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

General Trees

- Tree is one of the most important ***nonlinear data structures***
- Trees provide a natural organization for data, and consequently have become ubiquitous structures in
  - file systems,
  - graphical user interfaces,
  - databases,
  - websites, and
  - many other computer systems
- The relationships in a tree are ***hierarchical***, with some objects being "above" and some "below" others
- Actually, the main terminology for tree data structures comes from ***family trees***, with the terms "parent," "child," "ancestor," and "descendant"

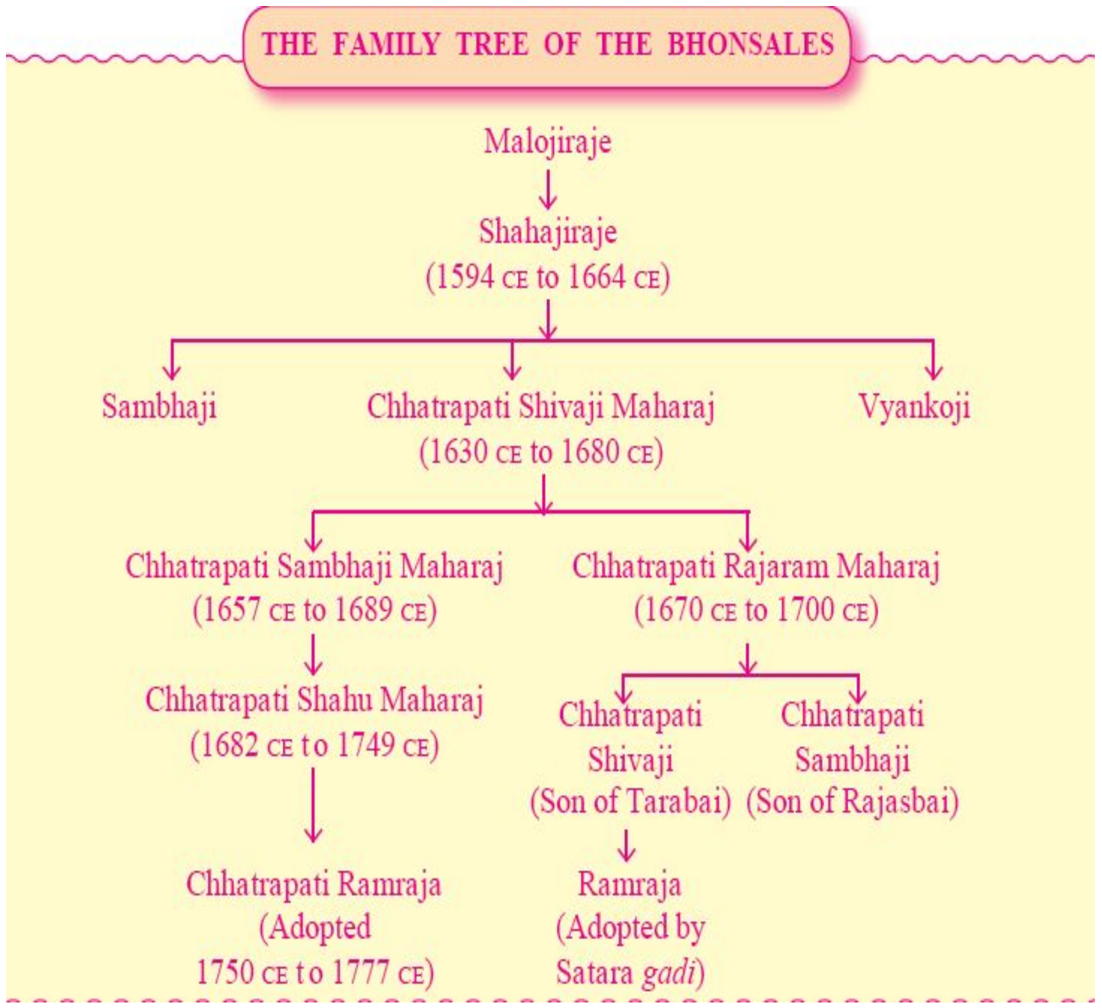
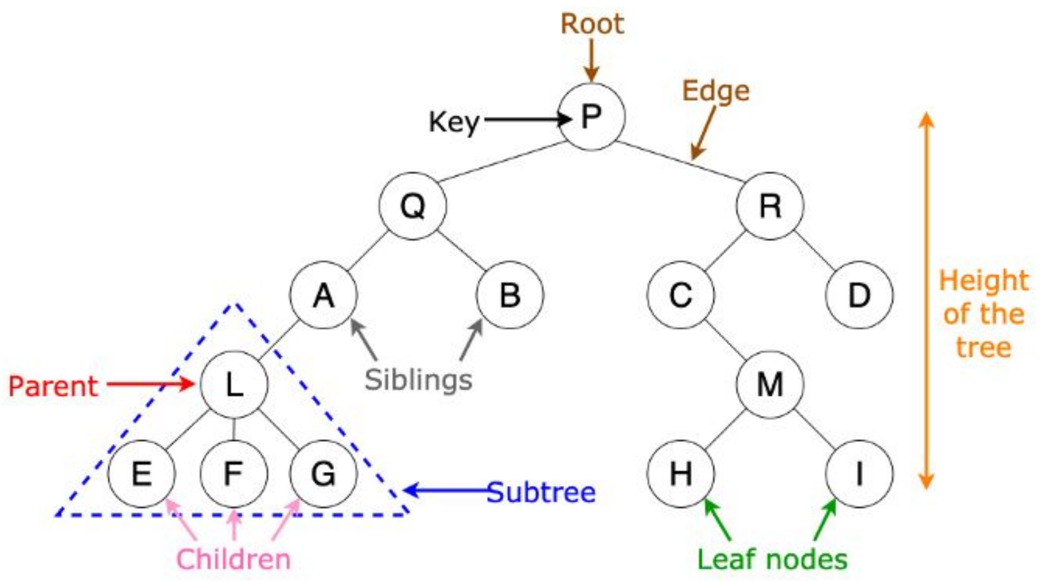

THE FAMILY TREE OF THE BHONSALES
Malojiraje
Shahajiraje (1594 CE to 1664 CE)
Sambhaji
Chhatrapati Shivaji Maharaj (1630 CE to 1680 CE)
Vyankoji
Chhatrapati Sambhaji Maharaj (1657 CE to 1689 CE)
Chhatrapati Rajaram Maharaj (1670 CE to 1700 CE)
Chhatrapati Shahu Maharaj (1682 CE to 1749 CE)
Chhatrapati Shivaji (Son of Tarabai)
Chhatrapati Sambhaji (Son of Rajasbai)
Chhatrapati Ramraja (Adopted 1750 CE to 1777 CE)
Ramraja (Adopted by Satara *gadi*)

Image reference-
[illegible]

