 1);
                    isSuccessful = true;
                    numberOfEntries--;
                }
            }
        }

        return isSuccessful;
    }

    /**
     * Checks if the array is full.
     *
     * @return {@code true} if the array is full, {@code false} otherwise.
     */
    private boolean isArrayFull() {
        return arr.length == numberOfEntries;

    }

    /**
     * Doubles the size of the array.
     */
    private void doubleArray() {
        T[] oldArray = arr;
        arr = (T[]) new Object[2 * oldArray.length];
        System.arraycopy(oldArray, 0, arr, 0,
                numberOfEntries);
    }

    /**
     * Returns a string representation of the list.
     *
     * @return A string representation of the list.
     */
    @Override
    public String toString() {
        String outputStr = "";
        for (int index = 0; index < numberOfEntries; ++index) {
```

```java
            outputStr += arr[index] + "\n";
        }

        return outputStr;
    }

    /**
     * Creates room for a new entry at the specified position.
     *
     * @param newPosition The position at which to create room.
     */
    private void makeRoom(int newPosition) {
        int newIndex = newPosition - 1;
        int lastIndex = numberOfEntries - 1;

        // move each entry to next higher index, starting at end of
        // array and continuing until the entry at newIndex is moved
        for (int index = lastIndex; index >= newIndex; index--) {
            arr[index + 1] = arr[index];
        }
    }

    /**
     * Removes the gap left by removing an entry at the given position.
     *
     * @param givenPosition The position at which to remove the gap.
     */
    private void removeGap(int givenPosition) {
        // move each entry to next lower position starting at entry after the
        // one removed and continuing until end of array
        int removedIndex = givenPosition - 1;
        int lastIndex = numberOfEntries - 1;

        for (int index = removedIndex; index < lastIndex; index++) {
            arr[index] = arr[index + 1];
        }
    }

    public static <T extends Comparable<T>> void insertionSort(
            ListInterface<T> a, Comparator<T> comparator, String val) {
        for (int unsorted = 1; unsorted < a.size(); unsorted++) {
            T firstUnsorted = a.getEntry(unsorted + 1);
            insertInOrder(firstUnsorted, a, unsorted, comparator, val);
        }
    }

    //inserts element at the correct index within thes sorted subarray
    private static <T extends Comparable<T>> int insertInOrder(T element,
            ListInterface<T> a, int end, Comparator<T> comparator, String val) {
        int index = end;
        if ("asc".equals(val)) {
            while ((index > 0) && (comparator.compare(element,
                    a.getEntry(index)) < 0)) {
                a.replace(index + 1, a.getEntry(index));
                //shifting
                index--;
            }
        } else if (val.equals("des")) {
            while ((index > 0) && (comparator.compare(
                    element, a.getEntry(index)) > 0)) {
                a.replace(index + 1, a.getEntry(
                        index)); //shifting
                index--;
            }
        }
        a.replace(index + 1, element);
        return 0;
    }

    /**
     * Checks if all provided elements are valid (non-null).
     *
     * @param newElements The elements to validate.
     * @return {@code true} if all elements are valid, {@code false} if at least
     * one element is null.
     */
    private boolean isElementsValid(T... newElements) {
        boolean valid = true;
        for (int i = 0; i < newElements.length && valid; i++) {
            if (newElements[i] == null) {
                valid = false;
            }
        }
        return valid;
    }

    /**
     * Returns an iterator over the elements in the list.
     *
     * @return An iterator over the elements in the list.
     */
    @Override
    public Iterator<T> getIterator() {
        return new getIterator();
    }


    @Override
    public <T extends Comparable<T>> void bubbleSort() {
        boolean sorted = false;
        for (int pass = 1; pass < this.size() && !sorted; pass++) {
            sorted = true;
            for (int index = 1; index <= this.size() - pass; index++) {
                // swap adjacent elements if first is greater than second
                if (((T) this.getEntry(index)).compareTo((T) (this.getEntry(index + 1)))
> 0) {
                    swap(index, index + 1); // swap adjacent elements
                    sorted = false;  // array not sorted because a swap was performed
                }
            }
        }
    }

    private void swap(int a, int b) {
        T temp = this.getEntry(a);
        this.replace(a, this.getEntry(b));
        this.replace(b, temp);
    }

    public int compare(Tutor tutor1, Tutor tutor2) {
        return tutor1.getSfaculty().compareTo(tutor2.getSfaculty());
    }

    /**
     * Inner class to implement the Iterator interface for the ArrayList.
     *
     * @param <T>
     */
    public class getIterator<T> implements Iterator<T> {

        private int index;

        /**
         * Constructs a new ListIterator.
         */
        public getIterator() {
            index = 0;
        }

        /**
         * Checks if there are more elements to iterate over.
         *
         * @return {@code true} if there are more elements, {@code false}
         * otherwise.
         */
        @Override
        public boolean hasNext() {
            return index < numberOfEntries;
        }

        /**
         * Retrieves the next element in the iteration.
         *
         * @return The next element in the iteration.
         */
        @Override
        public T next() {
            if (!hasNext()) {
                return null;
            }
            T nextEntry = (T) arr[index];
            index++; // advance iterator
            return nextEntry;
        }
    }
}
```