Morris Inorder

We first consider an ‘abstract-code’ version of the algorithm that Joe Morris uses.

Notation:

- We use ’void’ for the empty tree and if v is an item and L and R are tree then build(v, L, R) gives a tree with left subtree L and right subtree R and root value v. If t is a non-empty tree then

  t = build(t.value, t.left, t.right).

  For convenience, for ‘processing’ a node we add its value to a List.

- Let us have an operation, ‘concatenation’ denoted by infix ++ so that if s and t are list then s ++ t is the concatenation of s and t. If x is an item then [x] is the list containing just x. So to ‘add’ an item x to a list s we use [x] ++ s. The empty list is denoted by []

Inorder Traversal

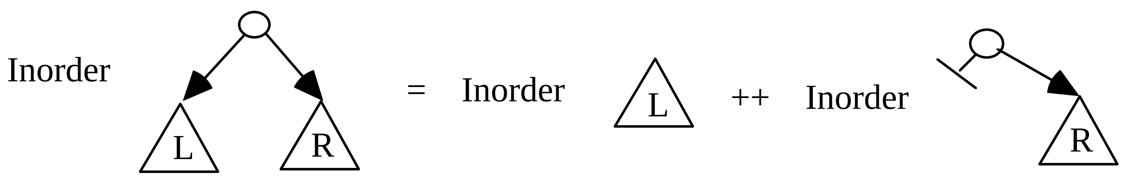
Using our list notation we re-write the routine Inorder. Let us abbreviate BIN_NODE[G] to TREE[G], with the ‘benign’ ambiguity of regarding a node as a tree.

```
Inorder (t : TREE[G]) : LIST[G] is
    do
        if t = void then
            result := [] -- empty list
        else
            result := Inorder(t.left)  ++ [t.value] ++ Inorder(t.right)
    end -- Inorder
```

For non-empty t, we get,

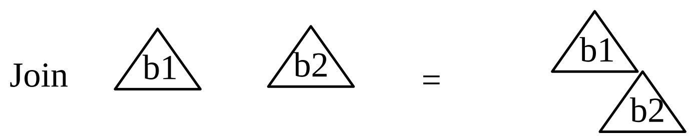
Inorder(t)
= Inorder(t.left) ++ [t.value] ++ Inorder(t.right)
= Inorder(t.left) ++ Inorder(build(t.value, void, t.right))
= Inorder(b1) ++ Inorder(b2)
    where b1 = t.left
        b2 = build(t.value, void, t.right)

Diagram:

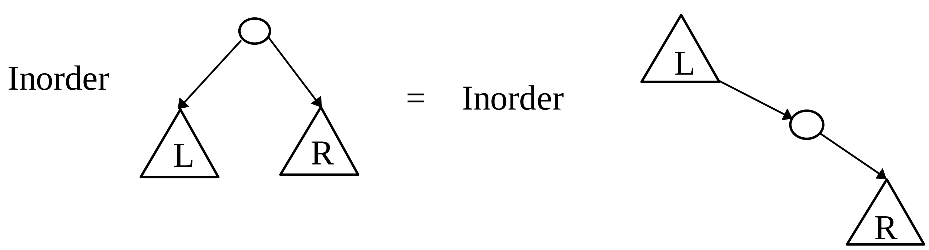


Consider a function Join s.t.

Join(b1, b2) 'joins' b2 to the right most of b1



We get,



Inorder(t)
= Inorder(b1) ++ Inorder(b2)
= Inorder(Join(b1,b2))
    where b1 = t.left
        b2 = build(t.value, void, t.right)

We can give 'abstract-code' for Join,

```
Join(b1,b2 : TREE[G]) is
    do
        if b1 = void then
            result := b2
        else
            result := build(b1.value, b1.left, Join(b1.right, b2))
        end if
    end Join
```

Morris Inorder -- Abstract Code

```
Morris_Inorder(t0 : TREE[G]) : LIST[G] is
        t : TREE[G]
        s : LIST[G]
    do
        from
            t := t0
            s := []
        until
            t = void
        loop
            if t.left = void then
                s := s ++ [t.value]
                t := t.right
            else
                t := Join(t.left, build(t.value, void, t.right))
            end
        end
    end -- Morris_Inorder
```

In Eiffel, we rewrite this as

```
Morris_Inorder(t0:BIN_NODE[STRING]) is
    local
        rm,t : BIN_NODE[STRING]
    do
        from
            t := t0
        until
            t = void
        loop
            if t.left = void then
                io.put_string(t.value)
                io.put_string("  ")
                t := t.right
            else
                from
                    rm := t.left
                until
                    rm.right=void or rm.right=t
                loop
                    rm := rm.right
                end
                if  rm.right = void then
                    rm.Right_Set(t)
                    t := t.left
                else
                    io.put_string(t.value)
                    io.put_string("  ")
                    rm.Right_Set(void)
                    t := t.right
                end
            end
        end -- loop
    end -- Morris_Inorder
```

Discussion:

In Joe Morris' solution, the non-recursive program uses the original data structure for binary tree and so does not use a thread node but yet the solution has similarities to the threaded tree solution in that during traversal a link is formed from a 'right tip' to its inorder successor. More precisely, if t is the root of a subtree then a link is formed from the rightmost of the left subtree of t, call it rm, to t itself. The inorder successor of rm is then t. After rm has being dealt with in the traversal, its right link is restored to void. During program execution, the binary tree is altered to contain cycles but these cycles are removed later.