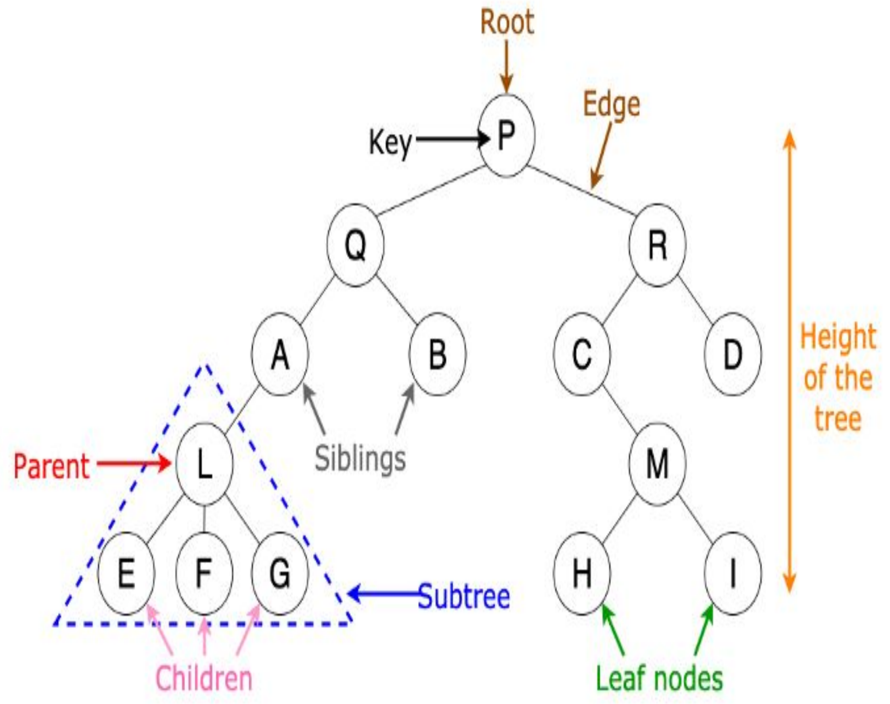
Tree Definitions

- A tree is an abstract data type that stores elements *hierarchically*
- With the exception of the top element, each element in a tree has a ***parent element*** and zero or more ***children elements***
- A tree is usually visualized by placing *elements inside ovals or rectangles*, and by drawing the *connections between parents and children with straight lines*
- The top element the ***root*** of the tree, but it is drawn as the highest element, with the other elements being connected below **(just the opposite of a botanical tree)**
- **Applications:**
  - **Organization charts**
  - **File systems**
  - **Programming environments**



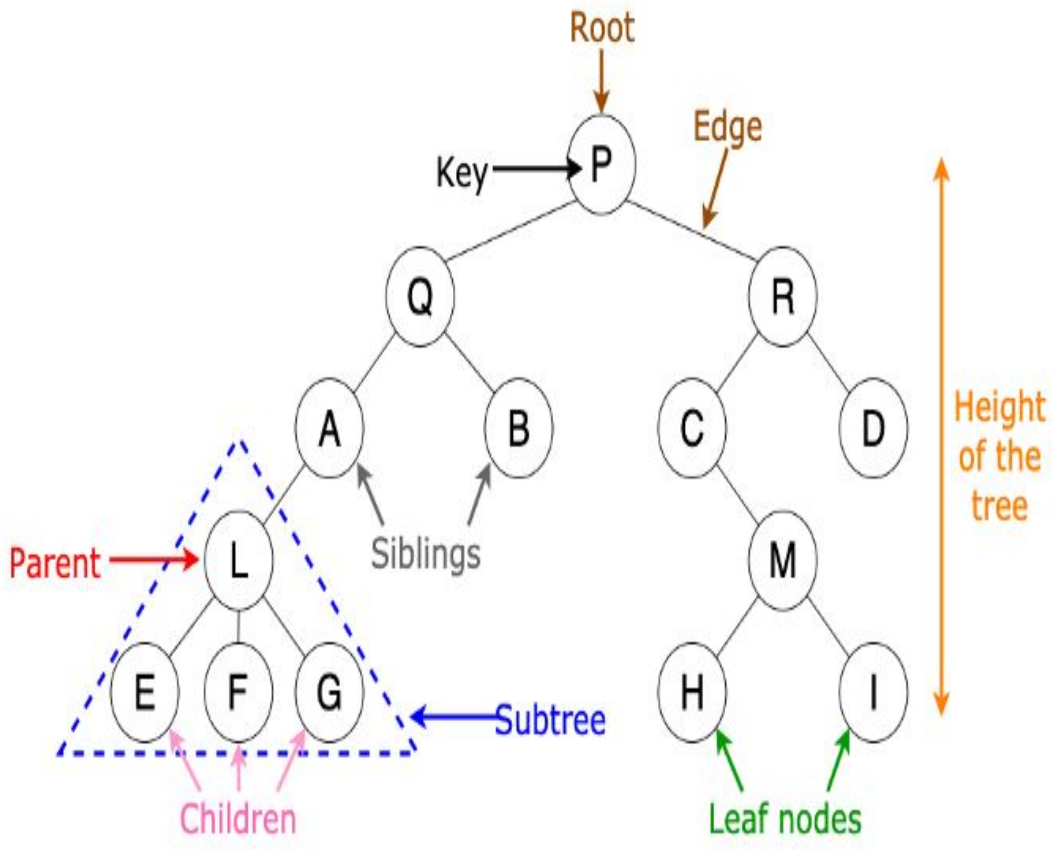
Formal Tree Definition

- 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 that according to our definition, a tree can be empty, meaning that it does not have any nodes
- One can define a tree ***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



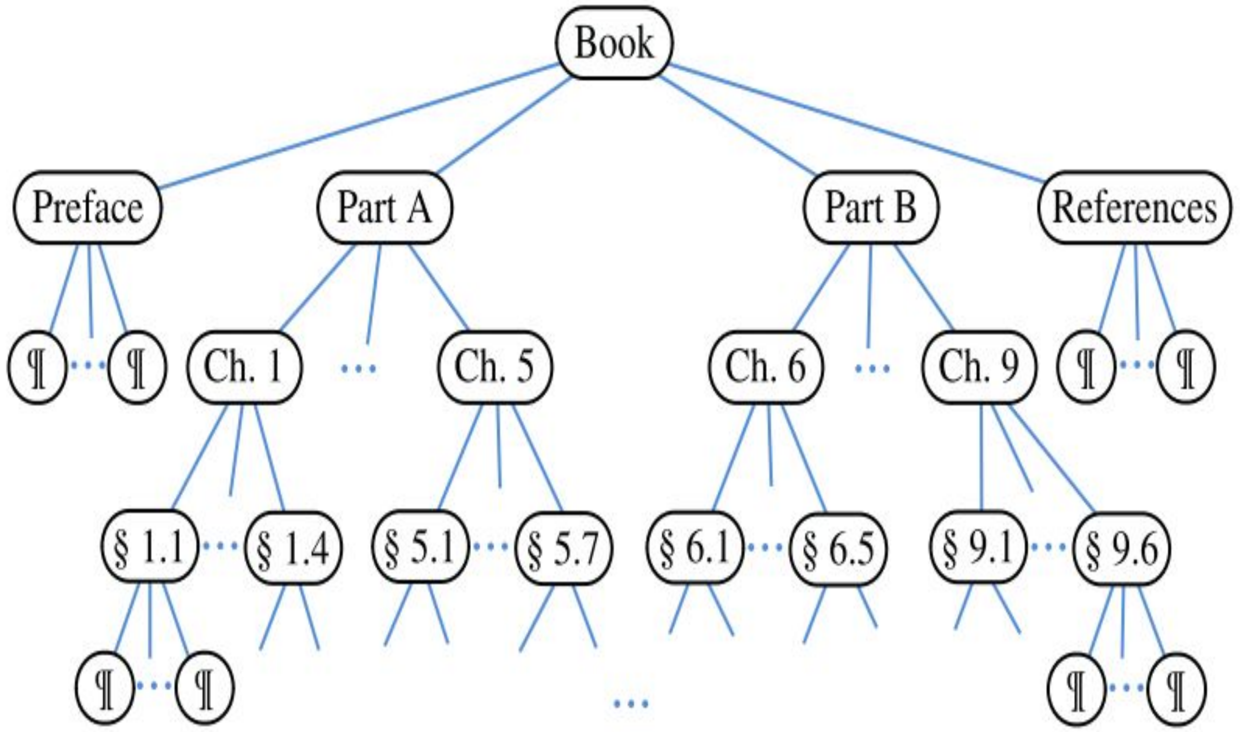
Tree Properties

- Two nodes that are children of the same parent are ***siblings***
- A node v is *external* if v has no children also known as ***leaves***
- A node v is *internal* if it has one or more children
- An ***edge*** of tree T is *a pair of nodes (u,v)* such that u is the parent of v, or vice versa
    - example (P,Q)
- A ***path*** of T is *a sequence of nodes* such that any two consecutive nodes in the sequence form an edge
    - Example: P-Q-B; P-R-C-M-H
- ***Depth*** of a node: number of ancestors
- ***Height*** of a tree: maximum depth of any node
- ***Subtree***: tree consisting of a node and its descendants



