TREES

WHAT IS A TREE

In computer science, a tree is an abstract model of hierarchical structure (a type of nonlinear data structure)

- Trees consist of nodes with a parent-child relation
- Trees also provide a natural or organization for data,
  - Organization charts
  - File systems
  - Programming environments

Figure 8.3: Tree representing a portion of a file system.
**TREE TERMINOLOGY**

- **Root**: node without parent (A)
- **Internal node**: node with at least one child (A, B, C, F)
- **External node (a.k.a. leaf)**: node without children (E, I, J, K, G, H, D)
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc.
- **Descendant of a node**: child, grandchild, grand-grandchild, etc.
- **Subtree**: tree consisting of a node and its descendants
**Tree Terminology Cont**

- **siblings**: two nodes that are children of the same parent
- **Depth of a node**: number of ancestors
- **Height of a tree**: maximum depth of any node (3)
- **Edge**: a pair of nodes \((u, v)\) such that \(u\) is the parent of \(v\), or vice versa.
- **Path**: a sequence of nodes such that any two consecutive nodes in the sequence form an edge
FORMAL TREE DEFINITION

- Formally, we define a tree $T$ as a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties:
  - If $T$ is nonempty, it has a special node, called the root of $T$, that has no parent.
  - Each node $v$ of $T$ different from the root has a unique parent node $w$; every node with parent $w$ is a child of $w$.

- Note: a tree can be empty (no nodes)
  => Tree can be defined recursively such that a tree $T$ is either empty or consists of a node $r$, called the root of $T$, and a (possibly empty) set of subtrees whose roots are the children of $r$. 
A tree is **ordered** if there is a meaningful linear order among the children of each node;

- An order is usually visualized by arranging siblings left to right, according to their order.

An ordered tree associated with a book.
We use positions as an abstraction for a node of a tree

A position object for a tree supports the method:

- `getElement()`: Returns the element stored at this position.

**Accessor methods** for navigating through positions of a tree $T$

- `root()`: Returns the position of the root of the tree (or null if empty).
- `parent(p)`: Returns the position of the parent of position $p$ (or null if $p$ is the root).
- `children(p)`: Returns an iterable collection containing the children of position $p$ (if any).
- `numChildren(p)`: Returns the number of children of position $p$. 
**Query methods**, which are often used with conditionals statements:

+ `isInternal(p)`: Returns true if position $p$ has at least one child.
+ `isExternal(p)`: Returns true if position $p$ does not have any children.
+ `isRoot(p)`: Returns true if position $p$ is the root of the tree.

**General methods**, unrelated to the specific structure of the tree:

+ `size()`: Returns the number of positions (and hence elements) that are contained in the tree.
+ `isEmpty()`: Returns true if the tree does not contain any positions (and thus no elements).
+ `iterator()`: Returns an iterator for all elements in the tree (so that the tree itself is Iterable).
+ `positions()`: Returns an iterable collection of all positions of the tree.

Additional update methods may be defined by data structures implementing the Tree ADT. (Discussed later)
A TREE INTERFACE IN JAVA

Methods for a Tree interface:

```java
/** An interface for a tree where nodes can have an arbitrary number of children. */
public interface Tree<E> extends Iterable<E> {
    Position<E> root();
    Position<E> parent(Position<E> p) throws IllegalArgumentException;
    Iterable<Position<E>> children(Position<E> p)
        throws IllegalArgumentException;
    int numChildren(Position<E> p) throws IllegalArgumentException;
    boolean isInternal(Position<E> p) throws IllegalArgumentException;
    boolean isExternal(Position<E> p) throws IllegalArgumentException;
    boolean isRoot(Position<E> p) throws IllegalArgumentException;
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Iterable<Position<E>> positions();
}
```
AN ABSTRACTTREE BASE CLASS IN JAVA

If a concrete implementation provides three fundamental methods—root(), parent(p), and children(p)—all other behaviors of the Tree interface can be derived within the AbstractTree base class.

```java
/** An abstract base class providing some functionality of the Tree interface. */
public abstract class AbstractTree<E> implements Tree<E> {
    public boolean isInternal(Position<E> p) { return numChildren(p) > 0; }
    public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }
    public boolean isRoot(Position<E> p) { return p == root(); }
    public boolean isEmpty() { return size() == 0; }
}
```

An initial implementation of the AbstractTree base class. (We add additional functionality to this class as the chapter continues.)
Let \( p \) be a position within tree \( T \). The depth of \( p \) is the number of ancestors of \( p \), other than \( p \) itself.

