Eliminating Recursion from Inorder Traversal

As an example of the problem of writing non-recursive programs we give 3 versions of non-recursive programs for Inorder Traversal. A recursive version was given before as

```
Inorder(t : BIN_NODE[G]) is
    do
        if t /= void
            Inorder(t.left)
            "Process (t.value)"
            Inorder(t.right)
        end
    end -- Inorder
```

The non-recursive versions are less readable, more complicated but may be more efficient in both time and space.

Also some languages (e.g. Assembler, Fortran, Occam) don't support recursion and so the non-recursive versions can be written in these languages.

Version 1. Using an Explicit Stack

In this version the recursion is made explicit by using a Stack. It is not essentially more efficient as recursion uses an implicit stack.

Version 2.   Threaded Trees

A Threaded Trees are used to implement a binary tree. This program has not the overhead of a stack but uses extra storage (a boolean-bit per node). A Threaded Tree takes advantage of the void links on the leaf nodes so as to have 'threaded links' to the inorder successor. An extra 'flag' attribute is needed per node to distinguish between a 'thread' and a 'link'. This version "trades space for time".

Version 3.   Morris Inorder

The 3rd is an elegant version by Joe Morris, ex-TCD / UCD, who is now at Dublin City University. This version, like Threaded trees, takes advantage of the empty links from the leaf nodes but it does not use an extra flag bit. It has the efficiency of Threaded trees without the extra space. Joe Morris has also proved this program correct using the program verification techniques of Hoare / Dijkstra.

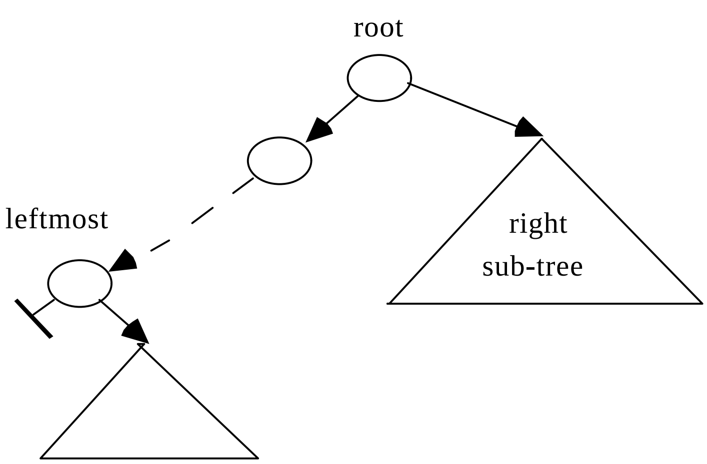
Version 1. Inorder Traversal Using an Explicit Stack

```
Non_Rec_Inorder(t:BIN_NODE[STRING]) is
    local
        stk : STACK[BIN_NODE[STRING]]
        it : BIN_NODE[STRING]
    do
        !!stk.make
        from
            it := t
        until
            it = void and stk.empty
        loop
            from
            until
                it = void
            loop
                stk.add(it)
                it := it.left
            end
            it := stk.item
            stk.remove
            io.put_string(it.value)
            io.put_string("  ")    -- process node
            it := it.right
        end
    end -- Non_Rec_Inorder
```

Strategy of this program:

In Inorder traversal, the 'first' node is the left-most node. The program finds the first node, while stacking all the items on the path to the leftmost node. The leftmost node is also stacked but then immediately removed (and processed). We then move to the right node (if any) of this leftmost node and this node is now the root of a (sub)tree.

We repeat the inorder traversal on this (sub)tree. The whole program halts when the stack is empty.



We can test this program in the context of Binary Search Trees by creating a BST and use Inorder to output the nodes. The nodes are printed in alphabetical order, i.e. the nodes will be sorted.

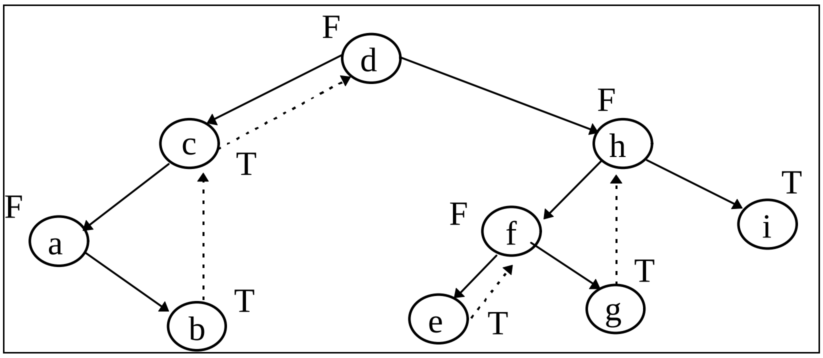
```
class     INORDER_TEST
creation
          make
feature

     make is
          local
               bt : BST[STRING]
          do
               io.put_string("%NEnter word, -quit- to end:")
               !!bt
               from
                    io.read_string
               until
                    equal(io.last_string, "quit")
               loop
                    bt.add(io.last_string)
                    io.put_string("%NEnter another word : ")
                    io.read_string
               end
               Print_Tree(bt.root,2)
               io.new_line
               Non_Rec_Inorder(bt.root)
          end -- make

     Non_Rec_Inorder(t:BIN_NODE[STRING]) is
          As above

     Print_Tree(t:BIN_NODE[STRING]; Indent:INTEGER) is
          do
               if t /= void then
                    Print_Tree(t.right, Indent+4)
                    io.put_string(Spaces(Indent))
                    io.put_string(t.value)
                    io.new_line
                    Print_Tree(t.left, Indent+4)
               end
          end -- Print_Tree
etc.
```

