# Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

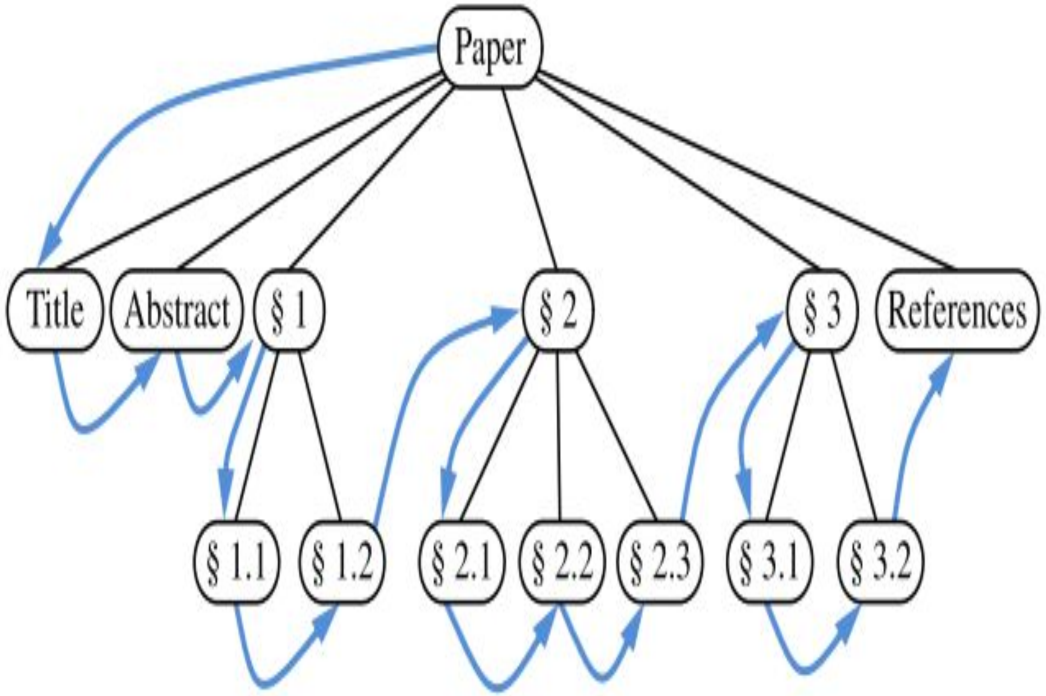
Slides are prepared from
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

# Tree Traversal Algorithms

- A traversal of a tree T is a systematic way of accessing, or “visiting,” all the positions of T
- The specific action associated with the “visit” of a position p depends on the application of this traversal
- We will learn following traversal techniques:
  - Preorder Traversal
  - Postorder Traversal
  - Breadth-First Tree Traversal
  - Inorder Traversal

# Preorder Traversal

- In a preorder traversal of a tree T, the *root of T is visited first* and then the *subtrees rooted at its children are traversed recursively*
- If the tree is ***ordered***, then the *subtrees are traversed according to the order of the children*