The depth of \( p \) can also be recursively defined as follows:

- If \( p \) is the root, then the depth of \( p \) is 0.
- Otherwise, the depth of \( p \) is one plus the depth of the parent of \( p \).

```java
/* Returns 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));
}
```
We next define the **height** of a tree to be equal to the maximum of the depths of its positions (or zero, if the tree is empty).

If using the definition as is, the height computation become inefficient:

```java
/** Returns the height of the tree. */
private int heightBad() {
    int h = 0;
    for (Position<E> p : positions())
        if (isExternal(p))
            h = Math.max(h, depth(p));
    return h;
}
```

**Analysis:**
Positions(p): can be implemented to run in $O(n)$; Because heightBad calls algorithm depth($p$) on each leaf of $T$, its running time is $\alpha n + \sum_{p \in L}(d_p + 1))$, where $L$ is the set of leaf positions of $T$. In the worst case, the sum $\sum_{p \in L}(d_p + 1)$ is proportional to $n^2$. Thus, algorithm heightBad runs in $O(n^2)$ worst-case time.
Recursive definition to compute height.

Define the *height* of a position $p$ in a tree $T$ as follows:

- If $p$ is a leaf, then the height of $p$ is 0.
- Otherwise, the height of $p$ is one more than the maximum of the heights of $p$’s children.

The height of the root of a nonempty tree $T$, according to the recursive definition, equals the maximum depth among all leaves of tree $T$. 
The overall height of a nonempty tree can be computed by sending the root of the tree as a parameter.

Assuming that children\( (p) \) executes in \( O(c_p + 1) \) time, where \( c_p \) denotes the number of children of \( p \). Algorithm \( \text{height}(p) \) spends \( O(c_p + 1) \) time at each position \( p \) to compute the maximum, and its overall running time is

\[
O(\sum_p (c_p + 1)) = O(n + \sum_p c_p).
\]

Let \( T \) be a tree with \( n \) positions, and let \( c_p \) denote the number of children of a position \( p \) of \( T \). Then, summing over the positions of \( T \), \( \sum_p c_p = n - 1 \).
A binary tree is an ordered tree with the following properties:

- Every node has at most two children.
- Each child node is labeled as being either a left child or a right child.
- A left child precedes a right child in the order of children of a node.

The subtree rooted at a left or right child of an internal node $v$ is called a left subtree or right subtree, respectively, of $v$.

A binary tree is proper (full) if each node has either zero or two children.

- Every internal node has exactly two children.

A binary tree that is not proper is improper.

Alternative recursive definition: a binary tree is either

- a tree consisting of a single node, or
- a tree whose root has an ordered pair of children, each of which is a binary tree
BINARY TREES CONT. : ARITHMETIC EXPRESSION TREE

- Leaves are associated with variables or constants.
- Internal nodes are associated with one of the operators +, −, ∗, and /.
- Each node in such a tree has a value associated with it.
  - If a node is leaf, then its value is that of its variable or constant.
  - If a node is internal, then its value is defined by applying its operation to the values of its children.
- A typical arithmetic expression tree is a proper binary tree.
- If allowed unary operators, like negation (−), then tree is improper binary.

The value associated with the internal node labeled “/” is 2.
BINARY TREES CONT. : DECISION TREE

- Binary tree associated with a decision process
  + internal nodes: questions with yes/no answer
  + external nodes: decisions
- Example: dining decision

Want a fast meal?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>How about coffee?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starbucks</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Chipotle</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>On expense account?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Gracie’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Café Paragon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BINARY TREE ABSTRACT DATA TYPE

- Binary tree is a specialization of a tree that supports three additional accessor methods:
  - left(p): Returns the position of the left child of p (or null if p has no left child).
  - right(p): Returns the position of the right child of p (or null if p has no right child).
  - sibling(p): Returns the position of the sibling of p (or null if p has no sibling).

