A (rooted) tree has a distinguished node called the root, along with zero or more subtrees $T_1, \ldots, T_k$. 

$r$ is connected to each subtree root via a directed edge out of $r$. 
roots of subtrees are children of r
r is their parent
nodes with a common parent are siblings

A path from u to w is a sequence $u = v_1, v_2, \ldots, v_k = w$
where each node is a child of the previous one.

- $u$ is an ancestor of $v$ if $u \neq v$.
- $v$ is a descendant of $w$ (proper).

A leaf node has no children.

A path length is the number of edges.
The depth of node \( v \) is the length of the path from root to \( v \).

Height of \( v \) is length of largest path to a descendant leaf.

(Leaves have height 0.)

Height of tree = height of root.
Implement:

A node has an element and points to first child and next sibling.

```
class TreeNode {
    Object element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

```
root
```

```
B  C  D  E  F  G
|   |   |   |   |   |
A  H  I  J  K  L  M  N
|   |   |   |   |
G  P  Q
```
Traversals

Ex: filesystem - directories are nodes that can have children and their contents.

Listing files:
- print node name

For each member of directory:
- print name recursively
- explore contents
public void preorder(TreeNode node) {
    doSomething(node);
    TreeNode child = node.firstChild;
    while (child != null) {
        preorder(child);
        child = child.nextSibling;
    }
}

Called a preorder traversal.
**Postorder traversal:**

Recursively process each sub tree then the root.

```java
public void postorder(TreeNode node) {
    TreeNode child = node.firstChild;
    while (child != null) {
        postorder(child);
        child = child.nextSibling;
    }

    doSomething(node);
}
```
Both take $O(n)$ time if you spend $O(1)$ time per node.
Binary tree: each node has at most 2 children designated left & right child.

Expression tree:
Leaves are operands (constant or variable) of a mathematical expression.
Other nodes are operators to apply to whole expression to left or right.

\[(a + (b * c)) + ((d \#_e) + \#_g)\]

In order traversal:

Left subtree then root then right subtree.
Binary search tree:
Elements need some kind of comparison.
If root has element $x$, everything in left subtree $\leq x$ everything in right subtree $\geq x$
Search key: part of element used for comparisons.

Contains: See if \( x = \text{root element} \).

If less, search left.
If greater, search right.
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>> {
    public BinarySearchTree() {
        this.root = null;
    }

    public boolean contains(AnyType x) {
        return contains(x, this.root);
    }

    // ...

    private static class BinaryNode<AnyType> {
        // ...
    }

    private BinaryNode<AnyType> root;

    private boolean contains(AnyType x, BinaryNode<AnyType> node) {
        if (node == null) {
            return false;
        }

        int compareResult = x.compareTo(node.element);

        if (compareResult < 0) { // x < node.element
            return contains(x, node.left);
        } else if (compareResult > 0) { // x > node.element
            return contains(x, node.right);
        } else { // x = node.element
            return true;
        }
    }
}
I deal:

Can only do \( O(\log n) \) comparisons (height is \( \leq \log n \))

\( O(\log n) \) time contains
Worst case time of $\Theta(n \log n)$. 
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>> {
    // ...

    public AnyType findMin() {
        if (root == null) {
            throw new UnderflowException();
        }
        return findMind(root).element;
    }

    // ...

    private BinaryNode<AnyType> findMin(BinaryNode<AnyType> node) {
        if (node == null) {
            return null;
        } else if (node.left == null) {
            return node; // nothing less than v.element
        } else {
            return findMin(node.left);
        }
    }

    // ...
}
insert:

search for x. If not found, add a new node below where your search ended.
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>> {
    // ...

    public void insert(AnyType x) {
        root = insert(x, root);
    }

    // ...

    private BinaryNode<AnyType> insert(AnyType x, BinaryNode<AnyType> node) {
        if (node == null) {
            return new BinaryNode<AnyType>(x, null, null); // new leaf node
        }

        int compareResult = x.compareTo(node.element);
        if (compareResult < 0) { // x < node.element
            node.left = insert(x, node.left);
        } else if (compareResult > 0) { // x > node.element
            node.right = insert(x, node.right);
        } else { // x = node.element
            // Duplicate; do nothing.
        }

        return node;
    }

    // ...
}
Remove:
If element's node has no children, have parent link to null instead.
If element's node has one child, update parent to link directly to child.

remove(4)
Otherwise, \( vi \) node containing removed element

1) Find min. element \( w \) in right subtree of element were removing.

   Set \( v. \text{element} \leftarrow w \)

2) Remove node that originally contained \( w \).

\[
\begin{array}{c}
\text{2) Remove node that originally contained } w \text{. remove(2)}
\end{array}
\]
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>> {
    // ...

    public void remove(AnyType x) {
        root = remove(x, root);
    }
    // ...

    private BinaryNode<AnyType> remove(AnyType x, BinaryNode<AnyType> node) {
        if (node == null) {
            return node; // Item not found; do nothing.
        }
        int compareResult = x.compareTo(node.element);
        if (compareResult < 0) { // x < node.element
            node.left = remove(x, node.left);
        } else if (compareResult > 0) { // x > node.element
            node.right = remove(x, node.right);
        } else if (node.left != null && node.right != null) { // two children
            node.element = findMin(node.right).element;
            node.right = remove(node.element, node.right);
        } else if (node.left != null) {
            node = node.left;
        } else {
            node = node.right;
        }
        return node;
    }
    // ...
}