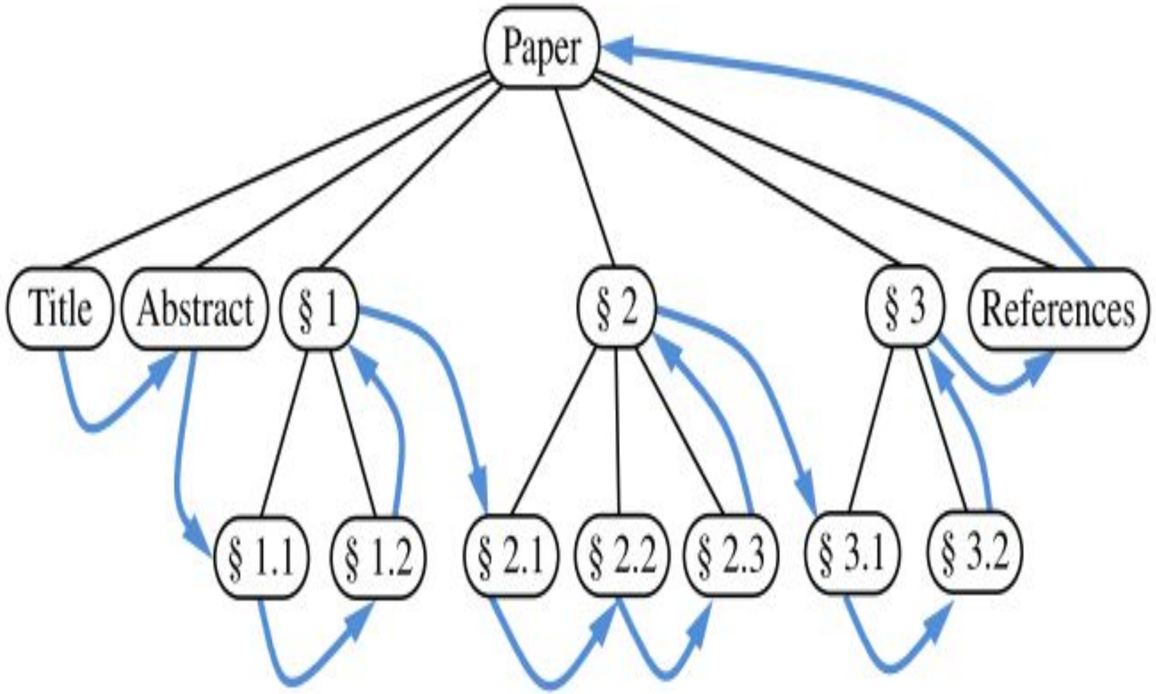



**Algorithm** preorder($p$):

  perform the "visit" action for position $p$ { this happens before any recursion }
  **for** each child $c$ in children($p$) **do**
    preorder($c$) { recursively traverse the subtree rooted at $c$ }

# Postorder Traversal

- The postorder traversal *recursively traverses the subtrees* rooted at the children of the root first, and then *visits the root*
- It is opposite of the preorder traversal



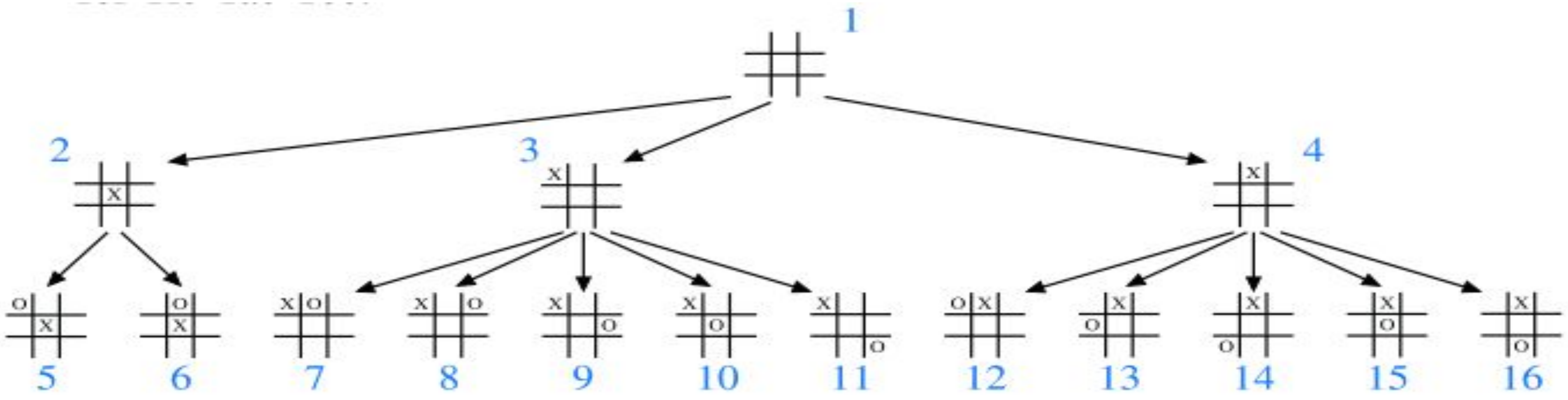

**Algorithm** postorder($p$):
  **for** each child $c$ in children($p$) **do**
    postorder($c$)  { recursively traverse the subtree rooted at $c$ }
  perform the "visit" action for position $p$  { this happens after any recursion }

# Running-Time Analysis

- Both preorder and postorder traversal algorithms are efficient ways to access all the positions of a tree
- At each position $p$, the nonrecursive part of the traversal algorithm requires time $O(c_p+1)$, where $c_p$ is the number of children of $p$, under the assumption that the "visit" itself takes $O(1)$ time
- The analysis of traversal algorithms is similar to that of algorithm ***height***
- The overall running time for the traversal of tree $T$ is $O(n)$, where $n$ is the number of positions in the tree

