```
/*
 * LinkedTree.java
 *
 * Computer Science S-111
 *
 * Modifications and additions by:
 *     name:
 *     email:
 */

import java.util.*;

/**
 * LinkedTree - a class that represents a binary tree containing data
 * items with integer keys.  If the nodes are inserted using the
 * insert method, the result will be a binary search tree.
 */
public class LinkedTree {
    // An inner class for the nodes in the tree
    private class Node {
        private int key;         // the key field
        private Object data;     // the rest of the data item
        private Node left;       // reference to the left child/subtree
        private Node right;      // reference to the right child/subtree
        private Node parent;     // reference to the parent

        private Node(int key, Object data, Node left, Node right, Node parent){
            this.key = key;
            this.data = data;
            this.left = left;
            this.right = right;
            this.parent = parent;
        }

        private Node(int key, Object data) {
            this(key, data, null, null, null);
        }
    }

    // the root of the tree as a whole
    private Node root;

    public LinkedTree() {
        root = null;
    }

    /**
     * Prints the keys of the tree in the order given by a preorder traversal.
     * Invokes the recursive preorderPrintTree method to do the work.
     */
    public void preorderPrint() {
        if (root != null)
            preorderPrintTree(root);
    }

    /*
     * Recursively performs a preorder traversal of the tree/subtree
     * whose root is specified, printing the keys of the visited nodes.
     * Note that the parameter is *not* necessarily the root of the
     * entire tree.
     */
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null)
```

```
            preorderPrintTree(root.left);
        if (root.right != null)
            preorderPrintTree(root.right);
    }

    /**
     * Prints the keys of the tree in the order given by a postorder traversal.
     * Invokes the recursive postorderPrintTree method to do the work.
     */
    public void postorderPrint() {
        if (root != null)
            postorderPrintTree(root);
    }

    /*
     * Recursively performs a postorder traversal of the tree/subtree
     * whose root is specified, printing the keys of the visited nodes.
     * Note that the parameter is *not* necessarily the root of the
     * entire tree.
     */
    private static void postorderPrintTree(Node root) {
        if (root.left != null)
            postorderPrintTree(root.left);
        if (root.right != null)
            postorderPrintTree(root.right);
        System.out.print(root.key + " ");
    }

    /**
     * Prints the keys of the tree in the order given by an inorder traversal.
     * Invokes the recursive inorderPrintTree method to do the work.
     */
    public void inorderPrint() {
        if (root != null)
            inorderPrintTree(root);
    }

    /*
     * Recursively performs an inorder traversal of the tree/subtree
     * whose root is specified, printing the keys of the visited nodes.
     * Note that the parameter is *not* necessarily the root of the
     * entire tree.
     */
    private static void inorderPrintTree(Node root) {
        if (root.left != null)
            inorderPrintTree(root.left);
        System.out.print(root.key + " ");
        if (root.right != null)
            inorderPrintTree(root.right);
    }

    /*
     * Inner class for temporarily associating a node's depth
     * with the node, so that levelOrderPrint can print the levels
     * of the tree on separate lines.
     */
    private class NodePlusDepth {
        private Node node;
        private int depth;

        private NodePlusDepth(Node node, int depth) {
            this.node = node;
            this.depth = depth;
        }
    }
```

```
    /**
     * Prints the keys of the tree in the order given by a
     * level-order traversal.
     */
    public void levelOrderPrint() {
        LLQueue<NodePlusDepth> q = new LLQueue<NodePlusDepth>();

        // Insert the root into the queue if the root is not null.
        if (root != null)
            q.insert(new NodePlusDepth(root, 0));

        // We continue until the queue is empty.  At each step,
        // we remove an element from the queue, print its value,
        // and insert its children (if any) into the queue.
        // We also keep track of the current level, and add a newline
        // whenever we advance to a new level.
        int level = 0;
        while (!q.isEmpty()) {
            NodePlusDepth item = q.remove();

            if (item.depth > level) {
                System.out.println();
                level++;
            }
            System.out.print(item.node.key + " ");

            if (item.node.left != null)
                q.insert(new NodePlusDepth(item.node.left, item.depth + 1));
            if (item.node.right != null)
                q.insert(new NodePlusDepth(item.node.right, item.depth + 1));
        }
    }

    /**
     * Searches for the specified key in the tree.
     * Invokes the recursive searchTree method to perform the actual search.
     */
    public Object search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }

    /*
     * Recursively searches for the specified key in the tree/subtree
     * whose root is specified. Note that the parameter is *not*
     * necessarily the root of the entire tree.
     */
    private static Node searchTree(Node root, int key) {
        if (root == null)
            return null;
        else if (key == root.key)
            return root;
        else if (key < root.key)
            return searchTree(root.left, key);
        else
            return searchTree(root.right, key);
    }

    /**
     * Inserts the specified (key, data) pair in the tree so that the
     * tree remains a binary search tree.
     */
    public void insert(int key, Object data) {
        // Find the parent of the new node.
```

```
        Node parent = null;
        Node trav = root;
        while (trav != null) {
            parent = trav;
            if (key < trav.key)
                trav = trav.left;
            else
                trav = trav.right;
        }

        // Insert the new node.
        Node newNode = new Node(key, data);
        if (parent == null)    // the tree was empty
            root = newNode;
        else if (key < parent.key)
            parent.left = newNode;
        else
            parent.right = newNode;
    }

    /**
     * Deletes the node containing the (key, data) pair with the
     * specified key from the tree and return the associated data item.
