

C:\Users\Rich\Documents\NetBeansProjects\Lab08\src\AbstractTree.java

```

1 /**
2  *
3  * @author Goodrich, Tamassia, Goldwasser
4  */
5
6 import java.util.ArrayList;
7 import java.util.Iterator;
8 import java.util.List;
9
10 public abstract class AbstractTree<E> implements Tree<E> {
11     @Override
12     public boolean isInternal(Position<E> p) {return numChildren(p) > 0 ;}
13     @Override
14     public boolean isExternal(Position<E> p) {return numChildren(p) == 0;}
15     @Override
16     public boolean isRoot(Position<E> p) {return p == root();}
17     @Override
18     public boolean isEmpty() { return size() == 0;}
19
20     @Override
21     public Iterable<Position<E>> positions() {return preorder();}
22
23     /**adds positions of the subtree rooted at Position p to the given snapshot */
24     private void preorderSubtree(Position<E> p, List<Position<E>> snapshot)
25     {
26         snapshot.add(p);    // for preorder, we add position p before exploring subtree
27         for(Position<E> c: children(p))
28             preorderSubtree(c,snapshot);
29     }
30     public Iterable<Position<E>> preorder()
31     {
32         List<Position<E>> snapshot = new ArrayList<>();
33         if(!isEmpty())
34             preorderSubtree(root(),snapshot);    // fill the snapshot recursively
35         return snapshot;
36     }
37
38     //-----Postorder traversal-----
39
40     /**Adds positions of the subtree rooted at Position p to the given snapshot*/
41     private void postorderSubtree(Position<E> p, List<Position<E>> snapshot)
42     {
43         for(Position<E> c:children(p))
44             postorderSubtree(c,snapshot);
45         snapshot.add(p);    //for postorder, we add position p after exploring subtrees
46     }
47
48     /**
49     * @return an iterable collection of positions of the tree, reported in postorder.
50     */
51     public Iterable<Position<E>> postorder()
52     {
53         List<Position<E>> snapshot = new ArrayList<>();
54         if(!isEmpty())
55             postorderSubtree(root(), snapshot); // fill the snapshot recursively
56         return snapshot;

```

which extends Java's Iterable interface, thus that make the abstract tree itself iterable

implementation of abstract binary tree which extends Abstract tree

1 rule from linked binary tree class

Pre-order

root

Left → Right

recursively

```

57 }
58 //-----End of code for postOrder -----
59
60
61 //----- Nested iterator class -----
62 /* 8.16 This class adapts the iteration produced by positions() to return elements */
63 private class ElementIterator implements Iterator<E>
64 {
65     Iterator<Position<E>> posIterator = positions().iterator();
66     @Override
67     public boolean hasNext() {return posIterator.hasNext();}
68     @Override
69     public E next() {return posIterator.next().getElement();}
70     @Override
71     public void remove() {posIterator.remove();}
72 }
73
74 /******* End of Nested ElementIterator Class *****/
75
76 /**returns an iterator of the elements stored in the tree */
77 @Override
78 public Iterator<E> iterator() { return new ElementIterator();}
79
80 /**
81  * Returns an iterable collection of positions of the tree in breadth-first order.
82  * code fragment 8.21
83  * @return
84  */
85 public Iterable<Position<E>> breadthFirst()
86 {
87     List<Position<E>> snapshot = new ArrayList<>();
88     if(!isEmpty())
89     {
90         Queue<Position<E>> fringe = new LinkedList<>();
91         fringe.enqueue(root()); // start with the root
92         while(!fringe.isEmpty())
93         {
94             Position<E> p = fringe.dequeue(); // remove from front of the queue
95             snapshot.add(p); // report this position
96             for(Position<E> c: children(p))
97                 fringe.enqueue(c); // add children back to the queue.
98         }
99     }
100     return snapshot;
101 }
102
103 }
104
105     public int depth(Position<E> p)
106     {
107         if(isRoot(p))
108             return 0;
109         else
110             return 1+ depth(parent(p));
111     }
112
113     /**
114     * Height of a tree is the maximum depth of any node.
115     * @param p node position(root) of the subtree
116     * @returns the height of the subtree rooted at Position p.
117     */
118     public int height(Position<E> p)
119     {
120         int h = 0;
121         for(Position<E> c: children(p))
122             h = Math.max(h, 1+ height(c));
123         return h;
124     }

```

It's not this going to return the positions in pre order

Iterable

- 1) create array list snapshot
- 2) create linked list
- 3) enqueue the root
- 4) inside the while loop
 - dequeue the root
 - save it as a position
 - add that position to the array list

has to be root to begin with

it is the root

enqueue all children of root

then dequeue, save it, then check for that children's children

repeat that process for enqueueing all the children of p (root, s)