- We again defer the definition and implementation of specialized update methods for binary trees.
BINARY TREE ADT: BINARYTREE INTERFACE

```java
/** An interface for a binary tree, in which each node has at most two children. */
public interface BinaryTree<E> extends Tree<E> {
    /** Returns the Position of p's left child (or null if no child exists). */
    Position<E> left(Position<E> p) throws IllegalArgumentException;
    /** Returns the Position of p's right child (or null if no child exists). */
    Position<E> right(Position<E> p) throws IllegalArgumentException;
    /** Returns the Position of p's sibling (or null if no sibling exists). */
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```
/** An abstract base class providing some functionality of the BinaryTree interface. */

```java
public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
    implements BinaryTree<E> {

    /** Returns the Position of p's sibling (or null if no sibling exists). */
    public Position<E> sibling(Position<E> p) {
        Position<E> parent = parent(p);
        if (parent == null) return null; // p must be the root
        if (p == left(parent)) // p is a left child
            return right(parent); // (right child might be null)
        else
            return left(parent); // p is a right child // (left child might be null)
    }
```
BINARY TREE ADT: ABSTRACTBINARYTREE BASE CLASS

```java
13   /** Returns the number of children of Position p. */
14   public int numChildren(Position<E> p) {
15       int count=0;
16       if (left(p) != null)
17           count++;
18       if (right(p) != null)
19           count++;
20       return count;
21   }
22   /** Returns an iterable collection of the Positions representing p's children. */
23   public Iterable<Position<E>> children(Position<E> p) {
24       List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2
25       if (left(p) != null)
26           snapshot.add(left(p));
27       if (right(p) != null)
28           snapshot.add(right(p));
29       return snapshot;
30   }
```
Proposition: Let $T$ be a nonempty binary tree, and let $n$, $n_E$, $n_I$, and $h$ denote the number of nodes, number of external nodes, number of internal nodes, and height of $T$, respectively. Then $T$ has the following properties:

1. $h+1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n+1) - 1 \leq h \leq n - 1$

Also, if $T$ is proper, then $T$ has the following properties:

1. $2h+1 \leq n \leq 2^{h+1} - 1$
2. $h+1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n+1) - 1 \leq h \leq (n-1)/2$
Relating Internal Nodes to External Nodes in a Proper Binary Tree

Proposition: In a nonempty proper binary tree $T$, with $n_E$ external nodes and $n_I$ internal nodes, we have $n_E = n_I + 1$. 
LINKED STRUCTURE FOR BINARY TREES

*linked structure*, with a node that maintains references to the element stored at a position $p$ and to the nodes associated with the children and parent of $p$. 
OPERATIONS FOR UPDATING A LINKED BINARY TREE

- Means for changing the structure of content of a tree.
- Suggested update methods for a linked binary tree:
  - \texttt{addRoot}(\textit{e}): Creates a root for an empty tree, storing \textit{e} as the element, and returns the position of that root; an error occurs if the tree is not empty.
  - \texttt{addLeft}(\textit{p}, \textit{e}): Creates a left child of position \textit{p}, storing element \textit{e}, and returns the position of the new node; an error occurs if \textit{p} already has a left child.
  - \texttt{addRight}(\textit{p}, \textit{e}): Creates a right child of position \textit{p}, storing element \textit{e}, and returns the position of the new node; an error occurs if \textit{p} already has a right child.
  - \texttt{set}(\textit{p}, \textit{e}): Replaces the element stored at position \textit{p} with element \textit{e}, and returns the previously stored element.
  - \texttt{attach}(\textit{p}, T_1, T_2): Attaches the internal structure of trees \textit{T}_1 and \textit{T}_2 as the respective left and right subtrees of leaf position \textit{p} and resets \textit{T}_1 and \textit{T}_2 to empty trees; an error condition occurs if \textit{p} is not a leaf.
  - \texttt{remove}(\textit{p}): Removes the node at position \textit{p}, replacing it with its child (if any), and returns the element that had been stored at \textit{p}; an error occurs if \textit{p} has two children.

Each can be implemented in $O(1)$ worst-case time with our linked representation.
nested Node class, which implements the Position interface.