Ordered Trees

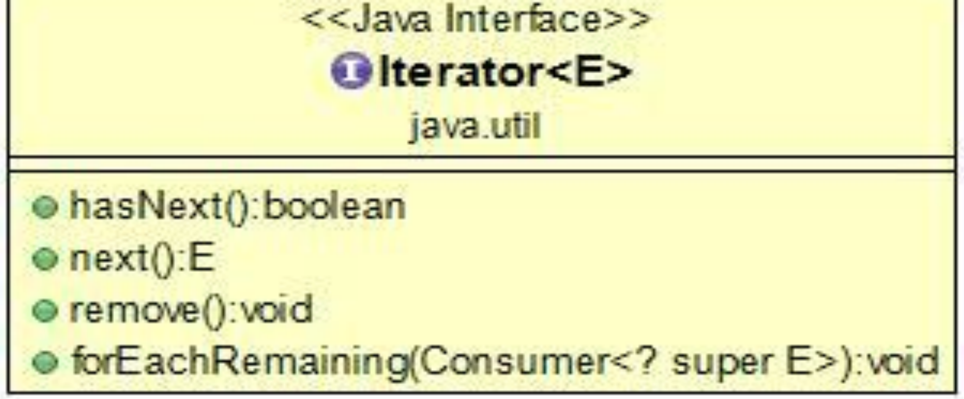
- A tree is ordered if there is a *meaningful linear order* among the children of each node
- one can identify the children of a node as being the first, second, third, and so on
- An order is usually visualized by *arranging siblings left to right*, according to their order
- Example: The components of a structured document, such as a book, are hierarchically organized as a tree whose internal nodes are parts, chapters, and sections, and whose leaves are paragraphs, tables, figures, and so on.



The Tree Abstract Data Type

- The tree ADT ***accessor methods***, allowing a user to navigate the various 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
- The tree ADT supports the following ***query methods***:
  - **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
- The tree ADT supports the following ***generic methods***:
  - **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

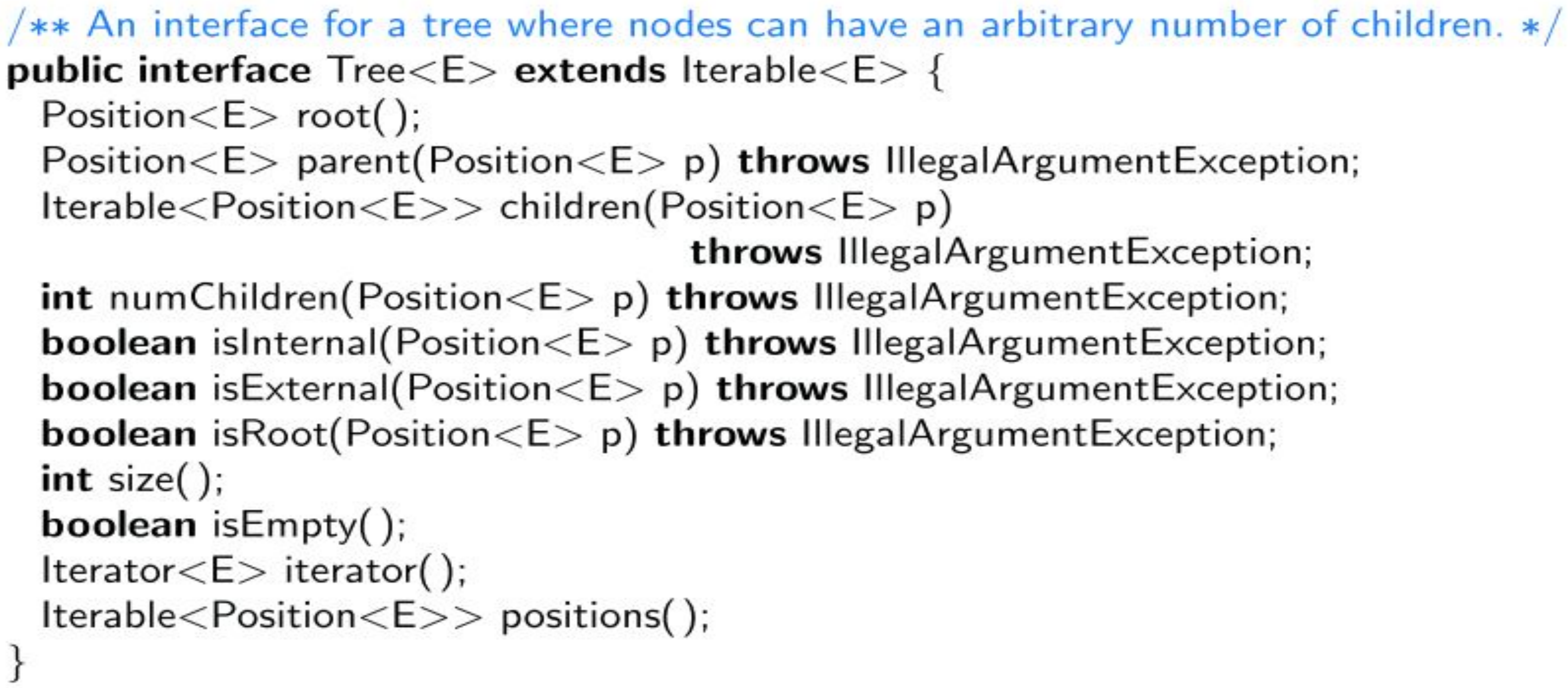
Java Iterator interface



- Iterators are used in Collection framework in Java to retrieve elements one by one
  - **hasNext()**: Returns true if the Iterator has more elements, and false if not
  - **next()**: Return the next element from the Iterator
  - **remove()**: Removes the latest element returned from next() from the Collection the Iterator is iterating over
  - **forEachRemaining()**: Iterates over all remaining elements in the Iterator and calls a Java Lambda Expression passing each remaining element as parameter to the lambda expression
- To use a Java Iterator you will have to ***obtain an Iterator instance from the collection*** of objects you want to iterate over
- The obtained ***Iterator keeps track of the elements in the underlying collection*** to make sure you iterate through all of them
- If you modify the underlying collection while iterating through an Iterator pointing to that collection, the Iterator will typically detect it, and throw an exception the next time you try to obtain the next element from the Iterator

A Tree Interface in Java

- The Java Tree interface extend Java's ***Iterable interface*** (and we include a declaration of the required iterator() method)



