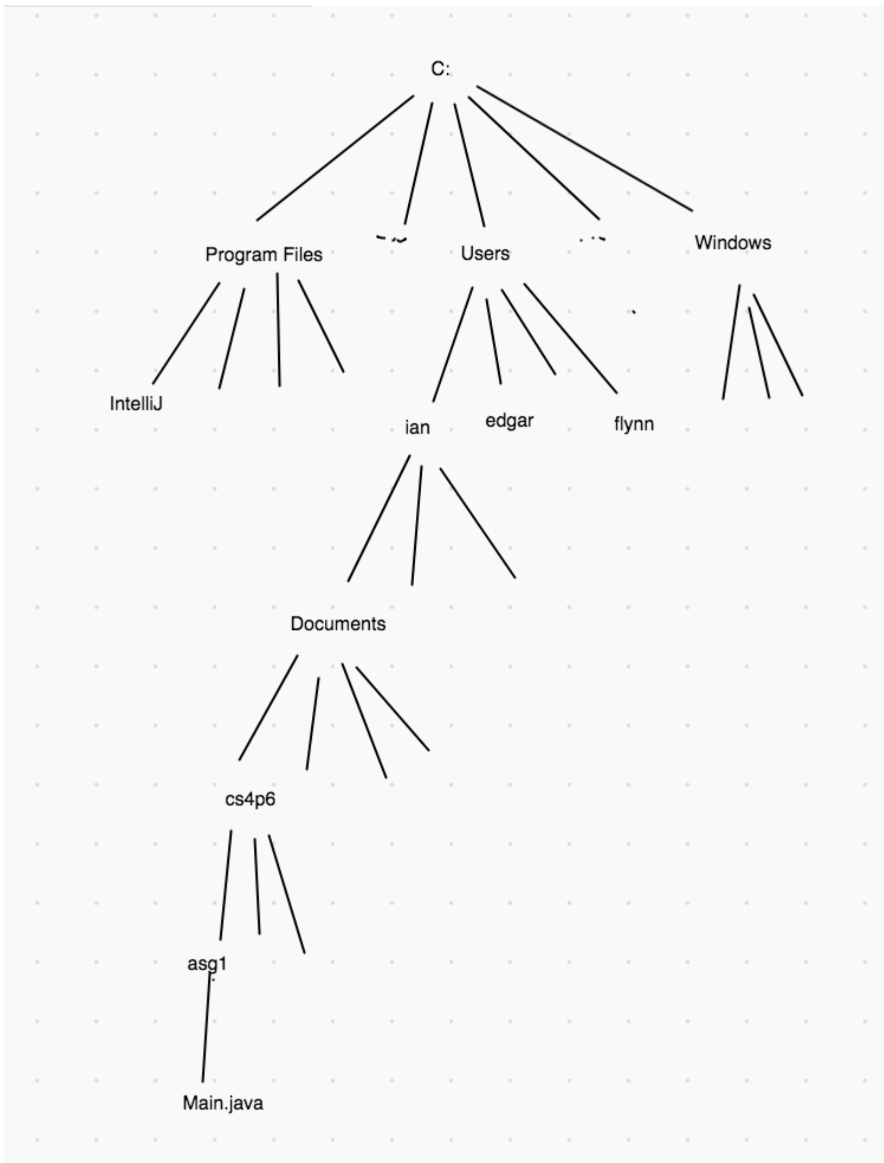
Trees

Common structure in Computer Science. Called tress because of their resemblance to actual trees!

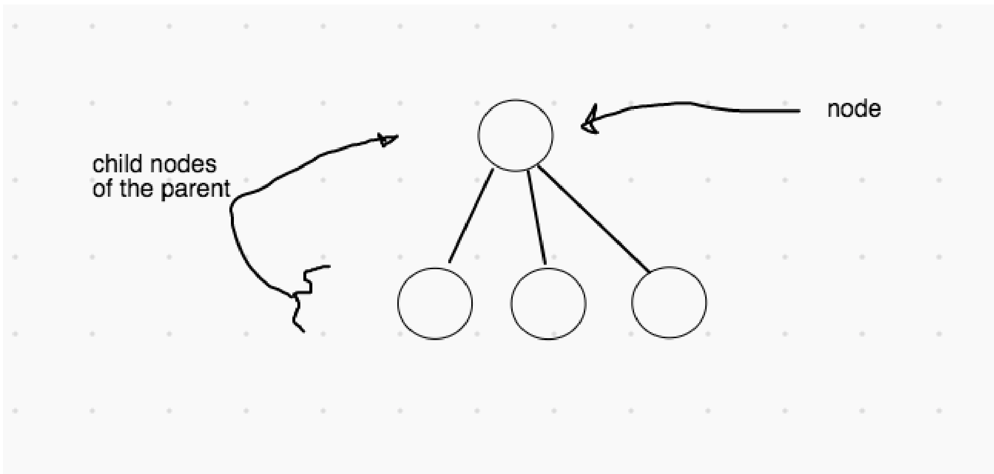
Example: Filesystems

In a filesystem, folders/directories store other folders and files, making a ”tree” structure:



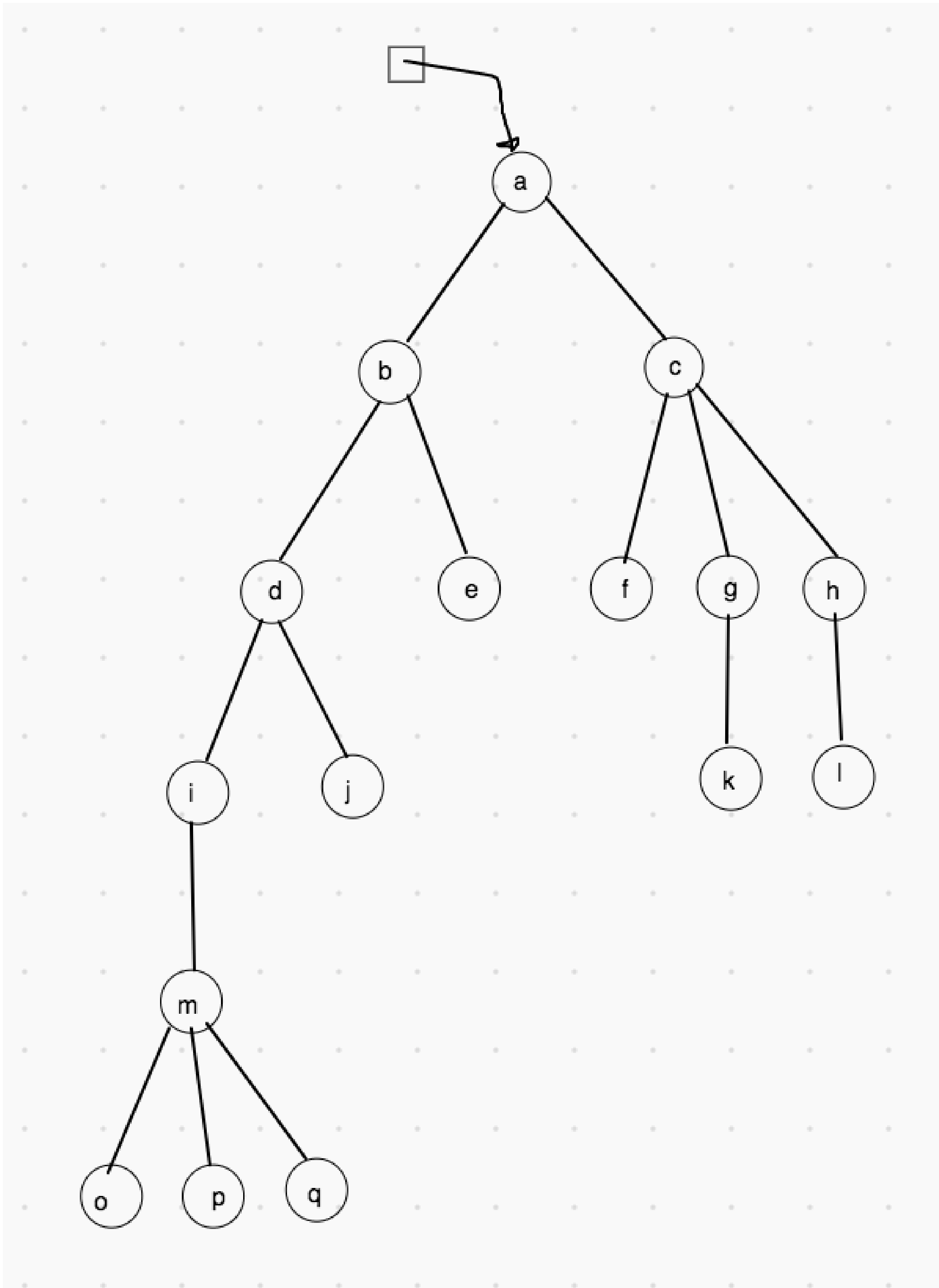
Other trees we've seen include: nested formats like HTML,XML and json, OOP inheritance, DFS/BFS search, etc…

Terminology



- A node is an element in the tree.
- A node can have children (connect by lines/edges).
- A node can have a single parent.

A tree is made up on nodes:



Family tree style terms

- The *child* nodes of c are {f, g, h}.

- The *parent* node of j is d.
- q has no children.
- a has no parent.
- The *descendants* of c are {f, g, h, k, l}.
- The *ancestors* of e are {b, a}, i.e.: just parent, parent of parent and so on.
- The descendants of a are {b, c, d, e, …, q}.
- The ancestors of q are {m,i,d,b,a}.
- The *siblings* of f are {g, h}.
- q has no descendants.