```java
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

    //------------- nested Node class -------------
    protected static class Node<E> implements Position<E> {
        private E element;       // an element stored at this node
        private Node<E> parent;  // a reference to the parent node (if any)
        private Node<E> left;    // a reference to the left child (if any)
        private Node<E> right;   // a reference to the right child (if any)
        /** Constructs a node with the given element and neighbors. */
        public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
            element = e;
            parent = above;
            left = leftChild;
            right = rightChild;
        }
        // accessor methods
        public E getElement() { return element; }
        public Node<E> getParent() { return parent; }
        public Node<E> getLeft() { return left; }
        public Node<E> getRight() { return right; }
        // update methods
        public void setElement(E e) { element = e; }
        public void setParent(Node<E> parentNode) { parent = parentNode; }
        public void setLeft(Node<E> leftChild) { left = leftChild; }
        public void setRight(Node<E> rightChild) { right = rightChild; }
    }
```
**Factory function to create a new node storing element e. */
```java
protected Node<E> createNode(E e, Node<E> parent,
                              Node<E> left, Node<E> right) {
    return new Node<E>(e, parent, left, right);
}
```

// LinkedBinaryTree instance variables
```java
protected Node<E> root = null; // root of the tree
private int size = 0; // number of nodes in the tree
```

// constructor
```java
public LinkedBinaryTree() {} // constructs an empty binary tree
```
JAVA IMPLEMENTATION OF A LINKED BINARY TREE STRUCTURE

```java
41     // nonpublic utility
42     /** Validates the position and returns it as a node. */
43     protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
44         if (!(p instanceof Node))
45             throw new IllegalArgumentException("Not valid position type");
46         Node<E> node = (Node<E>) p;  // safe cast
47         if (node.getParent() == node)  // our convention for defunct node
48             throw new IllegalArgumentException("p is no longer in the tree");
49         return node;
50     }
51
52     // accessor methods (not already implemented in AbstractBinaryTree)
53     /** Returns the number of nodes in the tree. */
54     public int size() {
55         return size;
56     }
57
58     /** Returns the root Position of the tree (or null if tree is empty). */
59     public Position<E> root() {
60         return root;
61     }
62 ```
JAVA IMPLEMENTATION OF A LINKED BINARY TREE STRUCTURE 4

63    /** Returns the Position of p's parent (or null if p is root). */
64    public Position<E> parent(Position<E> p) throws IllegalArgumentException {
65        Node<E> node = validate(p);
66        return node.getParent();
67    }
68
69    /** Returns the Position of p's left child (or null if no child exists). */
70    public Position<E> left(Position<E> p) throws IllegalArgumentException {
71        Node<E> node = validate(p);
72        return node.getLeft();
73    }
74
75    /** Returns the Position of p's right child (or null if no child exists). */
76    public Position<E> right(Position<E> p) throws IllegalArgumentException {
77        Node<E> node = validate(p);
78        return node.getRight();
79    }
// update methods supported by this class
/** Places element e at the root of an empty tree and returns its new Position. */
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. */
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 left child");
    Node<E> child = createNode(e, parent, null, null);
    parent.setLeft(child);
    size++;
    return child;
}
JAVA IMPLEMENTATION OF A LINKED BINARY TREE STRUCTURE 6

```java
101 /*** Creates a new right child of Position p storing element e; returns its Position. */
102 public Position<E> addRight(Position<E> p, E e)
103     throws IllegalArgumentException {
104     Node<E> parent = validate(p);
105     if (parent.getRight() != null)
106         throw new IllegalArgumentException("p already has a right child");
107     Node<E> child = createNode(e, parent, null, null);
108     parent.setRight(child);
109     size++;
110     return child;
111 }
112
113 /*** Replaces the element at Position p with e and returns the replaced element. */
114 public E set(Position<E> p, E e) throws IllegalArgumentException {
115     Node<E> node = validate(p);
116     E temp = node.getElement();
117     node.setElement(e);
118     return temp;
119 }
```
/** Attaches trees t1 and t2 as left and right subtrees of external p. */

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();
    if (!t1.isEmpty()) {
        // attach t1 as left subtree of node
        t1.root.setParent(node);
        node.setLeft(t1.root);
        t1.root = null;
        t1.size = 0;
    }
    if (!t2.isEmpty()) {
        // attach t2 as right subtree of node
        t2.root.setParent(node);
        node.setRight(t2.root);
        t2.root = null;
        t2.size = 0;
    }
}
remove(p): intentionally sets the parent field of a deleted node to refer to itself, in accordance with our conventional representation of a defunct node (as detected within the validate method).