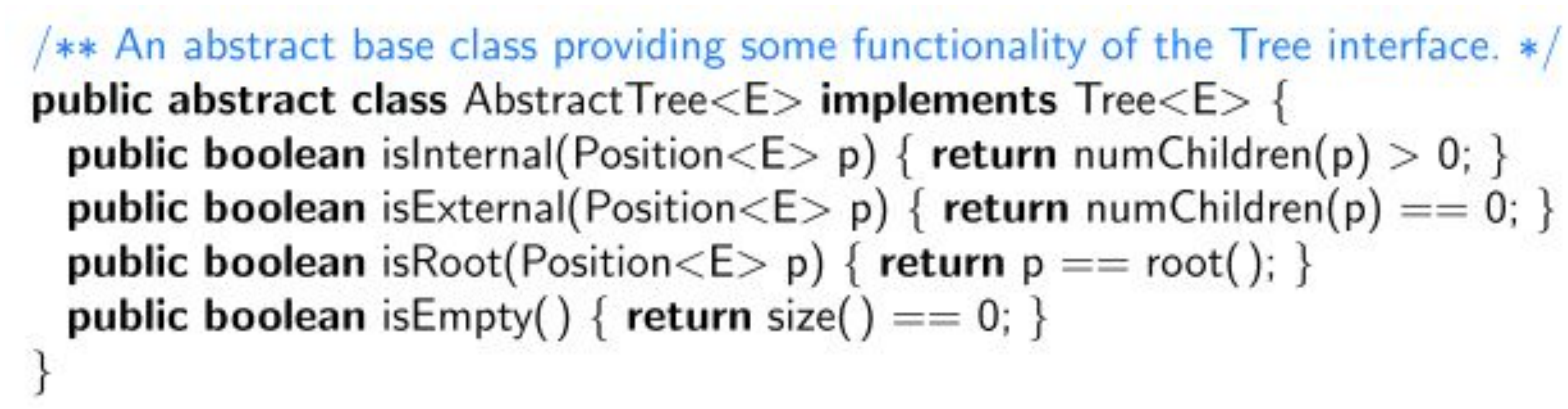
```
/** An interface for a tree where nodes can have an arbitrary number of children. */
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                                  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 Abstract Class in Java

- An ***abstract class*** may *define concrete implementations for some of its methods*, while leaving other abstract methods without definition
- An ***abstract class*** is designed to *serve as a base class, through inheritance*, for one or more concrete implementations of an interface
- When *some of the functionality of an interface is implemented* in an abstract class, less work remains to complete a concrete implementation
- The **standard Java libraries** include many such abstract classes, including several within the Java Collections Framework
    - For example, there is an *AbstractCollection* class that implements some of the functionality of the Collection interface,
    - an ***AbstractQueue*** class that implements some of the functionality of the Queue interface, and
    - an ***AbstractList*** class that implements some of the functionality of the List interface

An AbstractTree Base Class in Java

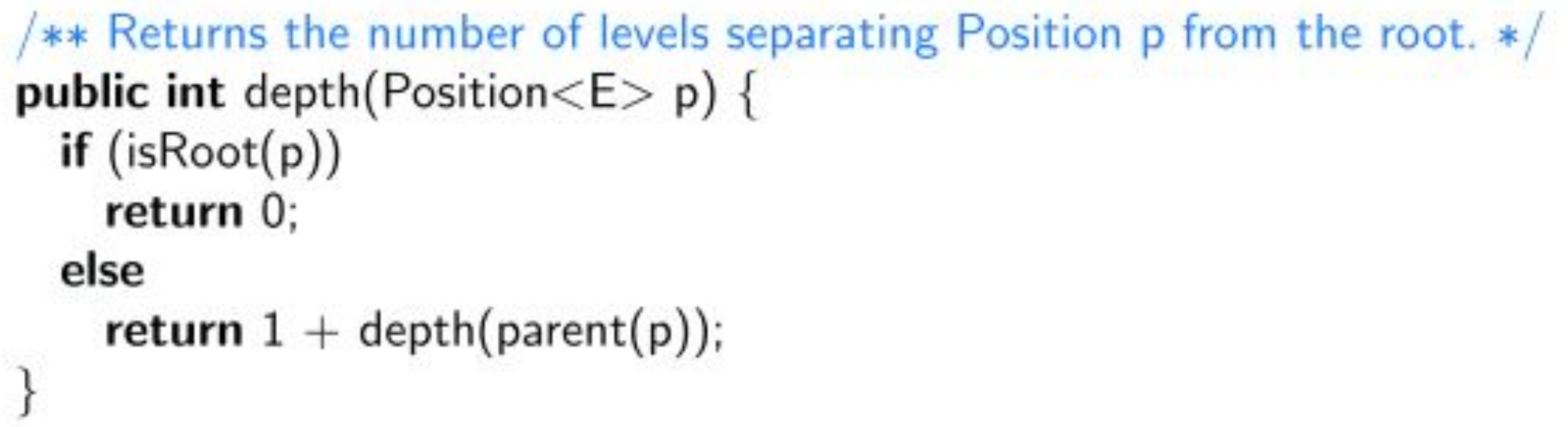
- An AbstractTree base class, demonstrating how many tree-based algorithms can be described independently of the low-level representation of a tree data structure