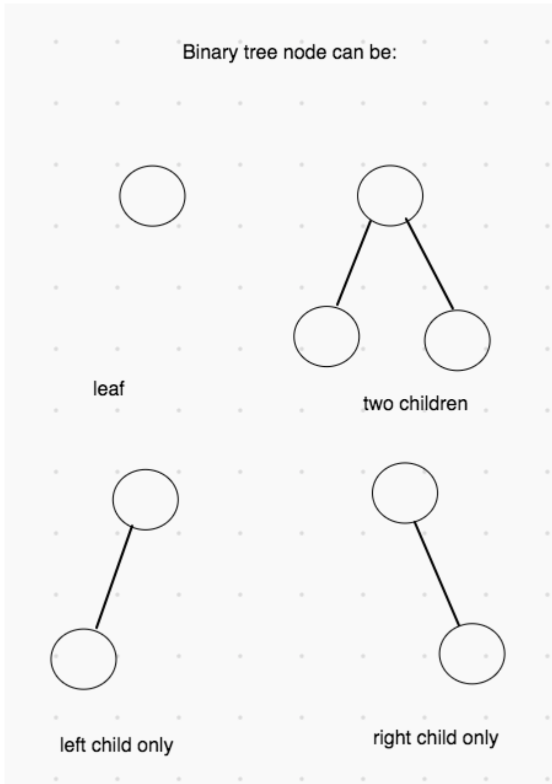
Nature style terms

- the *leaf* nodes are {o, p, q, k, l, e, f, j}.
- the root node is a.
- the internal nodes are {m, i, d, b, c, g, h, a}, non-leaves, so we include the root.
- the path from e to a: [e, b, a].
- the path from q to a: [q, m, i, d, b, a].
- the path from a to o: [a, b, d, i, m, o].
- we will exclude paths to non-ancestors, non-descendants.

Binary trees

Tree where each node has at most 2 children. The example isn't binary: m and c have 3 children.

There are 4 possible node configurations:

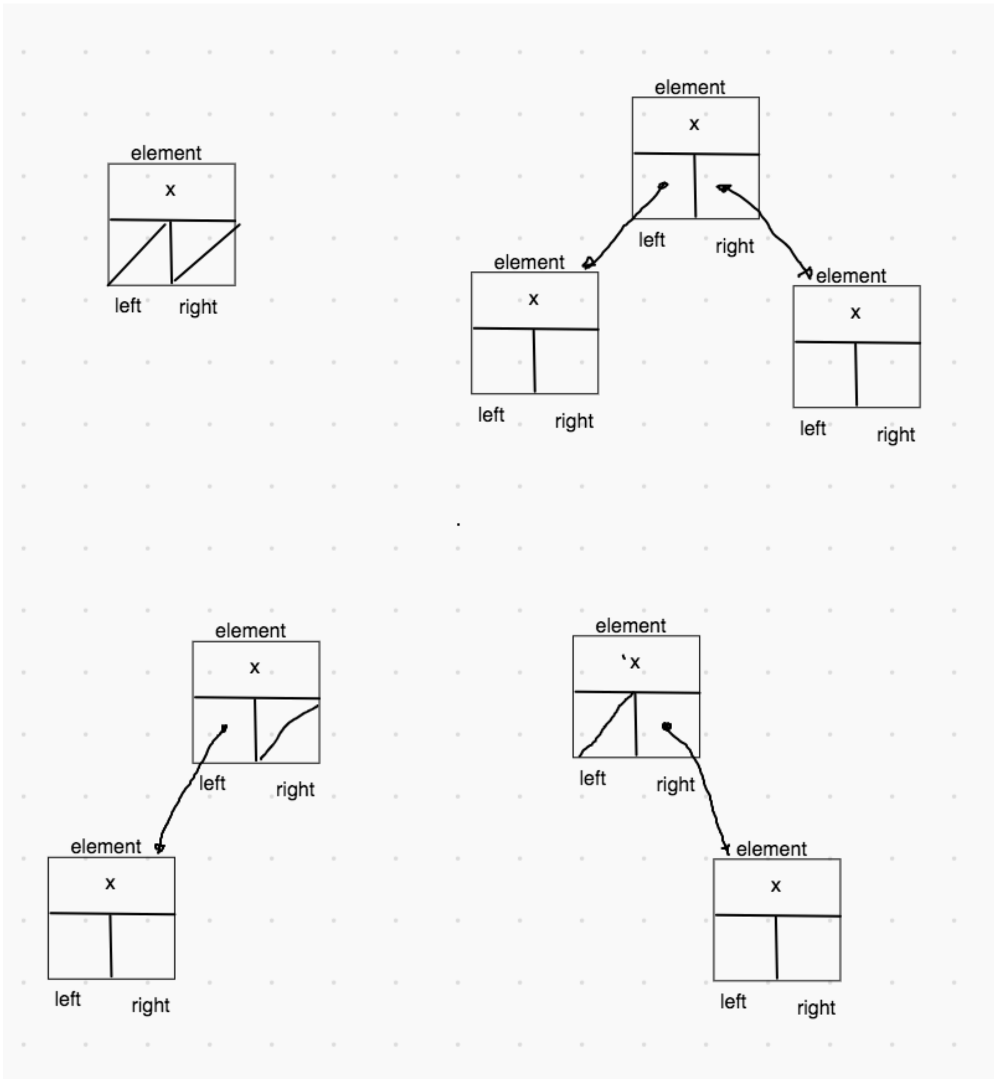


We will make a difference between having 1 child on the ”left” and 1 child on the ”right”.