// help garbage collection

// our convention for defunct node
PERFORMANCE OF THE LINKED BINARY TREE IMPLEMENTATION

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>root, parent, left, right, sibling, children, numChildren</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isInternal, isExternal, isRoot</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addRoot, addLeft, addRight, set, attach, remove</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>depth($p$)</td>
<td>$O(d_p + 1)$</td>
</tr>
<tr>
<td>height</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Running times for the methods of an $n$-node binary tree implemented with a linked structure. The space usage is $O(n)$. $d_p$ is depth of node.
ARRAY-BASED REPRESENTATION OF A BINARY TREE

- Utilize the way of numbering the positions of $T$.
- For every position $p$ of $T$, let $f(p)$ be the integer defined as follows.
  - If $p$ is the root of $T$, then $f(p) = 0$.
  - If $p$ is the left child of position $q$, then $f(p) = 2f(q)+1$.
  - If $p$ is the right child of position $q$, then $f(p) = 2f(q)+2$.

$f$ is known as level numbering of the positions in a binary tree $T$, for it numbers the positions on each level of $T$ in increasing order from left to right.
an array-based structure $A$, with the element at position $p$ of $T$ stored at index $f(p)$ of the array.
ARRAY-BASED REPRESENTATION OF A BINARY TREE

**Advantage:**

+ Position $p$ can be represented by the single integer $f(p)$,
+ Position-based methods such as root, parent, left, and right can be implemented using simple arithmetic operations on the number $f(p)$.
  - The left child of $p$ has index $2f(p) + 1$,
  - The right child of $p$ has index $2f(p) + 2$,
  - The parent of $p$ has index $\lfloor (f(p) - 1)/2 \rfloor$.

**Disadvantage:**

+ Space usage of an array-based representation depends greatly on the shape of the tree;
  - Worst case space usage: $N = 2^n - 1$, where $n$ is the number of nodes in $T$.
+ Many update operations for trees cannot be efficiently supported.
  - EX> Removing a node and promoting its child takes $O(n)$ time: the node and all its descendants.
General trees have no a priori limit on the number of children that a node may have.

Each node stores a single `container` of references to its children.
### LINKED STRUCTURE FOR GENERAL TREES 2

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>root, parent, isRoot, isInternal, isExternal</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>numChildren($p$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>children($p$)</td>
<td>$O(c_p + 1)$</td>
</tr>
<tr>
<td>depth($p$)</td>
<td>$O(d_p + 1)$</td>
</tr>
<tr>
<td>height</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
A traversal visits the nodes of a tree in a systematic manner
In a preorder traversal, a node is visited before its descendants
Application: print a structured document

Algorithm `preOrder(v)`
`visit(v)`
for each child `w` of `v`
`preorder(w)`
In a postorder traversal, a node is visited after its descendants.

Application: compute space used by files in a directory and its subdirectories.

**Algorithm** `postOrder(v)`

For each child `w` of `v`:

- `postOrder(w)`
- `visit(v)`
In an in-order traversal a node is visited after its left subtree and before its right subtree.

Application: draw a binary tree

```
x(v) = in-order rank of v
y(v) = depth of v
```

**Algorithm** `inOrder(v)`

```
if left(v) ≠ null
    inOrder(left(v))
visit(v)
if right(v) ≠ null
    inOrder(right(v))
```
Breadth-first traversal: traverses a tree so that we visit all the positions at depth $d$ before we visit the positions at depth $d+1$.

**Algorithm** breadthfirst( ):
- Initialize queue $Q$ to contain root( )
- **while** $Q$ not empty **do**
  - $p = Q$.dequeue( )
  - perform the “visit” action for position $p$
  - **for** each child $c$ in children($p$) **do**
    - $Q$.enqueue($c$)
Let $S$ be a set whose unique elements have an order relation. A binary search tree for $S$ is a proper binary tree $T$ such that, for each internal position $p$ of $T$:

- Position $p$ stores an element of $S$, denoted as $e(p)$.
- Elements stored in the left subtree of $p$ (if any) are less than $e(p)$.
- Elements stored in the right subtree of $p$ (if any) are greater than $e(p)$.

Running time of searching in a binary search tree $T$ is proportional to the height of $T$. 
We can implement the iterator method by adapting an iteration produced by the positions method.

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

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

We first need to choose tree traversal algorithms to implement the positions method.