```
/** 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; }
}
```

Computing Depth of element

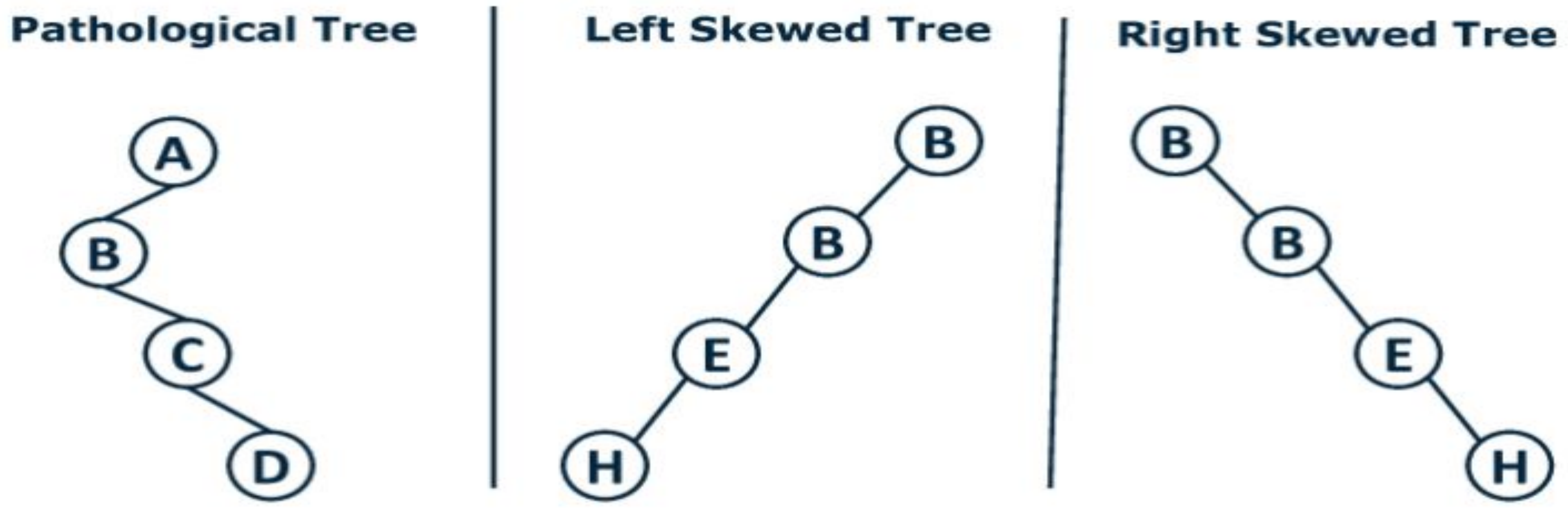
- Let p be a position within tree T, then the **depth** of p is the *number of ancestors of p, other than p itself*
- Note that this definition implies that the depth of the root of T is 0
- The depth of p can also be recursively defined as follows:
  - If p is the **root**, then the *depth* of p is 0
  - Otherwise, the *depth* of p is *one plus the depth of the parent of p*



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

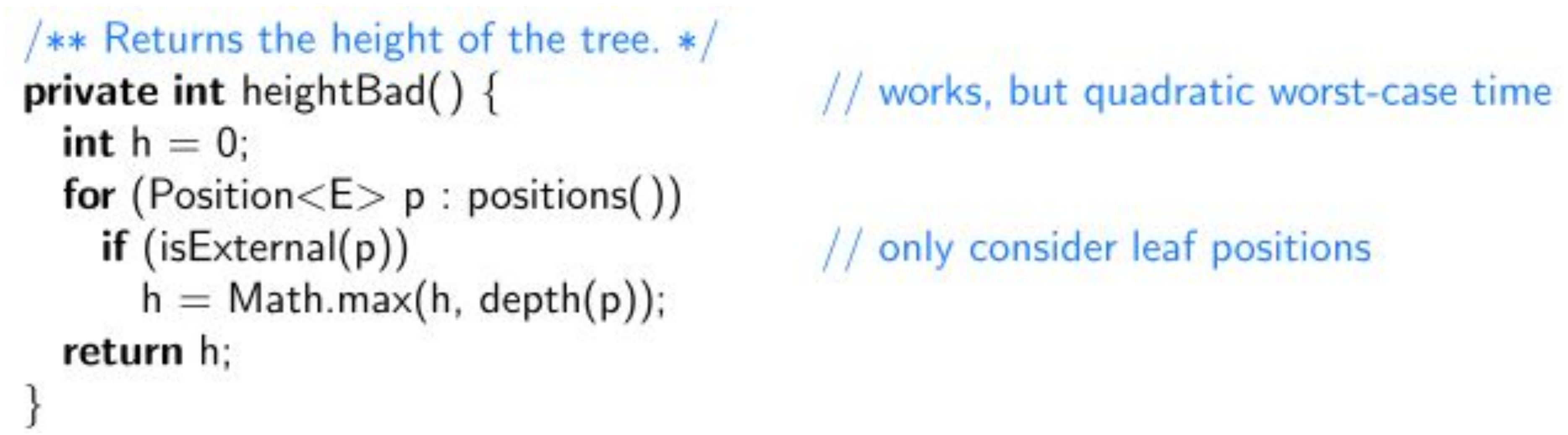
Computing Depth of element - Analysis

- The *running time of depth(p)* for position p is ***O(dp +1)***, where dp denotes the depth of p in the tree
- Algorithm depth(p) runs in ***O(n)*** *worst-case time*, where n is the total number of positions of T
    - a position of T may have depth n−1 if all nodes form a single branch



Computing Height of Tree

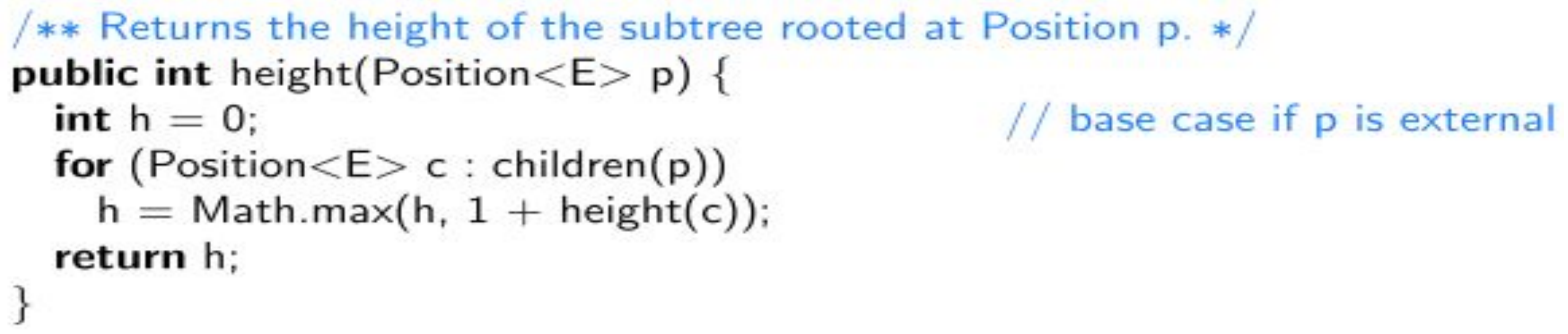
- The ***height*** of a tree to be equal to the *maximum of the depths of its positions* (or zero, if the tree is empty)
- It is easy to see that the position with maximum depth must be a leaf
- The heightBad entire iteration runs in O(n) time, where n is the number of positions of T
- The heightBad calls algorithm depth(p) on each leaf of T, its running time is $O(n+\sum_{P \in L}(dp + 1))$, where L is the set of leaf positions of T
- In the worst case, the sum $\sum_{P \in L}(dp + 1)$ is proportional to n^2
- Thus, algorithm heightBad runs in $O(n^2)$ worst-case time