# Breadth-First Tree Traversal

- Breadth-First Tree Traversal traverses a tree so that we visit all the positions at depth ***d*** before we visit the positions at depth ***d+1***
- A breadth-first traversal is a common approach used in software for playing games
- A game tree represents the possible choices of moves that might be made by a player (or computer) during a game, with the root of the tree being the initial configuration for the game



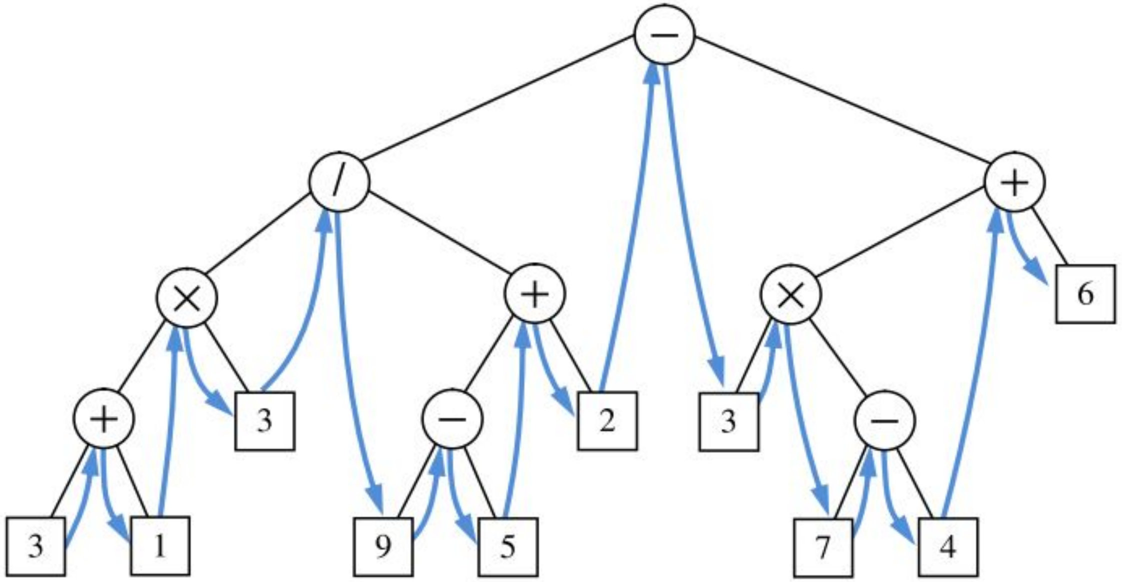
# Breadth-First Tree Traversal

- A breadth-first traversal is not recursive, since we are not traversing entire subtrees at once
- We use a queue to produce a FIFO (i.e., first-in first-out) semantics for the order in which we visit nodes
- The overall running time is O(n), due to the n calls to enqueue and n calls to dequeue

```
Algorithm breadthfirst():
  Initialize queue Q to contain root()
  while Q not empty do
    p = Q.dequeue()                     { p is the oldest entry in the queue }
    perform the "visit" action for position p
    for each child c in children(p) do
      Q.enqueue(c)      { add p's children to the end of the queue for later visits }
```

