# Lab107

Steven Glasford

3-12-2019

## 1 Client.java

```java
/**
 * A main class containing the realest part of the program.
 * @author Steven Glasford
 * @version 3-5-2019
 */
public class Client {

    /**
     * @param args No command line arguments.
     */
    public static void main(String[] args) {
        //Manually create an instance of a binary expression tree
        //for (((5+2)*(2-1)/((2+9))+((7-2)-1))*8)
        LinkedBinaryTree treeShit = new LinkedBinaryTree();



        //build the tree, use some Positions to make it easier for me to
        //build the tree in a fashion similar to the following:
        /*
        where the letters repesent the place where the Position variable
        are supposed to represent, and the ? character represents the
        division character, so as to avoid looking like another arm of
        the tree.

                   root *
                       / \
                     A+   8
                     / \
                    /   \
                   /     \
                  /       \
                B?        C-
                / \       / \
               /   \    F-   1
             D*    E+   / \
             / \   / \ 7   2
            /   \ 2   9
          G+    H-
          / \   / \
         5   2 2   1

        */
        Position root;
        Position A;
        Position B;
        Position C;
        Position D;
```

```java
    Position E;
    Position F;
    Position G;
    Position H;

    root = treeShit.addRoot("*");
    treeShit.addRight(root, "8");
    A = treeShit.addLeft(root, "+");
    B = treeShit.addLeft(A, "/");
    C = treeShit.addRight(A, "-");
    treeShit.addRight(C,"1");
    D = treeShit.addLeft(B,"*");
    E = treeShit.addRight(B, "+");
    treeShit.addLeft(E,"2");
    treeShit.addRight(E, "9");
    F = treeShit.addLeft(C, "-");
    treeShit.addLeft(F, "7");
    treeShit.addRight(F, "2");
    G = treeShit.addLeft(D, "+");
    treeShit.addLeft(G, "5");
    treeShit.addRight(G, "2");
    H = treeShit.addRight(D, "-");
    treeShit.addLeft(H, "2");
    treeShit.addRight(H, "1");

    //take the root of the tree and print it out in pre order fashion
    System.out.println("The preorder traversal of the tree.");
    System.out.println(preorder(treeShit, root) + "\n");

    //take the root of the tree and print it out in in order fashion
    System.out.println("The inorder traversal of the tree.");
    System.out.println(inorder(treeShit,root) + "\n");

    //take the root of the tree and print it out in post order fashion
    System.out.println("The postorder traversal of the tree.");
    System.out.println(postorder(treeShit,root) + "\n");

    //take the root of the tree and print it out in breadth first fashion
    System.out.println("The breadth-first traversal of the tree.");
    System.out.println(breadth(treeShit, root) + "\n");

    //print the tree using parantheses
    System.out.println("The parenthesized version of the tree.");
    System.out.println(parenthesize(treeShit, root) + "\n");
}

/**
 * Print the tree in preorder notation.
 * @param <E>   A generic placement name.
 * @param T     The tree in question.
 * @param p     The root of the tree you are investigating.
 * @return a string containing the tree in preorder notation
 */
public static <E> String preorder(LinkedBinaryTree<E> T, Position<E> p){
    String middle = "";
    String left = "";
    String right = "";

    //save the middle element for later printing
    if (p != null)
        middle = (String) p.getElement();
```

```java
        //get the left child and save it
        if (T.left(p) != null)
            left = (String) preorder(T, T.left(p));
        else
            return middle;

        //get the right child
        if (T.right(p) != null)
            right = (String) preorder(T,T.right(p));
        else
            return middle;

        return middle + " " + left + " " + right;
    }

    /**
     * Return a string of a tree in inorder notation.
     * @param <E>   A generic placeholder for the data contained in the tree.
     * @param T     The tree in question.
     * @param p     The root of the tree.
     * @return  a string of inorder traversal of a tree
     */
    public static <E> String inorder(LinkedBinaryTree<E> T, Position<E> p){
        String middle = "";
        String left = "";
        String right = "";

        //save the middle element for later printing
        if (p != null)
            middle = (String) p.getElement();

        //get the left child and save it
        if (T.left(p) != null)
            left = (String) inorder(T, T.left(p));
        else
            return middle;

        //get the right child
        if (T.right(p) != null)
            right = (String) inorder(T,T.right(p));
        else
            return middle;

        return left + " "  + middle + " " + right;
    }

    /**
     * Given the root of a tree, print out the postorder transversal.
     * @param <E>
     * @param p The root of a tree.
     * @return  The output of the string in postorder notation.
     */
    public static <E> String postorder(LinkedBinaryTree<E> T, Position<E> p){
        String middle = "";
        String left = "";
        String right = "";

        //save the middle element for later printing
        if (p != null)
            middle = (String) p.getElement();
```

```java
    //get the left child and save it
    if (T.left(p) != null)
        left = (String) postorder(T, T.left(p));
    else
        return middle;

    //get the right child
    if (T.right(p) != null)
        right = (String) postorder(T,T.right(p));
    else
        return middle;

    return left + " " + right + " " + middle;
}


/**
 * Print out the tree in breadth-first notation, using breadth-first
 * traversal.
 * @param <E>   A generic placement.
 * @param T     The binary tree in question.
 * @param p     The root of the tree.
 * @return      A string containing all of the information from the tree
 *              in breadth-first traversal notation; will be null if
 *              it is empty.
 */
public static <E> String breadth(LinkedBinaryTree<E> T, Position<E> p){
    //determine if the root is real, return null if not
    if (p == null)
        return null;

    //initialize the queue.
    LinkedQueue queue = new LinkedQueue();
    //initialize a string that will eventually be returned.
    String bread = "";
    //enqueue the root of the tree.
    queue.enqueue(p);

    //go through the breadth-first traversal.
    while (!queue.isEmpty()) {
        //remove the position from the queue.
        p = (Position) queue.dequeue();

        //add the newly removed position to the string.
        bread += p.getElement() + " ";

        //add the left child to the queue.
        if (T.left(p) != null)
            queue.enqueue(T.left(p));
        //add the right child to the queue.
        if (T.right(p) != null)
            queue.enqueue(T.right(p));
    }
    //return the final string
    return bread;
}

/**
 * Prints parenthesized representation of subtree of T rooted at p.
 * @param <E>   A generic placement name.
```

```java
 * @param T      The tree in question.
 * @param p      The root of the tree.
 * @return       The root as a string.
 */
public static <E> String parenthesize(LinkedBinaryTree<E> T,
        Position<E> p){
    //the middle of the tree (the root)
    String middle = "";
    //the left child
    String left = "";
    //the right child
    String right = "";
    //get the middle part of the equation thing
    if (p != null)
        middle = (String) p.getElement();

    //get the left part of the equation and recurse
    if (T.left(p) != null)
        left = parenthesize(T, T.left(p));
    //return the middle element prematurely if it has no children
    else
        return middle;

    //get the right part of the equation and recurse
    if (T.right(p) != null)
        right = parenthesize(T, T.right(p));
    //return the middle element prematurely if it has no children
    else
        return middle;

    //print out the tree with the parentheses
    return ("(" + left + middle + right + ")");
}

/**
 * Prints preorder representation of subtree of T rooted at p
 * having depth d.
 * @param <E>    A generic representation of the whatever you want
 *               printed.
 * @param T      The root of the tree you want to investigate.
 * @param p      The current position of you are in
 * @param d      The current depth of the tree.
 */
public static <E> void printPreorderIndent(Tree<E> T, Position<E> p,
        int d){
    //indent based on d.
    System.out.println(spaces(2*d) + p.getElement());
    for (Position<E> c : T.children(p))
        //child depth is d+1
        printPreorderIndent(T,c,d+1);
}

/**
 * A function to add a certain number of spaces and return the number of
 * spaces as a string.
 * @param num    The number of spaces you want.
 * @return       The number of spaces you want.
 */
public static String spaces(int num){
    String temp = "";
    for (int i = 0; i < num; num++){
```

```
            temp += " ";
        }
        return temp;
    }
}
```