Version 2. Threaded Trees

Since the context is Inorder Traversal, we consider Right Threaded Trees.

Each node object in a Binary Tree has 2 link attributes, left and right, therefore, in a N node tree we have 2N links, of which only N-1 are used (the root has no link to it; root has in-degree 0). The other N+1 links are unused.

In a Threaded tree we make the right link of each leaf node reference (point to) the inorder successor. These links must be distinguished form the original 'tree links'.



F indicates that the right link is <u>not</u> a Thread

T indicates that the right link is a Thread

We introduce a new class THREAD_TREE implemented via a THREAD_NODE, an adapted version of BIN_NODE

```
class THREAD_NODE[G]
    feature
        left, right : THREAD_NODE[G]
        value : G
        rthread : Boolean
    etc. -- operations for settings the attributes.
    end
```

Inorder Traversal on a Threaded Tree.

Assuming a threaded tree, as in the e.g. above, we can write a non-recursive version of Inorder Traversal.

Each node in the threaded tree will have a right link. If the right link is a thread then it will refer to the inorder successor. If a node has a proper (original) right link, then we need to find its inorder successor.

```
next(tn : THEAD_NODE[G]) : THREAD_NODE[G] is -- the inorder succesor of tn
    require
            tn /= void
    local
        s : THREAD_NODE[G]
    do
        s := tn.right
        if not tn.rthread then
        -- if right link is proper  find leftmost of right 'subtree'
            from
            until
                s.left = void
            loop
                s := s.left
            end
        end
        result := s
    end -- next
```

<u>Comments</u>:
The function 'next' finds the inorder successor of a node in a threaded tree. Consider the following cases:

- **Node, tn, is an internal node with a proper right link, i.e. rthread is false. The reference, s, initially goes right and when the right link is not void it finds the leftmost of the right subtree of tn.**
- **Node, tn, is a leaf, i.e. its right link is a thread and so rthread is true. Since rthread is true, tn.right is the successor of tn.**
- **Node, tn, it the rightmost of the whole tree. In a threaded tree the rightmost node has no successor and its right link is void. For convenience its rthread flag is set to true. In this case the successor is tn.right which is void. If the node has no successor the function next returns void.**

Inorder Traversal

The function, next, has done most of the job. To implement Inorder we find the 'first' node, the leftmost of the whole tree. Starting with the leftmost node we traverse through the threaded tree using the function 'next'.

```
Inorder (t : THREAD_NODE[G]) is
    local
        p : THREAD_NODE[G]
    do  -- Find leftmost node
        from
            p := t
        until
            p.left = void
        loop
            p := p.left
        end  -- p is at start
        --traverse through tree via the next function
        from
        until
            p = void
        loop
            "process node p"
            p := next(p)
        end
    end -- Inorder
```

Building a Threaded Tree

Given threaded trees, L and R and a root value, v.

```
build(v:G; L,R : THREAD_NODE[G]): THREAD_NODE[G] is
    local
        p : THREAD_NODE[G]
    do
        !!result
        result.value_set(v)
        if L /= void then
            from -- find rightmost of L
                p := L
            until
                p.right = void
            loop
                p := p.right
            end
            p.right_set(result) -- right thread to new root node
             -- p.rthread_set(true)  -- already set to true
            result.left_set(L)
        end
        -- link in right subtree
        if R /= void then
            result.right_set(R)
            -- result.rthread_set(false)  -- set by default by Eiffel
        else
            result.rthread_set(true)  -- set rightmost node to true
    end
end -- build
```

Comment:

In building a threaded tree, we don't have the property that if t is a threaded tree then so is t.left and t.right.
This property is useful for designing recursive programs.

In a threaded tree, t.right is a threaded tree, but t.left is not. In building the full threaded tree we change the right most link of the original t.left.

Converting a Binary Tree to a Threaded Tree

A Binary tree is implemented via BIN_NODE and a threaded tree via THREAD_NODE. We take advantage of the recursive structure of Binary Trees to convert them to Threaded trees.

```
bin2thread(b:BIN_NODE[G]):THREADND[G] is
	local
		l, r : THREADND[G]
	do
		if b /= void then
			l := bin2thread(b.left)
			r := bin2thread(b.right)
			result := build(b.value, l, r)
		end
	end -- bin2thread
```

Comment:

The left and right threaded trees are recursively constructed and then 'build' is used to construct a full threaded tree out of these.