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

Computing Height of Tree - Improved Version

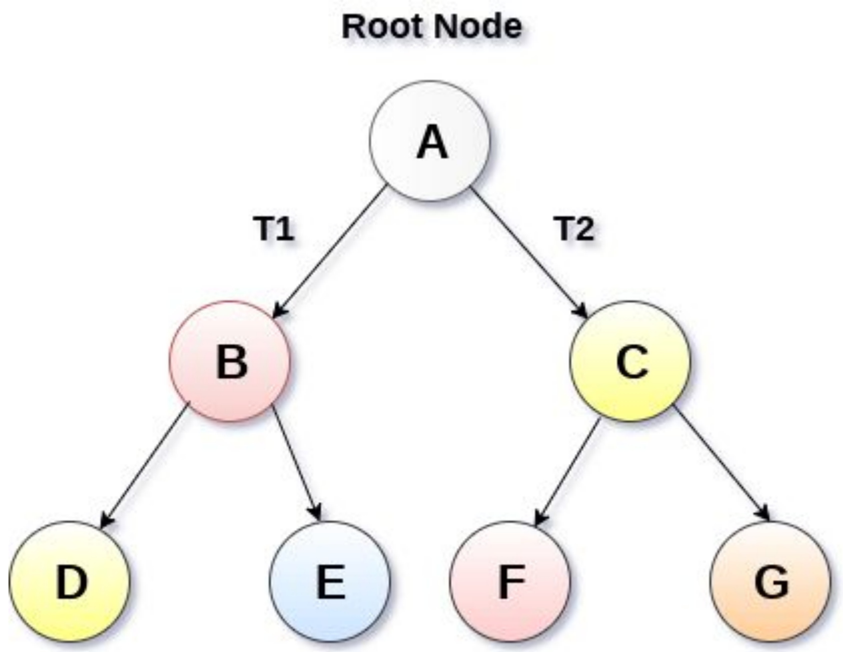
- We can compute the height of a tree more efficiently, in O(n) worst-case time, by considering a ***recursive definition***
- Formally, we define the height of a position p in a tree T as follows:
    - If p is a leaf, then the height of p is 0
    - Otherwise, the height of p is one more than the maximum of the heights of p's children
- The overall height of a nonempty tree can be computed by sending the root of the tree as a parameter
- Height algorithm iteration is executed in ***$O(c_p +1)$*** time, where c_p denotes the number of children of p
- Algorithm height(p) spends $O(c_p+1)$ time at each position p to compute the maximum, and its overall running time is ***$O(\sum_p(c_p +1))$*** =***$O(n+\sum_p c_p)$***



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

Binary Trees

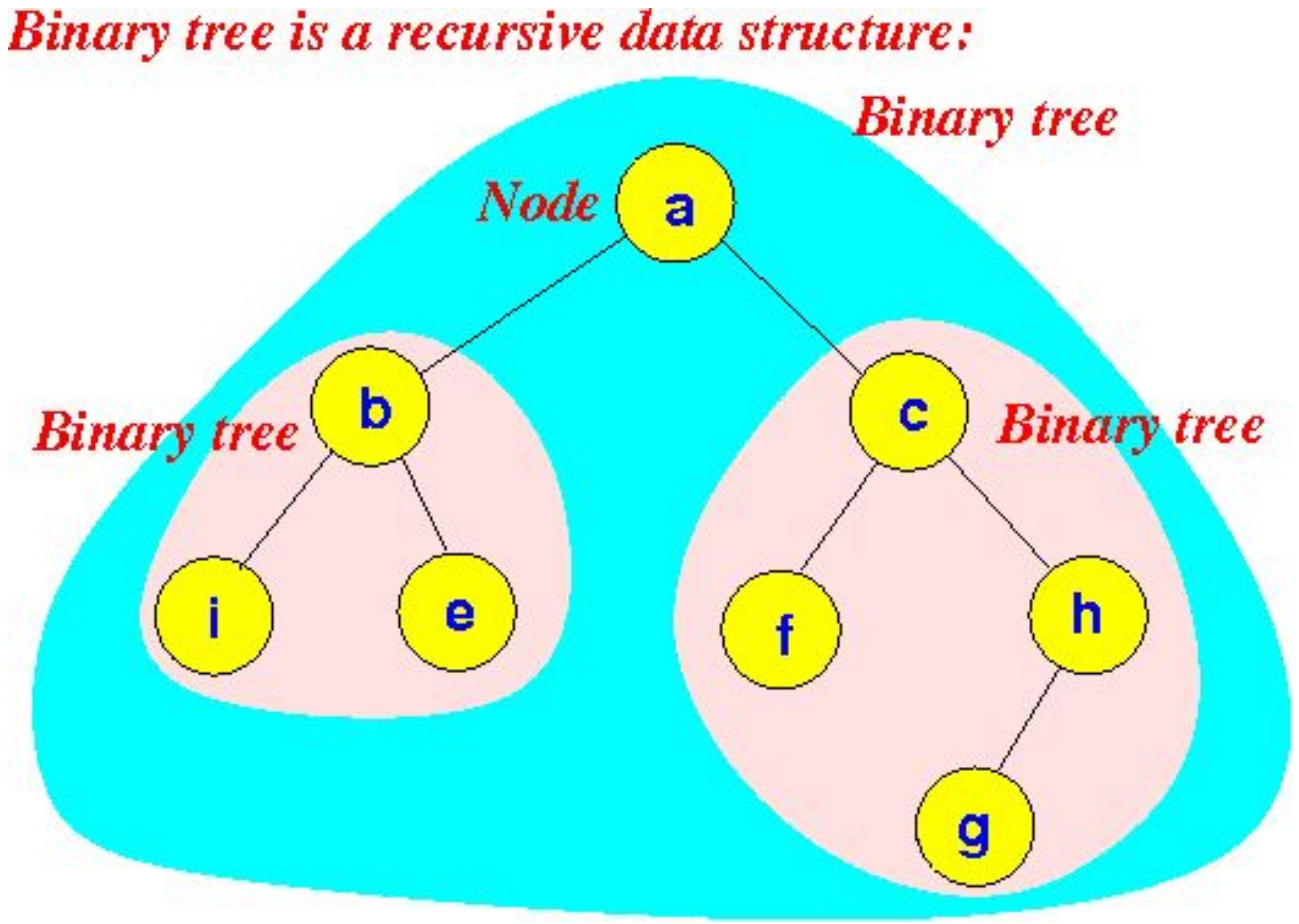
- A binary tree is an ordered tree with the following properties:
    1. Every node has at most two children
    2. Each child node is labeled as being either a left child or a right child
    3. A left child precedes a right child in the order of children of a node
- The subtree rooted at a left or right child of an internal node v is called a *left subtree* or *right subtree*, respectively, of v
- A binary tree is **proper** if *each node has either zero or two children* it is also called as **full binary trees**
- Thus, in a proper binary tree, every internal node has exactly two children



Binary Tree