## 2 AbstractBinaryTree.java

```
import java.util.ArrayList;
import java.util.List;

/**
 * An abstract base class providing some functionality of the BinaryTree
 * Interface.
 * @author      Steven Glasford, Goodrick, Tamassia, Goldwasser
 *              Data Structures & Algorithms 6th Edition
 * @version     3-5-2019
 * @param   <E> A generic parameter
 */
public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
    implements BinaryTree<E> {

    /**
     * Returns the Position of p s sibling (or null if no sibling exists).
     * @param p The position of the other sibling for a node.
     * @return  The position of the sibling.
     */
    @Override
    public Position<E> sibling(Position<E> p){
        Position<E> parent = parent(p);
        //p must be the root
        if(parent == null) return null;
        //p is a left child
        if (p == left(parent))
            //(right child might be null)
            return right(parent);
        //p is a right child
        else
            //(left child might be null)
            return left(parent);
    }

    /**
     * Returns the number of children of Position p.
     * @param p The node you are testing for.
     * @return  The number of children for a particular node, in int.
     */
    @Override
    public int numChildren(Position<E> p) {
        int count = 0;
        if (left(p) != null)
            count++;
        if (right(p) != null)
            count++;
        return count;
    }

    /**
     * Returns an iterable collection of the Positions representing p children.
     * @param p The position you want to mess around with.
     * @return  The iterable collection of the Positions representing
     *          p children.
     */
    @Override
    public Iterable<Position<E>> children(Position<E> p) {
        //max capacity of 2
```

```java
        List<Position<E>> snapshot = new ArrayList<>(2);
        if (left(p) != null)
            snapshot.add(left(p));
        if (right(p) != null)
            snapshot.add(right(p));
        return snapshot;//by progressive
    }

    /**
     * Adds positions of the subtree rooted at Position p to the
     * given snapshot.
     * @param p         The position to begin with.
     * @param snapshot  The lists of positions in which the thing is located.
     */
    private void inorderSubtree(Position<E> p, List<Position<E>> snapshot){
        if (left(p) != null)
            inorderSubtree(left(p),snapshot);
        snapshot.add(p);
        if (right(p) != null)
            inorderSubtree(right(p),snapshot);
    }

    /**
     * Returns an iterable collection of positions of the tree, reported
     * in inorder.
     * @return an iterable collection of positions of the tree, reported
     * in inorder.
     */
    public Iterable<Position<E>> inorder() {
        List<Position<E>> snapshot = new ArrayList<>();
        if (!isEmpty())
            //fill the snapshot recursively
            inorderSubtree(root(),snapshot);
        return snapshot;
    }

    /**
     * Overrides positions to make inorder the default for binary trees.
     * @return an inorder operator.
     */
    @Override
    public Iterable<Position<E>> positions(){
        return inorder();
    }
}
```