# Inorder Traversal of a Binary Tree

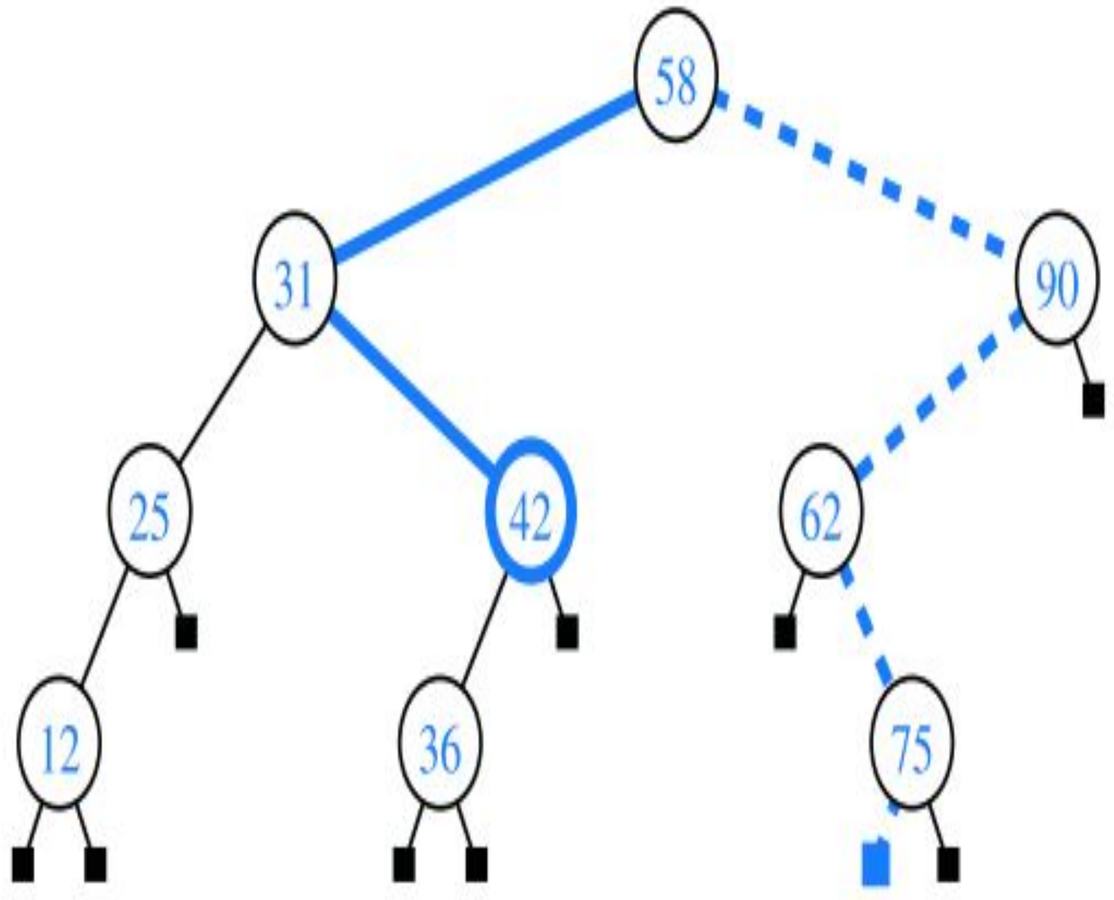
- An inorder traversal *visits a position between the recursive traversals of its left and right subtrees*
- The inorder traversal of a binary tree *T* can be informally viewed as *visiting the nodes of T* ***"from left to right"***
- For every position *p*, the inorder traversal visits *p* after all the positions in the left subtree of *p* and before all the positions in the right subtree of *p*




**Algorithm** inorder($p$):
    **if** $p$ has a left child $lc$ **then**
        inorder($lc$)     { recursively traverse the left subtree of $p$ }
    perform the "visit" action for position $p$
    **if** $p$ has a right child $rc$ **then**
        inorder($rc$)     { recursively traverse the right subtree of $p$ }