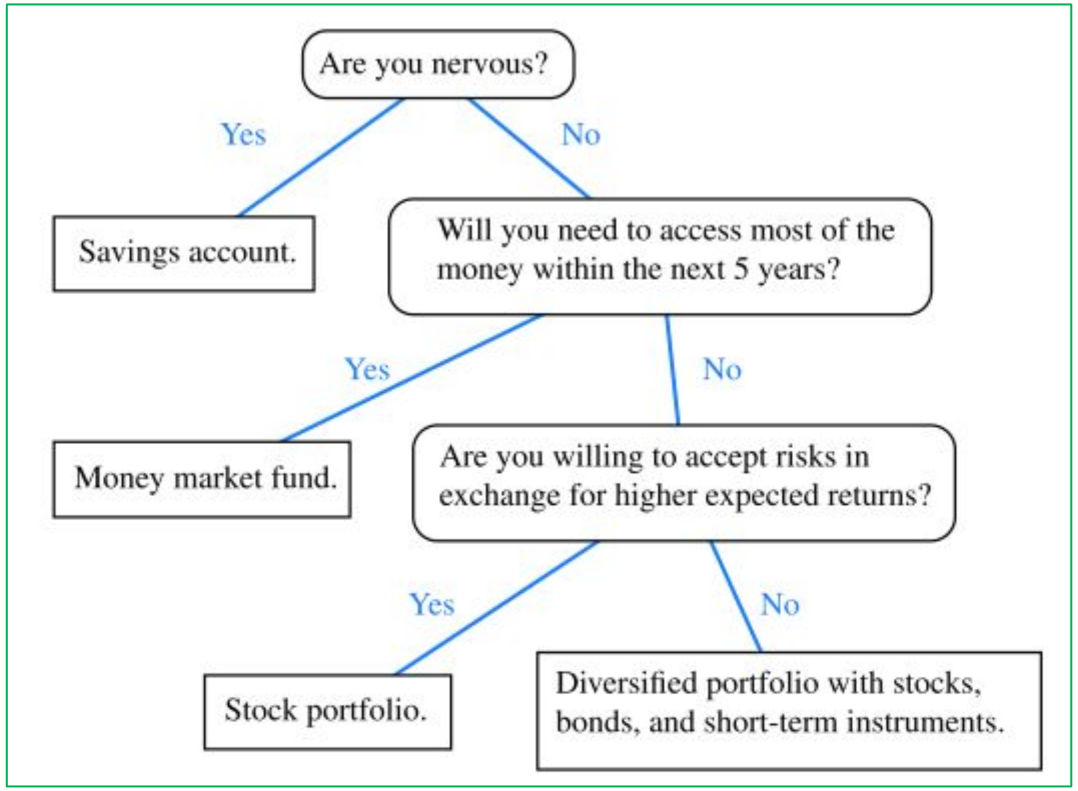
A Recursive Binary Tree Definition

- A binary tree is either:
  - An *empty tree*
  - A *nonempty tree* having a root node r, which stores an element, and *two binary trees that are respectively the left and right subtrees of r*.
- We note that one or both of those subtrees can be empty by this definition

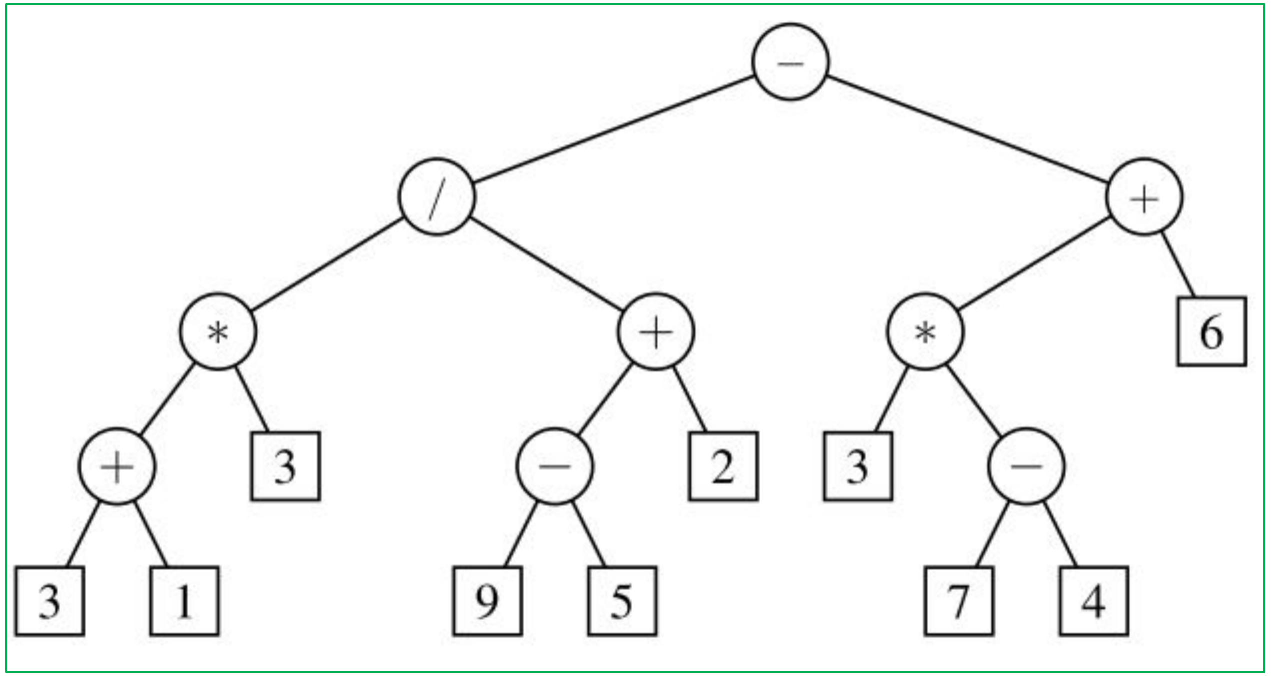


Binary Tree Applications

- Yes-or-No questions ***Decision Tree***
- A decision tree thatprovides recommendations to aprospective investor

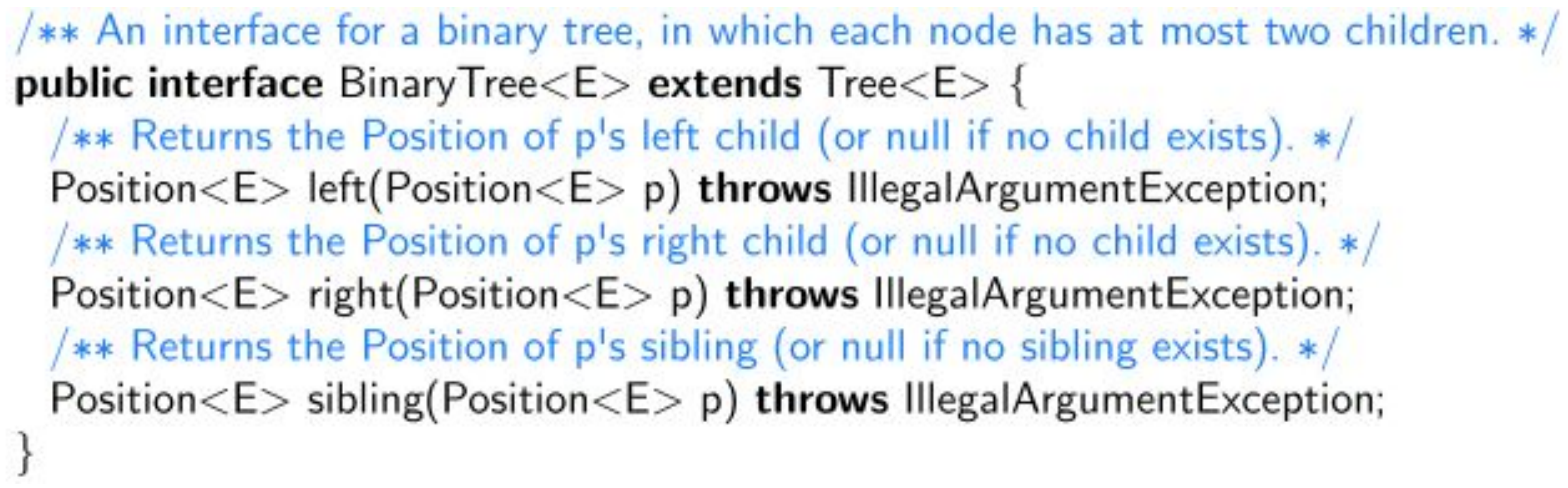

Are you nervous?
Yes
No
Savings account.
Will you need to access most of the money within the next 5 years?
Yes
No
Money market fund.
Are you willing to accept risks in exchange for higher expected returns?
Yes
No
Stock portfolio.
Diversified portfolio with stocks, bonds, and short-term instruments.

- An ***arithmetic expression*** represented by a binary tree
- Leaves are associated with variables or constants
- Internal nodes are associatedwithone ofthe operators +, −, ∗, and/
- The expression ((((3+1)∗3)/((9−5)+2))−((3∗(7−4))+6))



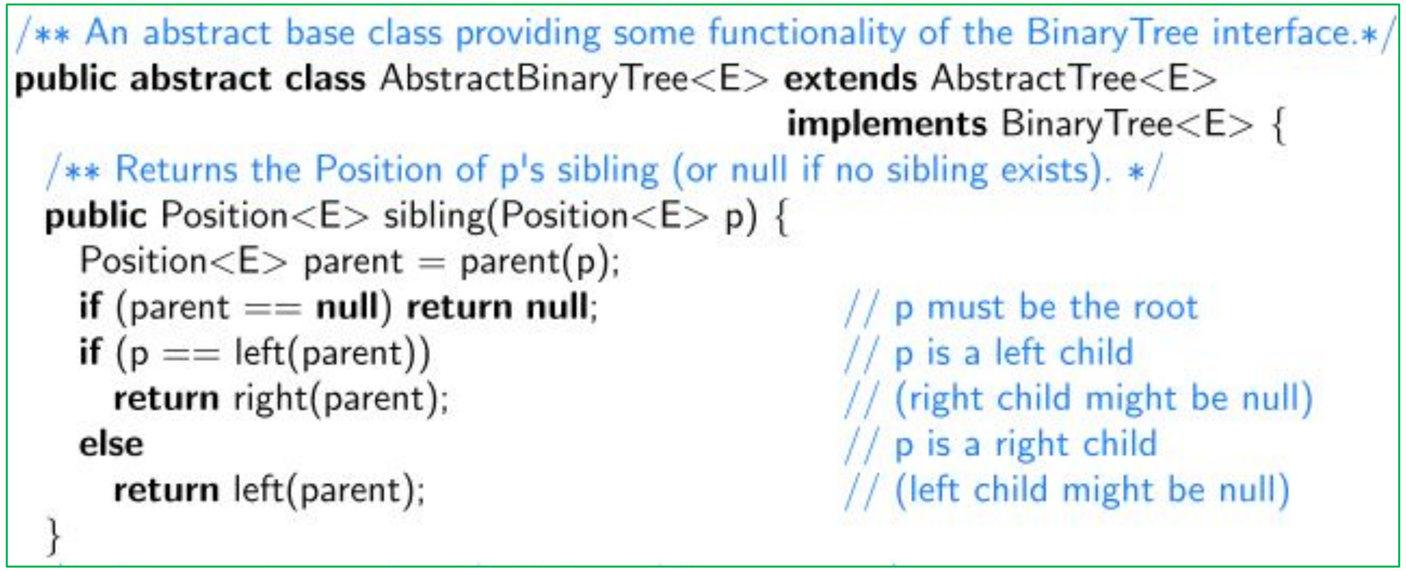
The Binary Tree Abstract Data Type

- As an abstract data type, a 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)
- This interface extends the Tree interface to add the three new behaviors
- In this way, a binary tree is expected to support all the functionality that was defined for general trees (e.g., root, isExternal, parent), and the new behaviors left, right, and sibling

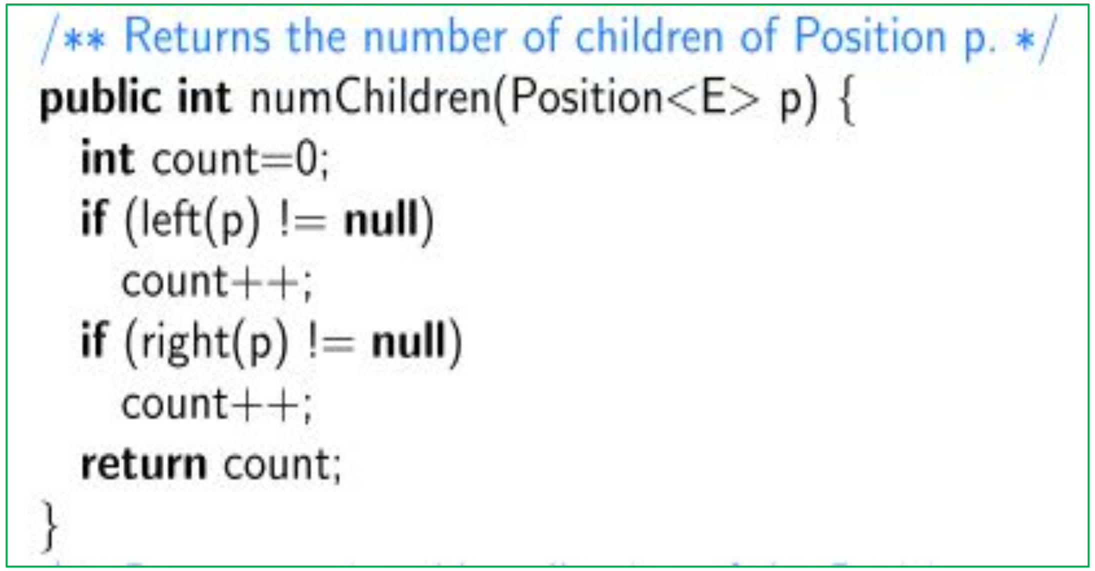


