ICSI213 Data Structures

Notice that students are expected to start the lab as soon as the description is available and seek feedback during the lab. Labs are contiguous study of the lecture or used as stepping-stones for the projects. Skipping lab activities would impact the learning significantly.

Lab 08 Traversals of Binary Trees Part II

For this lab, study the tree iterator implementation (**TreeIterator.java** on the next page.) Write a method *main* to test all the methods in this program. You will need to find other source files that are need. Submit all the programs in a zipped folder on Duifene on time.

main	5 points
Total/ 5	5 points

```
import java.util.LinkedList;

/**
 * An iterator over a binary tree.
 * @author Qi Wang
 * @version 1.0
 */
public class TreeIterator<T> implements java.util.Iterator<T> {
  /**
   * The collection traversed by this iterator
   */
  private BaseBinaryTree<T> tree;

  /**
   * An ordered list of nodes traversed by this iterator.
   * Empty list indicates no traversal type currently selected or
   * end of current traversal has been reached
   */
  private LinkedList <TreeNode<T>> list;

  /**
   * Constructs an iterator over a binary tree.
   * @param tree A reference to a binary tree
   */
  public TreeIterator(BaseBinaryTree<T> tree) {
          this.tree = tree;
      // empty queue indicates no traversal type currently
      // selected or end of current traversal has been reached
          this.list = new LinkedList<TreeNode<T>>();
  }

/**
   * Returns true if the iteration has more elements.
   * (In other words, returns true if next() would return an element rather than throwing an exception.)
   * @return A boolean value specifying if there is an element to be returned
   */
  public boolean hasNext() {
     return !this.list.isEmpty();
  }

  /**
   * Returns the next element in the iteration.
   * @return the next element in the iteration
   * @throws NoSuchElementException if the iteration has no more elements
   */
  public T next() throws java.util.NoSuchElementException {
      return this.list.remove().getElement();
```

```
}

/**
 * Removes from the underlying binary tree the last element returned by this iterator (optional operation).
 * It can be called only once per call to next() method. The behavior of an iterator is unspecified if the underlying
 * collection is modified while the iteration is in progress in any way other than by calling this method.
 * @throws UnsupportedOperationException if the remove operation is not supported by this iterator
 * @throws IllegalStateException if the next method has not yet been called, or the remove method has already been
 *                               called after the last call to the next method
 */
public void remove() throws UnsupportedOperationException{
 // Optional means we can implement it or throw an UnsupportedOperationException. This operation is optional because
 // sometimes we just don't want the iterator's content to be modified. An iterator is normally used for executing.
 //The default implementation throws an instance of UnsupportedOperationException and performs no other action.
    throw new UnsupportedOperationException();
}

/**
 * Sets the traversal type to be preorder.
 */
public void setPreorder() {
    this.list.clear();
    this.preorder(this.tree.root);
}

/**
 * Sets the traversal type to be inorder.
 */
public void setInorder() {
    this.list.clear();
    this.inorder(this.tree.root);
}

/**
 * Sets the traversal type to be postorder.
 */
public void setPostorder() {
    this.list.clear();
    this.postorder(this.tree.root);
}

/**
 * Traverses in preorder.
 * @param treeNode A reference to a tree node
 */
private void preorder(TreeNode<T> treeNode) {
  if(treeNode != null){
```

```
        this.list.add(treeNode);
        preorder(treeNode.getLeft());
        preorder(treeNode.getRight());

      }
    }

    /**
     * Traverses in inorder.
     * @param treeNode A reference to a tree node
     */
    private void inorder(TreeNode<T> treeNode) {
            if(treeNode != null){
                  inorder(treeNode.getLeft());
                  this.list.add(treeNode);
                  inorder(treeNode.getRight());
              }
    }

    /**
     * Traverses in postorder.
     * @param treeNode A reference to a tree node
     */
    private void postorder(TreeNode<T> treeNode) {
            if(treeNode != null){
                  postorder(treeNode.getLeft());
                  postorder(treeNode.getRight());
                  this.list.add(treeNode);
              }
    }
}
```