# 3 AbstractTree.java

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * An abstract base class providing some functionality of the Tree interface.
 * @author      Steven Glasford, Goodrick, Tamassia, Goldwasser
 *              Data Structures & Algorithms 6th Edition.
 * @param   <E> A generic parameter
 */
public abstract class AbstractTree<E> implements Tree<E> {
    /**
     * Determine if the node that is being tested is an internal
     * component.
     * @param p The node you want to determine if it is an internal
     *          component.
     * @return  true or false depending on whether or not the test node
     *          is an internal component.
     */
    @Override
    public boolean isInternal(Position<E> p) {return numChildren(p) > 0;}

    /**
     * Determine if a node is an external component, if it is a leaf.
     * @param p The node you want to test.
     * @return  True or false dependent on the conditions.
     */
    @Override
    public boolean isExternal(Position<E> p) {return numChildren(p) == 0;}

    /**
     * Determine if a node is a Root (like my car insurance).
     * @param p The node you want to test.
     * @return  True or false, depending on if the node is a root.
     */
    @Override
    public boolean isRoot(Position<E> p) {return p == root();}

    /**
     * Determine if a tree is empty.
     * @return  True or false, depending on if the tree is empty.
     */
    @Override
    public boolean isEmpty() {return size() == 0;}

    /**
     * Returns the number of levels separating Position p from the root.
     * @param p The node you want to test at.
     * @return  The number of levels separating Position p from the root.
     */
    public int depth(Position<E> p) {
        if (isRoot(p))
            return 0;
        else
            return 1 + depth(parent(p));
    }

    /**
```

```java
 * Returns the height of the tree.
 * Works, but has quadratic worst-case time.
 * @return the height of the tree.
 */
private int heightBad(){
    int h = 0;
    for (Position<E> p : positions())
        //only consider leaf positions
        if (isExternal(p))
            h = Math.max(h, depth(p));
    return h;
}

/**
 * Returns the height of the subtree rooted at Position p.
 * @param p The node you are testing from.
 * @return  the height of the tree
 */
public int height(Position<E> p){
    //base case if p is external
    int h = 0;
    for (Position<E> c : children(p))
        h = Math.max(h, 1 + height(c));
    return h;
}

/**
 * This class adapts the iteration produced by positions() to return
 * elements.
 */
private class ElementIterator implements Iterator<E> {
    Iterator<Position<E>> posIterator = positions().iterator();
    @Override
    public boolean hasNext() {return posIterator.hasNext();}
    //return element
    @Override
    public E next() {return posIterator.next().getElement();}
    @Override
    public void remove() {posIterator.remove();}
}