In code, we can represent a node with this class:

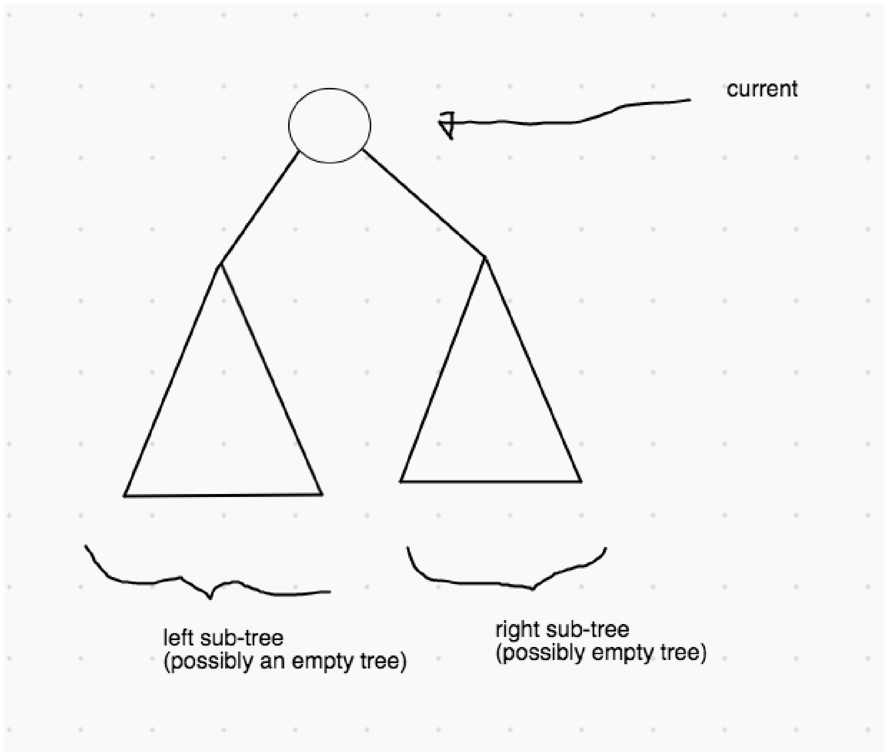
```
class Node<T> {
    public T element;
    public Node<T> left;
    public Node<T> right;
}
```

Drawing all the references, our 4 configurations look like this:



Binary tree property

Recursive algorithms on trees uses the fact the the left/right child of a node are roots to sub-trees:



Example: count the number of nodes in a binary tree!