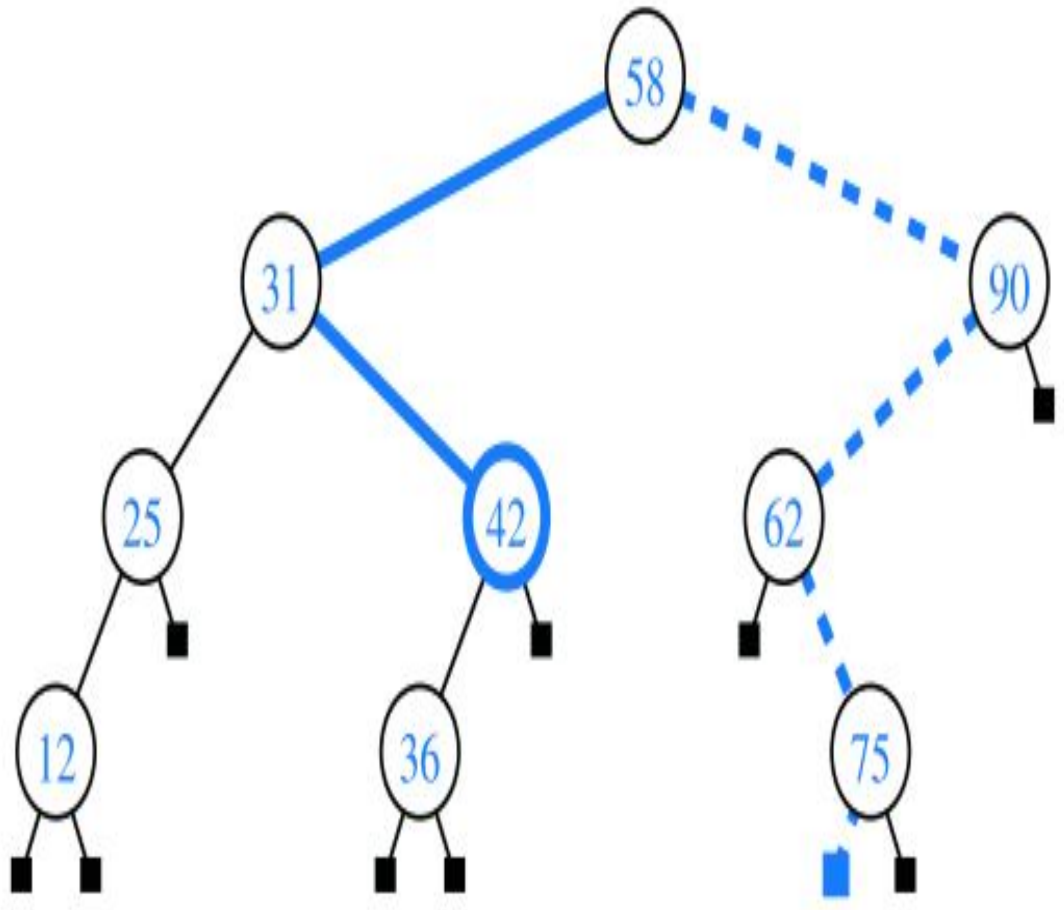
# Binary Search Trees

- A binary search tree is an ordered sequence of elements in a binary tree
- 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)***



# Binary Search Trees

- A Binary tree assure that an *inorder traversal* of a binary search tree ***T*** visits the elements in *nondecreasing order*
- We can use a binary search tree ***T*** for set ***S*** to find whether a given search value ***v*** is in ***S***, by traversing a path down the tree ***T***
- The process of searching is as follows:
  - start at the ***root***
  - At each internal position ***p*** encountered, we compare our search value v with the element ***e(p)*** stored at ***p***
  - If ***v < e(p)***, then the search continues in the **left subtree** of ***p***
  - If ***v = e(p)***, then the search **terminates successfully**
  - If ***v > e(p)***, then the search continues in the **right subtree** of ***p***
  - Finally, if we reach a leaf, the search **terminates unsuccessfully**
- The **running time** of searching in a **binary search tree *T*** is proportional to the **height** of ***T***



# Implementing Tree Traversals in Java

- The tree ADT we defined supports following methods:
  - **iterator()**: Returns an iterator for all elements in the tree.
  - **positions()**: Returns an iterable collection of all positions of the tree
- The tree traversal algorithms will use these iteratior for concrete implementations within the AbstractTree or AbstractBinaryTree base classes
- An iteration of all elements of a tree can easily be produced if we have an iteration of all positions of that tree

```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(); }
```

# Implementing Tree Traversals in Java

- To implement the positions() method, we have a choice of tree traversal algorithms
- We provide public implementations of each strategy that can be called directly by a user of our class
- one of those as a default order for the positions method of the AbstractTree class
    - For example, a public method, preorder(), that returns an iteration of the positions of a tree in preorder

```
public Iterable<Position<E>> positions() { return preorder(); }
```

# Implementing Tree Traversals in Java

- **Preorder Traversal**
- Define a private utility method, preorderSubtree, which allows us to parameterize the recursive process with a specific position of the tree that serves as the root of a subtree to traverse
- A recursive subroutine for performing a preorder traversal of the subtree rooted at position p of a tree
- This code should be included within the body of the AbstractTree class

```
/** Adds positions of the subtree rooted at Position p to the given snapshot. */
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);
}
```

# Implementing Tree Traversals in Java

- **Preorder Traversal**
- The 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 (assuming the tree is nonempty)
- We rely on a java.util.ArrayList instance as an Iterable instance for the snapshot buffer

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

# Implementing Tree Traversals in Java

- **Postorder Traversal**
- Support for performing a postorder traversal of a tree
- This code should be included within the body of the AbstractTree class

```
/** 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;
}
```

# Implementing Tree Traversals in Java

- **Breadth-First Traversal**
- The breadth-first traversal algorithm is not recursive; it relies on a queue of positions to manage the traversal process
- An implementation of the breadth-first traversal algorithm in the context of our AbstractTree class

```java
/** 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 LinkedQueue<>( );
    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;
}
```

# Implementing Tree Traversals in Java

- **Inorder Traversal for Binary Trees**
- The preorder, postorder, and breadth-first traversal algorithms are applicable to all trees
- 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
- We include its definition within the body of the AbstractBinaryTree class

```
/** 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();
}
```