/**
 * Returns an iterator of the elements stored in the tree.
 * @return an iterator of the elements stored in the tree.
 */
@Override
public Iterator<E> iterator() {return new ElementIterator();}

/**
 * defining the preorder as the default traversal algorithm for the
 * public positions method of an abstract tree.
 * @return
 */
@Override
public Iterable<Position<E>> positions() {return preorder();}

/**
 * Adds positions of the subtree rooted at Position p to the given
 * snapshot.
 * @param p          A position to be investigated
 * @param snapshot   by Progressive.
```

```java
 */
private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    //for preorder, we add position p before exploring subtrees
    snapshot.add(p);
    for(Position<E> c : children(p))
        preorderSubtree(c,snapshot);
}

/**
 * Returns an iterable collection of positions of the tree,
 * reported in preorder.
 * @return  an iterable collection of positions of the tree,
 *          reported in preorder.
 */
public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<>();
    if (isEmpty())
        //fill the snapshot recursively
        preorderSubtree(root(), snapshot);
    return snapshot;
}

/**
 * Adds positions of the subtree rooted at Position p to the
 * given snapshot
 * @param p         The position of the subtree
 * @param snapshot  by progressive
 */
private void postorderSubtree(Position<E> p, List<Position<E>> snapshot){
    for (Position<E> c : children(p))
        postorderSubtree(c,snapshot);
    //for postorder, we add position p after exploring subtrees
    snapshot.add(p);
}

/**
 * Returns an iterable collection of positions of the tree,
 * reported in postorder.
 * @return  an iterable collection of positions of the tree,
 *          reported in postorder.
 */
public Iterable<Position<E>> postorder(){
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty())
        //fill the snapshot recursively
        postorderSubtree(root(), snapshot);
    return snapshot;
}

/**
 * Returns an iterable collection of positions of the tree in
 * breadth-first order.
 * @return  an iterable collection of positions of the tree in
 *          breadth-first order.
 */
public Iterable<Position<E>> breadthfirst(){
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty()) {
        Queue<Position<E>> fringe = new LinkedQueue<>();
        //start with the root
        fringe.enqueue(root());
```

```
            while (!fringe.isEmpty()) {
                //remove from the front of the queue
                Position<E> p = fringe.dequeue();
                //report this position
                snapshot.add(p);
                for (Position<E> c : children(p))
                    //add children to the back of the queue
                    fringe.enqueue(c);
            }
        }
        return snapshot;
    }
}
```

# 4 LinkedBinaryTree.java

```java
import java.util.Iterator;

/**
 * Concrete implementation of a binary tree using a node-based, linked
 * structure.
 * @author      Steven Glasford, Goodrick, Tamassia, Goldwasser
 *              Data Structures & Algorithms 6th Edition.
 * @version     3-5-2019
 * @param <E>   A generic parameter
 */
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

    @Override
    public Iterator<E> iterator() {
        throw new UnsupportedOperationException("Not supported yet.");
        //To change body of generated methods, choose Tools | Templates.
    }

    @Override
    public Iterable<Position<E>> positions() {
        throw new UnsupportedOperationException("Not supported yet.");
        //To change body of generated methods, choose Tools | Templates.
    }
    //------------------nested Node class------------------//
    /**
     * A commonality of all nodes in the tree, smells like potatoes.
     * @param <E>   A generic input parameter.
     */
    protected static class Node<E> implements Position<E> {
        //an element stored at this node.
        private E element;
        //a reference to the parent node (if any).
        private Node<E> parent;
        //a reference to the left child (if any).
        private Node<E> left;
        //a reference to the right child (if any).
        private Node<E> right;

        /**
         * Constructs a node with the given element and neighbors.
         * @param e             The element you want to add.
         * @param above         The parent of the new child.
         * @param leftChild     The left child of the new parent.
         * @param rightChild    The right child of the new parent.
         */
        public Node(E e, Node<E> above, Node<E> leftChild,
                Node<E> rightChild) {
            element = e;
            parent = above;
            left = leftChild;
            right = rightChild;
        }

        //****************** accessor methods *************************//

        /**
         * Gets element in the node out of protection.
         * @return  The data within the tree.
```