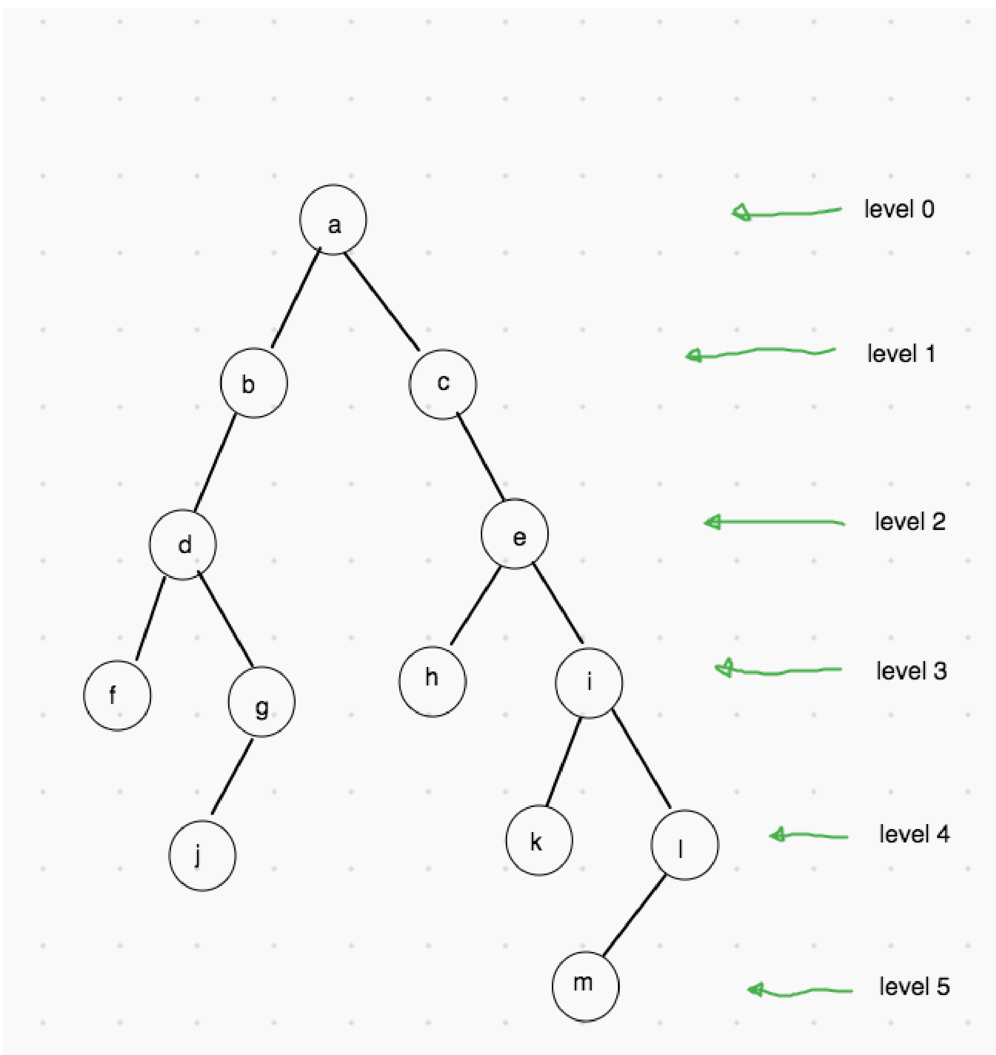
```
public int count(Node<T> current) {

      // empty tree
      if(current == null)
            return 0;

      // leaf node
      if(current.left == null && current.right == null)
            return 1;

      // internal node
      int c = 1;  // count the current node!
      if(current.left != null)
            c += count(current.left);
      if(current.right != null)
            c += count(current.right);
      return c;
}
```

We can simplify this to less lines of code:

```
public int count(Node<T> current) {
    if(current == null)
        return 0;
    return 1 + count(current.left) + count(current.right);
}
```

Exercise: write a method to compute the height of a tree

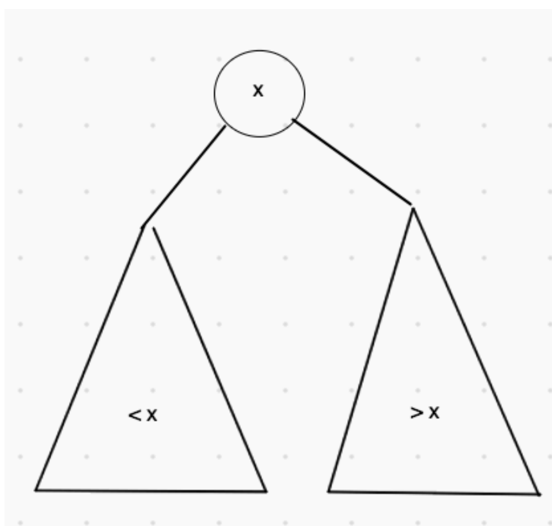
The *level* of a node is the length of the path from node to the root. The *height* is the maximum ”level” of the tree + 1.



```
public static int height(Node<T> current) {
    if (current == null)
       return 0;

    int leftHeight = height(current.left) + 1;
    int rightHeight = height(current.right) + 1;

    if (leftHeight > rightHeight)
       return leftHeight;
    else
       return rightHeight;
}
```

