**Representation of trees by binary trees**

Binary trees are worth studying in some detail because we can obtain a correspondence between trees and binary trees, enabling us to use binary tree operations to handle *any* tree. Recall the differences:

<table>
<thead>
<tr>
<th>TREE</th>
<th>BINARY TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never empty.</td>
<td>Can be empty.</td>
</tr>
<tr>
<td>A node may have 0, 1, 2, 3, ... children.</td>
<td>A node may have only 0, 1 or 2 children.</td>
</tr>
<tr>
<td>A single child is simply</td>
<td>A single child may be</td>
</tr>
<tr>
<td>“the (only) child”.</td>
<td>“the left child” or “the right child”.</td>
</tr>
</tbody>
</table>

Take any tree; link all the children of a node together and remove all links from parent to child except that to the leftmost child. For example, the tree \((A(B(E, F, G), C, D(H, I)))\) has the following graphical representations as a tree and a binary tree, respectively.

![Tree and Binary Tree Diagrams](image)

Clearly the root of the binary tree will never have a non-empty right subtree; we may conveniently use the list head as the root, so that `HEAD.INFO` will now contain the data for the root, our previous conventions for the list head remaining unchanged. (This will necessitate minor changes in the traversal algorithms given earlier.)

Notice that if we *right*-thread the above binary tree, the structure we obtain may be depicted in either of the following forms:

![Right-Threaded Binary Tree Diagrams](image)

We now have a “cycle” passing from a node through its children and back to the parent, i.e., `LLINK` points to the first child, and `RLINK` points to the next sibling or, in the case of the last child, back to the parent.
The above transformation of a tree into a binary tree is a special case of the general rule for the natural correspondence between forests and binary trees. As an example (from Knuth), consider the forest \((A(B, C(K)), D(E(H), F(J), G))\):

The order of visiting the nodes if we traverse the binary tree is

- pre-order: \(ABCKDEHFJG\),
- in-order: \(BKCAHEJFGD\).

These correspond to tree (or forest) traversal orders, viz. tree pre-order and tree post-order, respectively. Notice the relationship of these traversal orders to the parenthesized notation for the forest.

The following example (from Knuth again) shows the usefulness of the binary tree transformation. The tree below and the right-threaded binary tree to which it can be converted both represent the algebraic formula

\[3 \times \sqrt{x + 1} - a/(2 \times x)\]

Pre-order traversal yields the nodes of the binary tree (excluding HEAD) as

\(- * 3 \times sqrt + x 1 / a * 2 x\)

which is the formula in Polish prefix notation.

In-order traversal yields the ordering

\[3 x 1 + sqrt * a 2 x * / -\]

which is in Polish postfix (or Reverse Polish) notation.
Now assume that the INFO field of each node contains:

(a) a DATA field holding the symbol (*, sqrt, 2, etc.),

(b) a TYPE field indicating whether the DATA field represents an operator (e.g., +), a constant (e.g., 3) or a variable (e.g., x),

(c) an OPS field, relevant only for an operator, showing the number of operands for that operator (e.g., sqrt takes one, / takes two).

The following algorithm will evaluate the formula, using an auxiliary stack B. (The APPLY function yields the result of applying an operator to its operand(s), and the VAL function obtains the value of a variable.)

```java
enum {operator, constant, variable};
Q ← insucc(&HEAD); // insuccU(&HEAD) if tree is unthreaded
while (Q ≠ &HEAD)
{
    switch (Q↑TYPE)
    {
        case variable:
            STACK(B, VAL(Q↑DATA));
            break;
        case constant:
            STACK(B, Q↑DATA);
            break;
        case operator:
            R ← UNSTACK(B);
            if (Q↑OPS = 2)
                STACK(B, APPLY(Q↑DATA, UNSTACK(B), R));
            else
                STACK(B, APPLY(Q↑DATA, R));
            break;
        default:
            throw new InvalidTypeException(Q↑TYPE);
    }
    Q ← insucc(Q); // insuccU(nil) if tree is unthreaded
}
return UNSTACK(B);
```