```java
     */
    @Override
    public E getElement()       {return element;}

    /**
     * Gets the parent out of protections, assuming they are due on
     * alimony payments.
     * @return  The parent to the child, null if the root.
     */
    public Node<E> getParent()  {return parent;}

    /**
     * Get the left child out of protection.
     * @return  Return the node of the left child.
     */
    public Node<E> getLeft()    {return left;}

    /**
     * Get the right child out of protection.
     * @return  Return the node of the left child.
     */
    public Node<E> getRight()   {return right;}


    //***************** update methods *******************//

    /**
     * Set the element from outside of protection.
     * @param e The element you want to add to the node.
     */
    public void setElement(E e) {element = e;}

    /**
     * Set the nodes parent, like adoption.
     * @param parentNode    The parent node.
     */
    public void setParent(Node<E> parentNode) { parent = parentNode;}

    /**
     * Set the left Child of the node.
     * @param leftChild The left child you want to set.
     */
    public void setLeft(Node<E> leftChild) { left = leftChild; }

    /**
     * Set the right Child of the node.
     * @param rightChild    The right child that you want to set.
     */
    public void setRight(Node<E> rightChild) { right = rightChild; }
} ////////////////////// end of nested Node class ////////////////////

/**
 * Factory function to create a new node storing element e.
 * @param e         The element you want to create a node for.
 * @param parent    The parent of the new node you just made.
 * @param left      The first, left, child of the new node.
 * @param right     The second, right, child of the new node.
 * @return          The new node.
 */
protected Node<E> createNode(E e, Node<E> parent, Node<E> left,
        Node<E> right){
```

```
        return new Node<>(e,parent,left,right);
    }

    //////////////////LinkedBinaryTree instance variables//////////////////

    //root of the tree, protected like dirt.
    protected Node<E> root = null;
    //number of nodes in the tree
    private int size = 0;

    /**
     * Constructor, constructs an empty binary tree
     */
    public LinkedBinaryTree(){}

    //////////////////////nonpublic utility///////////////////////////
    /**
     * Validates the position and returns it as a node.
     * @param p The position you want to create.
     * @return  The node from the position.
     * @throws  IllegalArgumentException If the position doesnt exist.
     */
    protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
        if (!(p instanceof Node))
            throw new IllegalArgumentException("Not valid position type");
        //safe cast
        Node<E> node = (Node<E>) p;
        //Our convention for defunct node
        if (node.getParent() == node)
            throw new IllegalArgumentException("p is no longer in the tree");
        return node;
    }

    //////Accesor methods (not already implemented in AbstractBinaryTree)//////
    /**
     * Returns the number of nodes in the tree.
     * @return  An integer of the size of the tree.
     */
    @Override
    public int size() {
        return size;
    }

    /**
     * Returns the root Position of the tree (or null if tree is empty).
     * @return  The position of the root.
     */
    @Override
    public Position<E> root(){
        return root;
    }

    /**
     * Returns the Positions of p s parent (or null if p is root)
     * @param p The position you are testing from.
     * @return  The position of the parent.
     * @throws  IllegalArgumentException if the position doesnt exist.
     */
    @Override
    public Position<E> parent(Position<E> p) throws IllegalArgumentException {
        Node<E> node = validate(p);
```

```java
        return node.getParent();
    }

    /**
     * Returns the Position of p s left child (or null if no child exists).
     * @param p The node you are trying to find the child for.
     * @return  The position of the left nodes left child.
     * @throws IllegalArgumentException if the position doesnt exist.
     */
    @Override
    public Position<E> left(Position<E> p) throws IllegalArgumentException {
        Node<E> node = validate(p);
        return node.getLeft();
    }

    /**
     * Returns the Position of p s right child (or null if no child exists)
     * @param p The position you are trying to find the right child for.
     * @return  The position of the right child.
     * @throws IllegalArgumentException if the input position doesnt exist.
     */
    @Override
    public Position<E> right(Position<E> p) throws IllegalArgumentException {
        Node<E> node = validate(p);
        return node.getRight();
    }

    //////////////update methods supported by this class//////////////////
    /**
     * Places element e at the root of an empty tree and returns
     * its new position.
     * @param e The element you want to be the root of the tree.
     * @return  The new position of the root of the tree.
     * @throws IllegalStateException    if the tree is not empty.
     */
    public Position<E> addRoot(E e) throws IllegalStateException {
        if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
        root = createNode(e, null, null, null);
        size = 1;
        return root;
    }