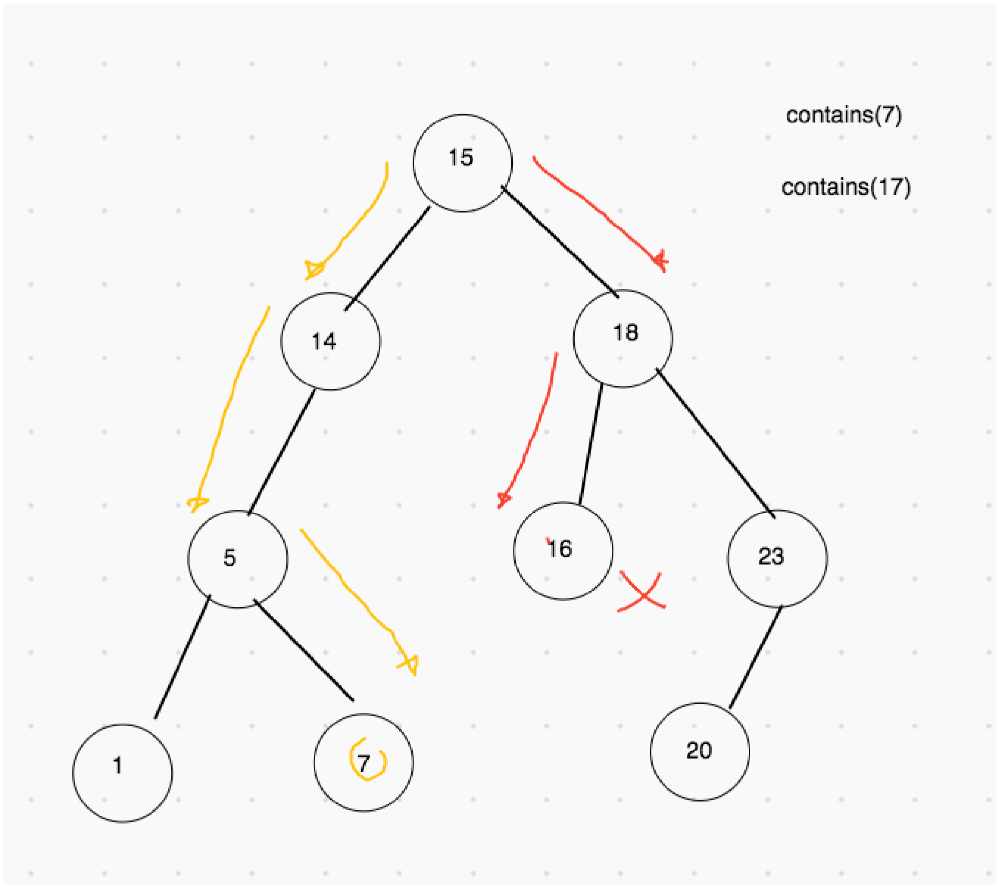
Binary Search Tree (BST)

A Binary Search Tree is a binary tree where each node statisfies the *binary search tree property*:



Exercise: implement a contains method for a BST

We can use the BST property to search efficiently for an element in a BST:




```
// precondition: current is the root of a BST
public <T extends Comparable<T>> boolean contains(Node<T> current, T value) {
    if (current == null)
        return false;
    int cmp = value.compareTo(current.value);
    if (cmp == 0)
        return true;
    if (cmp > 0)
        return contains(current.right, value)
    else
        return contains(current.left, value)
    }
```

Exercise: implement the `add(x)` method of the `TreeSet` class.

See video.

Binary Tree Traversals

We want to ”visit” each node in a binary tree. What we do when we visit depends on our algorithm.

Three main categories: preorder, postorder and inorder traversal.

Example: expression trees

double d = ((6 * 2) + (7 - 5)) / ((1 + 4) * 3);

Preorder traversal

”parent before children”

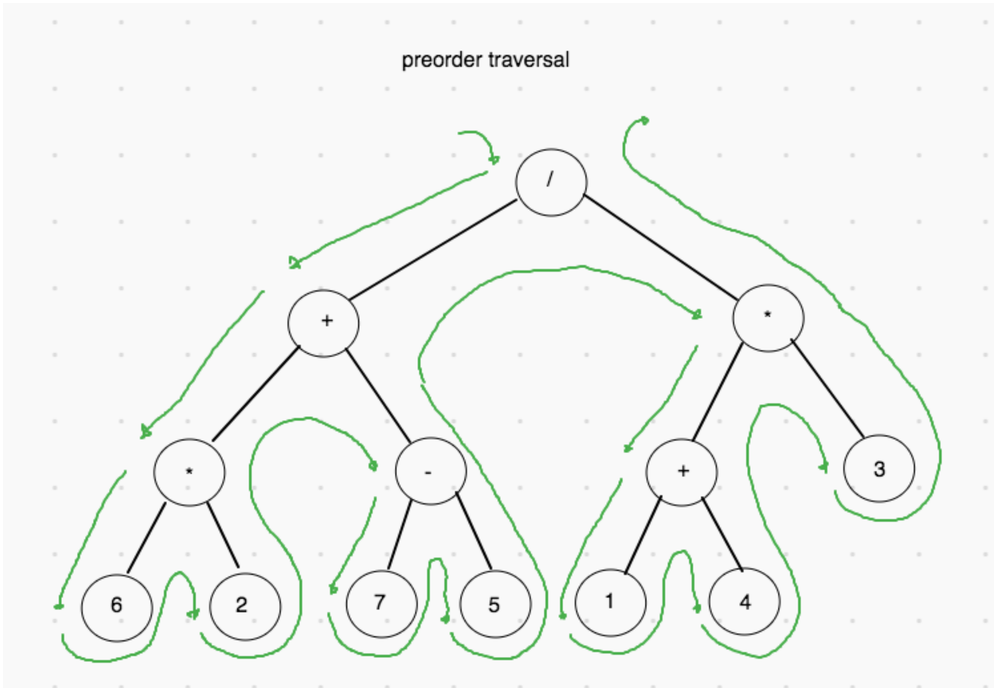
1. visit the current node.
2. preorder visit the left subtree.
3. preorder visit the right subtree.