     */
    public Object delete(int key) {
        // Find the node to be deleted and its parent.
        Node parent = null;
        Node trav = root;
        while (trav != null && trav.key != key) {
            parent = trav;
            if (key < trav.key)
                trav = trav.left;
            else
                trav = trav.right;
        }

        // Delete the node (if any) and return the removed data item.
        if (trav == null)   // no such key
            return null;
        else {
            Object removedData = trav.data;
            deleteNode(trav, parent);
            return removedData;
        }
    }

    /**
     * Deletes the node specified by the parameter toDelete.  parent
     * specifies the parent of the node to be deleted.
     */
    private void deleteNode(Node toDelete, Node parent) {
        if (toDelete.left != null && toDelete.right != null) {
            // Case 3: toDelete has two children.
            // Find a replacement for the item we're deleting -- as well as
            // the replacement's parent.
            // We use the smallest item in toDelete's right subtree as
            // the replacement.
            Node replaceParent = toDelete;
            Node replace = toDelete.right;
            while (replace.left != null) {
                replaceParent = replace;
                replace = replace.left;
            }

            // Replace toDelete's key and data with those of the
```

```
            // replacement item.
            toDelete.key = replace.key;
            toDelete.data = replace.data;

            // Recursively delete the replacement item's old node.
            // It has at most one child, so we don't have to
            // worry about infinite recursion.
            deleteNode(replace, replaceParent);
        } else {
            // Cases 1 and 2: toDelete has 0 or 1 child
            Node toDeleteChild;
            if (toDelete.left != null)
                toDeleteChild = toDelete.left;
            else
                toDeleteChild = toDelete.right;  // null if it has no children

            if (toDelete == root)
                root = toDeleteChild;
            else if (toDelete.key < parent.key)
                parent.left = toDeleteChild;
            else
                parent.right = toDeleteChild;
        }
    }

    /**
     * Determines the depth of the node with the specified key,
     * returning -1 if there is no such node.
     */
    public int depth(int key) {
        /*** implement this method for PS 4 ***/

        return -1;
    }

    /**
     * Determines if this tree is isomorphic to the other tree,
     * returning true if they are isomorphic and false if they are not.
     * Calls the private helper method isomorphic() to do the work.
     *
     * You should ***NOT*** change this method. Instead, you should
     * implement the private helper method found below.
     */
    public boolean isomorphicTo(LinkedTree other) {
        if (other == null)
            throw new IllegalArgumentException("parameter must be non-null");

        return isomorphic(this.root, other.root);
    }

    /**
     * Determines if the trees with the specified root nodes are
     * isomorphic, returning true if they are and false if they are not.
     */
    private static boolean isomorphic(Node root1, Node root2) {
        /*** implement this method for PS 4 ***/

        return false;
    }

    /** Returns a preorder iterator for this tree. */
    public LinkedTreeIterator preorderIterator() {
        return new PreorderIterator();
    }
```

```
    /** Returns an inorder iterator for this tree. */
    public LinkedTreeIterator inorderIterator() {
        /*** implement this method for PS 4 ***/

        return null;
    }

    /*** inner class for a preorder iterator ***/
    private class PreorderIterator implements LinkedTreeIterator {
        private Node nextNode;

        private PreorderIterator() {
            // The traversal starts with the root node.
            nextNode = root;
        }

        public boolean hasNext() {
            return (nextNode != null);
        }

        public int next() {
            if (nextNode == null)
                throw new NoSuchElementException();

            // Store a copy of the key to be returned.
            int key = nextNode.key;

            // Advance nextNode.
            if (nextNode.left != null)
                nextNode = nextNode.left;
            else if (nextNode.right != null)
                nextNode = nextNode.right;
            else {
                // We've just visited a leaf node.
                // Go back up the tree until we find a node
                // with a right child that we haven't seen yet.
                Node parent = nextNode.parent;
                Node child = nextNode;
                while (parent != null &&
                       (parent.right == child || parent.right == null)) {
                    child = parent;
                    parent = parent.parent;
                }

                if (parent == null)
                    nextNode = null;  // the traversal is complete
                else
                    nextNode = parent.right;
            }

            return key;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        LinkedTree tree = new LinkedTree();
        tree.insert(7, "root node");
        tree.insert(9, "7's right child");
        tree.insert(5, "7's left child");
        tree.insert(2, "5's left child");
        tree.insert(8, "9's left child");
        tree.insert(6, "5's right child");
        tree.insert(4, "2's right child");
```

```
        System.out.print("\n preorder: ");
        tree.preorderPrint();
        System.out.println();

        System.out.print("postorder: ");
        tree.postorderPrint();
        System.out.println();

        System.out.print("  inorder: ");
        tree.inorderPrint();
        System.out.println();

        System.out.print("\nkey to search for: ");
        int key = in.nextInt();
        in.nextLine();
        Object data = tree.search(key);
        if (data != null)
            System.out.println(key + " = " + data);
        else
            System.out.println("no such key in tree");

        System.out.print("\nkey to delete: ");
        key = in.nextInt();
        in.nextLine();
        data = tree.delete(key);
        if (data != null)
            System.out.println("removed " + data);
        else
            System.out.println("no such key in tree");

        System.out.print("\n preorder: ");
        tree.preorderPrint();
        System.out.println();

        System.out.print("postorder: ");
        tree.postorderPrint();
        System.out.println();

        System.out.print("  inorder: ");
        tree.inorderPrint();
        System.out.println();
    }
}
```