    /**
     * Creates a new left child of Position p storing element e;
     * returns its Position.
     * @param p The parent of the new child.
     * @param e The element you want to add to the new child.
     * @return  The position of the newly created left child.
     * @throws  IllegalArgumentException if the input position already
     *                                     has a child.
     */
    public Position<E> addLeft(Position<E> p, E e)
            throws IllegalArgumentException{
        Node<E> parent = validate(p);
        if (parent.getLeft() != null)
            throw new IllegalArgumentException("p already has a right child");
        Node<E> child = createNode(e,parent,null,null);
        parent.setLeft(child);
        size++;
        return child;
    }
```

```java
/**
 * Creates a new right child of Position p storing element e;
 * returns its Position.
 * @param p The parent of the new child.
 * @param e The element you want to add to the new child.
 * @return  The position of the new right child.
 * @throws IllegalArgumentException if the position already has a
 *                                  right child.
 */
public Position<E> addRight(Position<E> p, E e)
        throws IllegalArgumentException{
    Node<E> parent = validate(p);
    if (parent.getRight() != null)
        throw new IllegalArgumentException("p already has a right child");
    Node<E> child = createNode(e, parent, null, null);
    parent.setRight(child);
    size++;
    return child;
}

/**
 * Replaces the element at Position p with e and returns the
 * replaced element
 * @param p The position in which you are changing the element data for.
 * @param e The element you want to set the data for.
 * @return  The element you set for the particular node.
 * @throws IllegalArgumentException if the position and node do not exist.
 */
public E set(Position<E> p, E e) throws IllegalArgumentException {
    Node<E> node = validate(p);
    E temp = node.getElement();
    node.setElement(e);
    return temp;
}

/**
 * Attaches trees t1 and t2 as left and right subtrees of external p.
 * @param p      The new parent of the two trees.
 * @param t1     The left part of the tree.
 * @param t2     The right part of the tree.
 * @throws IllegalArgumentException if one node doesnt exist.
 */
public void attach(Position<E> p, LinkedBinaryTree<E> t1,
        LinkedBinaryTree<E> t2) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (isInternal(p)) throw new IllegalArgumentException("p must"
            + " be a leaf");
    size += t1.size() + t2.size();
    //attach t1 as left subtree of node
    if (!t1.isEmpty()){
        t1.root.setParent(node);
        node.setLeft(t1.root);
        t1.root = null;
        t1.size = 0;
    }

    //attach t2 as right subtree of node
    if (!t2.isEmpty()){
        t2.root.setParent(node);
        node.setRight(t2.root);
```

```java
            t2.root = null;
            t2.size = 0;
        }
    }

    /**
     * Removes the node at Position p and replaces it with its child,
     * if any.
     * @param p The position of the node you want to remove.
     * @return  The element that once was stored in the element you
     *          just removed.
     * @throws IllegalArgumentException if the element doesn't exist.
     */
    public E remove(Position<E> p) throws IllegalArgumentException {
        Node<E> node = validate(p);
        if (numChildren(p) == 2)
            throw new IllegalArgumentException("p has two children");
        Node<E> child = (node.getLeft() != null ? node.getLeft():
                node.getRight());
        if (child != null)
            //child s grandparent becomes its parent
            child.setParent(node.getParent());
        if (node == root)
            //child becomes root
            root = child;
        else{
            Node<E> parent = node.getParent();
            if (node == parent.getLeft())
                parent.setLeft(child);
            else
                parent.setRight(child);
        }
        size--;
        E temp = node.getElement();
        //help garbage collection
        node.setElement(null);
        node.setLeft(null);
        node.setRight(null);
        //our convention for defunct node
        node.setParent(node);
        return temp;
    }
    ////////////////////end of LinkedBinaryTree class//////////////////////
}
```

# 5 BinaryTree.java

```java
/**
 * An interface for a binary tree, in which each node has at most two children.
 * @author      Steven Glasford, Goodrick, Tamassia, Goldwasser
 *              Data Structures & Algorithms 6th Edition
 * @version     3-5-2019
 * @param   <E> A generic parameter.
 */
public interface BinaryTree<E> extends Tree<E> {
    /**
     * Returns the Position of p s left child (or null if no child exists).
     * @param p The position you want to find the left child for.
     * @return  The position of the left child from an input position.
     * @throws IllegalArgumentException If the node doesnt exist.
     */
    Position<E> left(Position<E> p) throws IllegalArgumentException;