Ex>
```
public Iterable<Position<E>> positions() { return preorder(); }
```
private `preorderSubtree` method allows us to parameterize the recursive process with a specific position of the tree that serves as the root of a subtree to traverse.

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

public `preorder` method: has the responsibility of creating an empty list for the snapshot buffer, and invoking the recursive method at the root of the tree.

```java
public Iterable<Position<E>> preorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty())
        preorderSubtree(root(), snapshot); // fill the snapshot recursively
    return snapshot;
}
```
/** Adds positions of the subtree rooted at Position p to the given snapshot. */
private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {
    for (Position<E> c : children(p))
        postorderSubtree(c, snapshot);
    snapshot.add(p); // for postorder, we add position p after exploring subtrees
}

/** Returns an iterable collection of positions of the tree, reported in postorder. */
public Iterable<Position<E>> postorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty())
        postorderSubtree(root(), snapshot); // fill the snapshot recursively
    return snapshot;
}
The inorder traversal algorithm, because it explicitly relies on the notion of a left and right child of a node, only applies to binary trees. (define it in AbstractBinaryTree class.)

```java
/** Adds positions of the subtree rooted at Position p to the given snapshot. */
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. */
public Iterable<Position<E>> inorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if (!isEmpty())
        inorderSubtree(root(), snapshot);  // fill the snapshot recursively
    return snapshot;
}

/** Overrides positions to make inorder the default order for binary trees. */
public Iterable<Position<E>> positions() {
    return inorder();
}
```
/** Returns 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 LinkedList<>();
        fringe.enqueue(root()); // start with the root
        while (!fringe.isEmpty()) {
            Position<E> p = fringe.dequeue(); // remove from front of the queue
            snapshot.add(p); // report this position
            for (Position<E> c : children(p)) {
                fringe.enqueue(c); // add children to back of queue
            }
        }
    }
    return snapshot;
}
Preorder traversal of the tree can be used to produce a table of contents for the document.

```
for (Position<E> p : T.preorder())
    System.out.println(p.getElement());
```

```
/** Prints preorder representation of subtree of T rooted at p having depth d. */
public static <E> void printPreorderIndent(Tree<E> T, Position<E> p, int d) {
    System.out.println(spaces(2*d) + p.getElement()); // indent based on d
    for (Position<E> c : T.children(p))
        printPreorderIndent(T, c, d+1); // child depth is d+1
}
Applications of Tree Traversals: Evaluate Arithmetic Expressions

- Specialization of a post-order traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
Algorithm `evalExpr(v)`
  if `isExternal(v)`
    return `v.element()`
  else
    `x ← evalExpr(left(v))`
    `y ← evalExpr(right(v))`
    ◊ ← operator stored at v
    return `x ◊ y`
```

```
+  
  -  
  ×  
  2  

  ×  
  3  
  2

  ×  
  5  
  1
```

© 2014 Goodrich, Tamassia, Goldwasser
APPLICATIONS OF TREE TRAVERSALS:
PRINT ARITHMETIC EXPRESSIONS

- Specialization of an in-order traversal
  + print operand or operator when visiting node
  + print "(" before traversing left subtree
  + print ")" after traversing right subtree

Algorithm `printExpression(v)`

```
if left(v) != null
  print(" (")
inOrder(left(v))
print(v.element())
if right(v) != null
  inOrder(right(v))
print(" )")
```

```
+ 
  /  
  ×    ×
  /     /
  2     3
  /  
 a 1

((2 \times (a - 1)) + (3 \times b))
```
**Euler tour traversal** are generic traversal of a tree

- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (pre-visit)
  - from below (in-visit)
  - on the right (post-visit)
- Complexity of the walk is $O(n)$,
EULER TOUR TRAVERSAL CONT.

**Algorithm** eulerTour(T, p):

perform the “pre visit” action for position p
for each child c in T.children(p) do
   eulerTour(T, c) \hspace{1cm} \{ recursively tour the subtree rooted at c \}
perform the “post visit” action for position p

**Algorithm** eulerTourBinary(T, p):

perform the “pre visit” action for position p
if p has a left child lc then
   eulerTourBinary(T, lc) \hspace{1cm} \{ recursively tour the left subtree of p \}
perform the “in visit” action for position p
if p has a right child rc then
   eulerTourBinary(T, rc) \hspace{1cm} \{ recursively tour the right subtree of p \}
perform the “post visit” action for position p