Preorder print:

```
public static void preorderPrint(Node<T> current) {
     if(current == null) return;
     System.out.print(current.element + " ");
     preorderPrint(current.left);
     preorderPrint(current.right);
}
```

output on our example will be:

```
/ + * 6 2 - 7 5 * + 1 4 3
```



The output is the prefix form of the expression.
Idea: if each operation were a binary function, the prefix expression is like:
div(add(mult(6, 2), sub(7, 5)), mult(add(1,4), 3))

Postorder traversal

”children before parent”

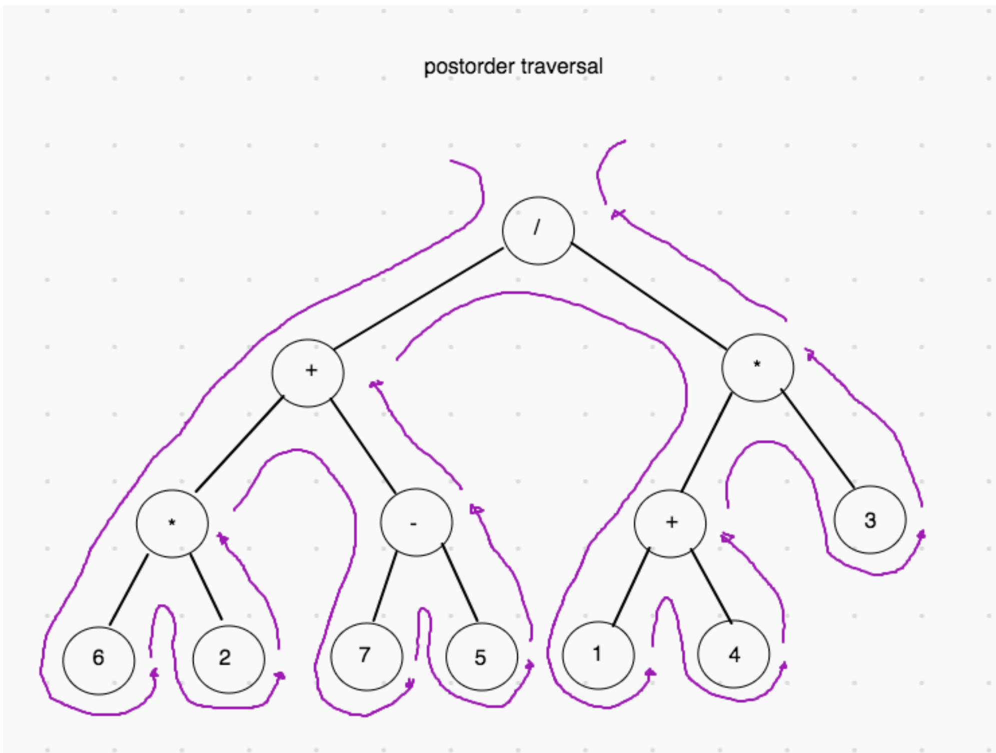
1. postorder visit the left subtree.
2. postorder visit the right subtree.
3. visit the current node.

Postorder print:

```
public static void postorderPrint(Node<T> current) {
     if(current == null) return;
     postorderPrint(current.left);
     postorderPrint(current.right);
     System.out.print(current.element + " ");
}
```

output on our example will be:

```
6 2 * 7 5 - + 1 4 + 3 * /
```



The output is the postfix form of the expression.

This resembles how a computer would evaluate the expression.

Inorder traversal

"left child, parent, right child"

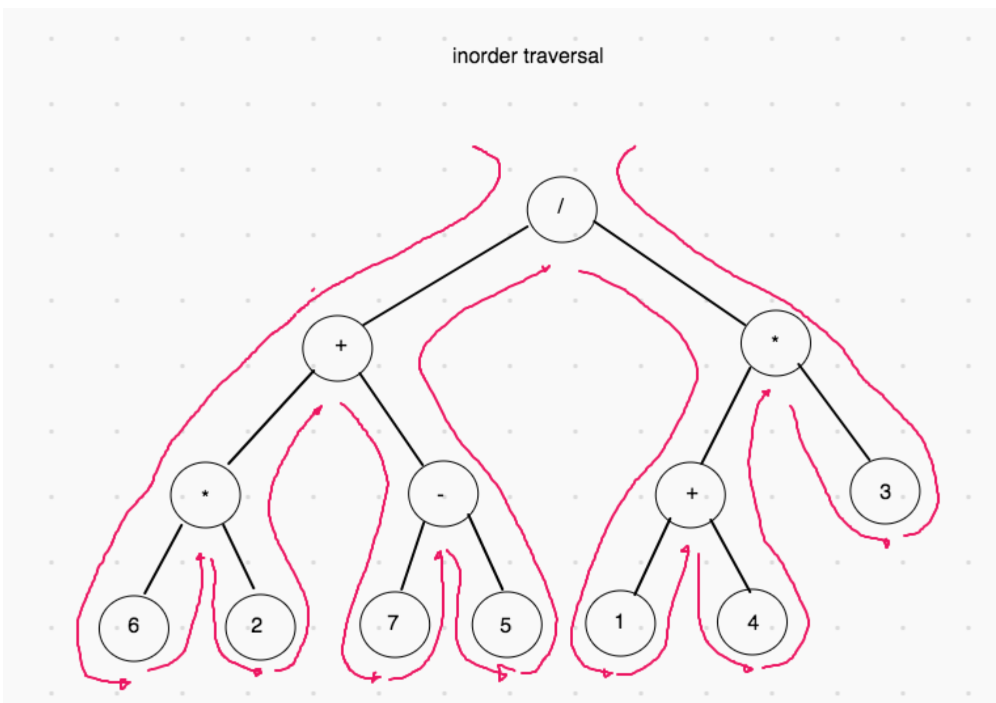
1. inorder visit the left subtree.
2. visit the current node.
3. inorder visit the right subtree.