    /**
     * Returns the Position of p s right child (or null if no child exists).
     * @param p The position of the parent node.
     * @return  The position of the right child.
     * @throws IllegalArgumentException If the input position doesnt exist.
     */
    Position<E> right(Position<E> p) throws IllegalArgumentException;

    /**
     * Returns the Position of p s sibling (or null if no sibling exists).
     * @param p The node you want to find its sibling for.
     * @return  The position of the other sibling.
     * @throws IllegalArgumentException If the input position doesnt exist.
     */
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```

## 6 Tree.java

```java
import java.util.Iterator;

/**
 * An interface for a tree where nodes can have an arbitrary number of children
 * @author       Steven Glasford, Goodrick, Tamassia, Goldwasser
 *               Data Structures & Algorithms 6th Edition
 * @version      3-5-2019
 * @param <E>    A generic parameter
 */

public interface Tree<E> extends Iterable<E> {
    /**
     * Make the root of the tree.
     * @return The root of the tree.
     */
    Position<E> root();

    /**
     * Make a parent of in the tree.
     * @param p The leaf you want to make into a parent.
     * @return A new parent node
     * @throws IllegalArgumentException If the node doesnt exist.
     */
    Position<E> parent(Position<E> p) throws IllegalArgumentException;

    /**
     * Make a child, without any of the fun sex positions.
     * @param p
     * @return
     * @throws IllegalArgumentException if the node doesnt exist.
     */
    Iterable<Position<E>> children(Position<E> p)
            throws IllegalArgumentException;

    /**
     * Determine how many children a catholic has.
     * @param p The catholic you want to determine the number of children for.
     * @return  The number of children raped by the priest (all of them).
     * @throws IllegalArgumentException if the node doesn't exist.
     */
    int numChildren(Position<E> p) throws IllegalArgumentException;

    /**
     * Determine if the node is an internal node within the tree.
     * @param p The node you want to test.
     * @return  Whether or not the node is an internal component.
     * @throws IllegalArgumentException if the node doesnt exist.
     */
    boolean isInternal(Position<E> p) throws IllegalArgumentException;

    /**
     * Determine if the node is an external, whether it is a leaf.
     * @param p The node that you want to test.
     * @return  Whether or not the node is a leaf.
     * @throws IllegalArgumentException If the node doesnt exist.
     */
    boolean isExternal(Position<E> p) throws IllegalArgumentException;
```

```java
    /**
     * Determine if a node is a root node.
     * @param p The node you want to test.
     * @return  Whether or not the node is a root.
     * @throws IllegalArgumentException If the node doesnt exist.
     */
    boolean isRoot(Position<E> p) throws IllegalArgumentException;

    /**
     * Determine the size of the tree.
     * @return An integer of the number of nodes in the tree.
     */
    int size();

    /**
     * Determine if the tree is empty.
     * @return Whether or not the tree is empty.
     */
    boolean isEmpty();

    /**
     * An iterator of the tree for easy passage through the tree.
     * @return an iterator.
     */
    @Override
    Iterator<E> iterator();

    /**
     * The position of the tree, usually this is a node, but can be a root,
     * like ginseng or ginger.
     * @return The iterable thing.
     */
    Iterable<Position<E>> positions();
}
```

# 7 LinkedQueue.java

```java
/**
 * Realization of a FIFO queue as an implementation of a SinglyLinkedSet.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwater
 * @author Steven Glasford
 * @version 2-21-2019
 * @param <E>
 */


public class LinkedQueue<E> implements Queue<E> {
    //an empty list
    private final SinglyLinkedList<E> list = new SinglyLinkedList<>();
    //new queue relies on the initially empty list
    public LinkedQueue() {}

    @Override
    public int size() {return list.size();}

    @Override
    public boolean isEmpty() {return list.isEmpty();}

    @Override
    public void enqueue(E element) {list.addLast(element);}

    @Override
    public E first() {return list.first();}

    @Override
    public E dequeue() {return list.removeFirst();}

}
```

# 8 Queue.java

```java
/**
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwater
 * @author Steven Glasford
 * @version 2-21-2019
 * @param <E>
 */

public interface Queue<E> {
    /**
     * Returns the number of elements in the queue
     * @return
     */
    int size();

    /**
     * Tests whether the queue is empty
     * @return
     */
    boolean isEmpty();

    /**
     * Inserts an element at the rear of the queue
     * @param e
     * @todo    modify so that this is required to throw a queue Full Exception
     *          if called on a full queue
     */
    void enqueue(E e);

    /**
     * returns, but does not remove, the first element of the queue
     * (null if empty).
     * @return
     */
    E first();

    /**
     * Removes and returns the first element of the queue (null if empty)
     * @return
     */
    E dequeue();
}
```