```java
/** An interface for a binary tree, in which each node has at most two children. */
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the Position of p's left child (or null if no child exists). */
  Position<E> left(Position<E> p) throws IllegalArgumentException;
  /** Returns the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

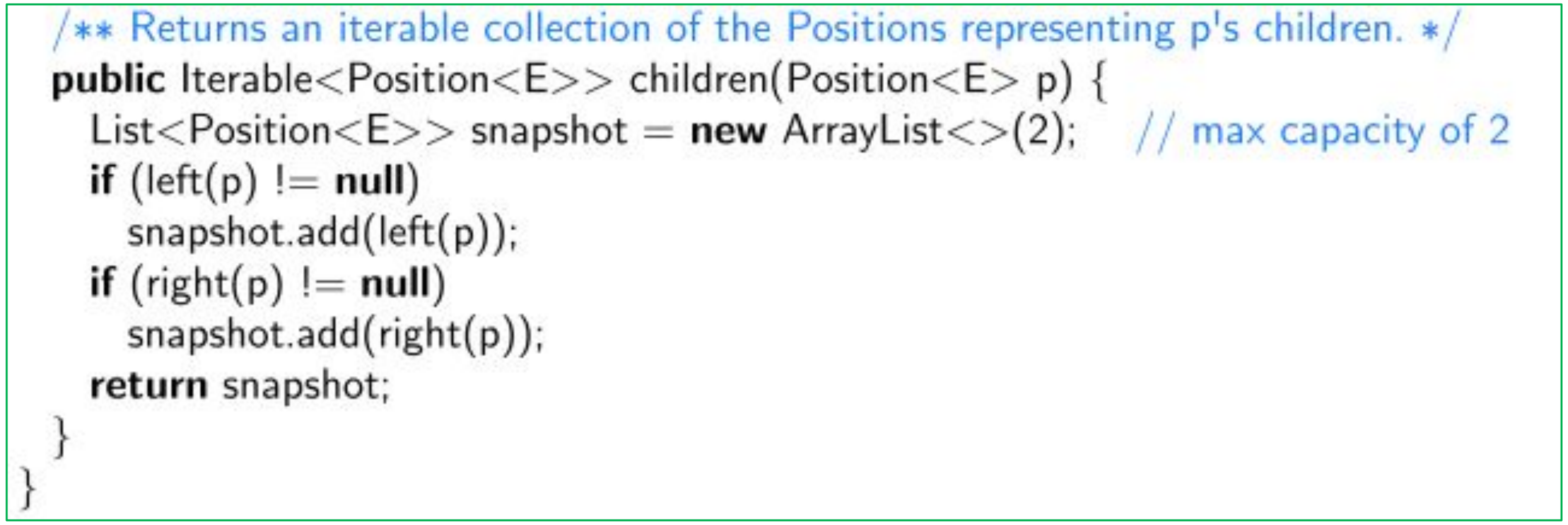
Defining an AbstractBinaryTree Base Class



```
/** An abstract base class providing some functionality of the BinaryTree interface.*/
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                                            // p is a right child
      return left(parent);                          // (left child might be null)
  }
```



```
/** Returns the number of children of Position p. */
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's children. */
  public Iterable<Position<E>> children(Position<E> p) {
    List<Position<E>> snapshot = new ArrayList<>(2);    // max capacity of 2
    if (left(p) != null)
      snapshot.add(left(p));
    if (right(p) != null)
      snapshot.add(right(p));
    return snapshot;
  }
}
```