Inorder print:

```
public static void inorderPrint(Node<T> current) {
     if(current == null) return;
     System.out.print("(");
     inorderPrint(current.left);
     System.out.print(current.element + " ");
     inorderPrint(current.right);
     System.out.print(")");
}
```

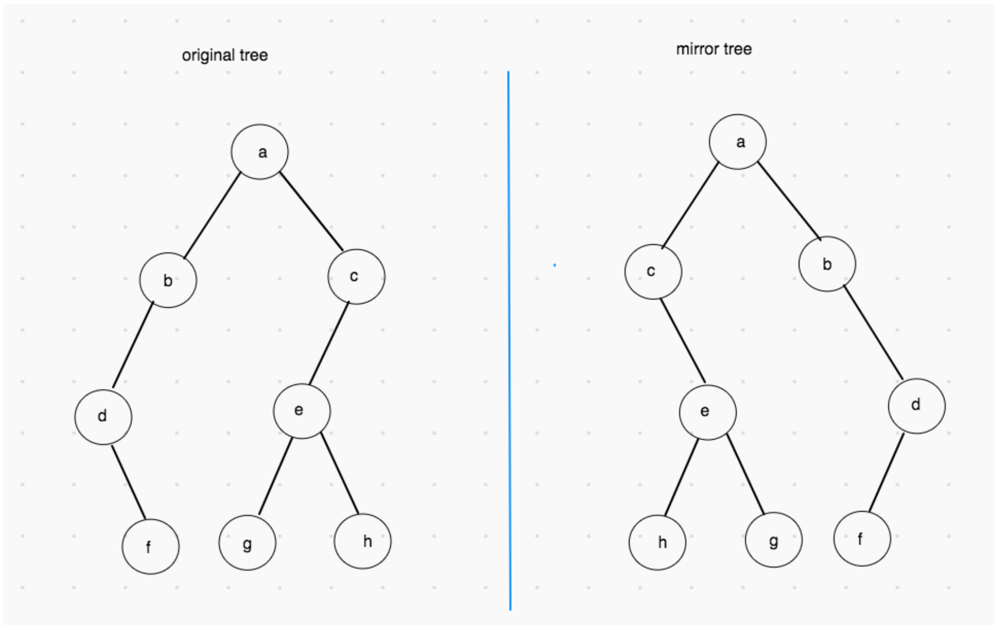
output on our example will be:

```
6 * 2 + 7 - 5 / 1 + 4 * 3
((((6) * (2)) + ((7) - (5))) / (((1) + (4)) * (3)))
```



The output is the infix form of the expression without parentheses.

Exercise: modify a binary to be the mirror image of itself




```
// preorder algorithm
public static void mirror(Node<T> current) {
     if(current == null) return;

     // "visit" is swap children
     Node<T> tmp = current.left;
     current.left = current.right;
     current.right = tmp;

     mirror(current.left);
     mirror(current.right);
}

// postorder algorithm
public static void mirror(Node<T> current) {
     if(current == null) return;

     mirror(current.left);
     mirror(current.right);

     // "visit" is swap children
     Node<T> tmp = current.left;
```

```
    current.left = current.right;
    current.right = tmp;
}


// inorder algorithm: broken it will leave the tree as it was (not a mirror)
public static void mirror(Node<T> current) {
    if(current == null) return;

    mirror(current.left);

    // "visit" is swap children
    Node<T> tmp = current.left;
    current.left = current.right;
    current.right = tmp;

    mirror(current.right);  // undoes the mirroring.
}
```

Exercise: create a mirror image of a binary tree

```
// postorder algorithm where "visit" creates a mirrorred node.
public static Node<T> createMirror(Note<T> current) {
    if(current == null) return null;

    Node<T> mirrorLeft = createMirror(current.left);
    Node<T> mirrorRight = createMirror(current.right);

    Node<T> tmp = new Node<>(current.element);
    tmp.left = mirrorRight;
    tmp.right = mirrorLeft;
    return tmp;
}
```