# 9 SinglyLinkedList.java

```
/**
 *
 * SinglyLinkedList Class
 * Code Fragments 3.14, 3.15
 * from
 * Data Structures & Algorithms, 6th edition
 * by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser
 * Wiley 2014
 * Transcribed by
 * @author Steven Glasford
 * @version January 31, 2019
 * @param <E> a generic placeholder name
 */
public class SinglyLinkedList<E> {
    /**
     *
     * @param <E> a generic placeholder name
     *
     * A subclass creating the Node
     */
    private static class Node<E>{
        //reference to the element stored at this node
        private final E element;
        //reference to the subsequent node in the list
        private Node<E> next;
        public Node(E e, Node<E> n){
            element = e;
            next = n;
        }

        /**
         *
         * @return Return the current element
         */
        public E getElement(){return element;}

        /**
         *
         * @return return the address of the next item in the linked list
         */
        public Node<E> getNext() {return next;}

        /**
         *
         * @param n the next item in the list
         */
        public void setNext(Node<E> n) {next = n;}
    }

    //head node of the list (or null if empty)
    private Node<E> head = null;
    //last node of the list (or null if empty)
    private Node<E> tail = null;
    //number of nodes in the list
    private int count = 0;

    /**
     * constructs an initially empty list
     */
```

```java
public SinglyLinkedList(){}

//access methods
/**
 *
 * @return Return the size of the linked list
 */
public int size() {return count;}

/**
 *
 * @return Determine if the linked list is empty
 */
public boolean isEmpty() {return count == 0;}

/**
 *
 * @return return the first element in the list
 *
 * returns (but does not remove) the first element
 */
public E first(){
    if (isEmpty()) return null;
    return head.getElement();
}

/**
 *
 * @return the last element in the linked list
 *
 * returns (but does not remove the last element
 */
public E last(){
    if (isEmpty()) return null;
    return tail.getElement();
}

//update methods

/**
 *
 * @param e A generic element
 *
 * adds element e to the front of the list
 */
public void addFirst(E e){
    //create and link a new node
    head = new Node<>(e, head);
    //special case: new node becomes tail also
    if (count == 0)
        tail = head;
    count++;
}

/**
 *
 * @param e A generic item
 *
 * adds element e to the end of the list
 */
public void addLast(E e) {
```

```java
        //node will eventually be the tail
        Node<E> newest = new Node<>(e,null);
        //special case: previously empty list
        if (isEmpty())
            head = newest;
        else
            tail.setNext(newest);
        tail = newest;
        count++;
    }

    /**
     *
     * @return return the item that was removed
     *
     * removes and returns the first element
     */
    public E removeFirst(){
        //nothing to remove
        if (isEmpty()) return null;
        E answer = head.getElement();
        //will become null if list had only one node
        head = head.getNext();
        count--;
        //special case as list is now empty
        if(count == 0)
            tail = null;
        return answer;
    }
}
```

## 10 Position.java

```
/**
 * Data Structures & Algorithms 6th Edition
 * Goodrick, Tamassia, Goldwasser
 * Code Fragement 7.7
 */
public interface Position<E> {
    /**
     * Returns the element stored at this position.
     *
     * @return the stored element
     * @thorws IllegalStateExceptoin if position no longer valid
     */
    E getElement( ) throws IllegalStateException;
}
```

## 11 output.txt

```
The preorder traversal of the tree.
* + / * + 5 2 - 2 1 + 2 9 - - 7 2 1 8

The inorder traversal of the tree.
5 + 2 * 2 - 1 / 2 + 9 + 7 - 2 - 1 * 8

The postorder traversal of the tree.
5 2 + 2 1 - * 2 9 + / 7 2 - 1 - + 8 *

The breadth-first traversal of the tree.
* + 8 / - * + - 1 + - 2 9 7 2 5 2 2 1

The parenthesized version of the tree.
(((((5+2)*(2-1))/(2+9))+((7-2)-1))*8)

BUILD SUCCESSFUL (total time: 0 seconds)
```

## 12 treeStructure.txt

```
          root *
              / \
            A+    8
            / \
           /   \
          /     \
         /       \
       B?         C-
       / \        / \
      /   \     F-    1
    D*     E+    / \
    / \    / \ 7    2
   /   \ 2    9
 G+     H-
 / \    / \
5   2 2    1
```