In traversing the tree, the program has the overhead of setting new links but overall the program is runs in O(n) time.

In the execution of the program, the reference/pointer, t, can be regarded to be in one of 3 possible situations,

- t has no left (sub) tree.
  If t.left = void then t is processed and t moves right.
- t has a left (sub) tree and right most of t.left = void
  In this case, we say t is 'unmarked'.
  A cycle is formed via the right most of t.left, (rm above)
  The right link of rm is linked to t. t is now marked.
- t is marked; the right most of t.left, rm, references t.
  The cycle is broken, and rm.right is restored to void.

In going left during the traversal, cycles are formed and in going right the cycles are broken and the tree restored.

```
class
    INORDER [G -> COMPARABLE]
feature

    stack_inorder (b: BST [G]): ARRAY [G] is
        local
            it: BIN_NODE [G];
            k: INTEGER;
            stk: LINKED_STACK [BIN_NODE [G]]
        do
            from
                !! stk.make;
                !! Result.make (1, b.size);
                it := b.root;
                k := 1
            until
                it = void and  stk.empty
            loop
                from
                until
                    it = void
                loop
                    stk.put (it);
                    it := it.left
                end ;
                it := stk.item;
                stk.remove;
                Result.put (it.value, k);
                k := k + 1;
                it := it.right
            end
        end ;
```

```
    morris_inorder (b: BST [G]): ARRAY [G] is
        local
            k: INTEGER;
            rm, t: BIN_NODE [G]
        do
            from
                t := b.root;
                !! Result.make (1, b.size);
                k := 1
            until
                t = void
            loop
                if  t.left = void then
                    Result.put (t.value, k);
                    k := k + 1;
                    t := t.right
                else
                    from
                        rm := t.left
                    until
                        rm.right=void or  rm.right=t
                    loop
                        rm := rm.right
                    end ;
                    if  rm.right = void then
                        rm.right_set (t);
                        t := t.left
                    else
                        Result.put (t.value, k);
                        k := k + 1;
                        rm.right_set (void);
                        t := t.right
                    end
                end
            end
        end ;
end  -- class INORDER
```

```
class
    BST [G -> COMPARABLE]
feature {NONE}

    update (bt: BIN_NODE [G]; x: G) is
        require
            bt /= void
        local
            t: BIN_NODE [G]
        do
            if x < bt.value then
                if bt.left = void then
                    !! t;
                    t.build (x, void, void);
                    bt.left_set (t);
                    size := size + 1
                else
                    update (bt.left, x)
                end
            elseif x > bt.value then
                if bt.right = void then
                    !! t;
                    t.build (x, void, void);
                    bt.right_set (t);
                    size := size + 1
                else
                    update (bt.right, x)
                end
            end
        end ;

feature

    root: BIN_NODE [G];

    size: INTEGER;



    add (x: G) is
```

```
        do
            if  root /= void then
                update (root, x)
            else
                !! root;
                root.value_set (x);
                size := 1
            end
        end ;


end  -- class BST
```

```
class
    INORDER_ROOT
creation
    make
feature

    b: BST [STRING];


    make is
        local
            tr: INORDER [STRING];
            trav: ARRAY [STRING]
        do
            file2tree ("data.txt");
            print_bst (b, 2);
            !! tr;
            io.put_string ("%NUsing stack_inorder:%N");
            trav := tr.stack_inorder (b);
            print_arr (trav, 1, b.size);
            io.put_string ("%NUsing morris_inorder:%N");
            trav := tr.morris_inorder (b);
            print_arr (trav, 1, b.size)
        end ;
```

```
    print_bst (t: BST [STRING]; indent: INTEGER) is
        do
            io.new_line; io.new_line;
            print_tree (b.root, 2);
            io.new_line
        end ;

file2tree (flname: STRING) is
        local
            in_file: PLAIN_TEXT_FILE;
            str: STRING
        do
            !! in_file.make_open_read (flname);
            from
                !! b;
                in_file.read_word
            until
                in_file.end_of_file
            loop
                str := clone (in_file.last_string);
                b.add (str);
                io.put_string ("%N Added word:  ");
                io.put_string (str);
                in_file.read_word
            end ;
            in_file.close
        end ;

    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_bst (t: BST [STRING]; indent: INTEGER) is
        do
            io.new_line;
            io.new_line;
            print_tree (b.root, 2);
            io.new_line
        end ;

    print_arr (a: ARRAY [STRING]; low, high: INTEGER) is
        local
            k: INTEGER
        do
            from
                io.new_line;
                k := low
            until
                k > high
            loop
                io.put_string (a.item (k));
                io.putchar (' ');
                k := k + 1
            end ;
            io.new_line
        end ;

    spaces (n: INTEGER): STRING is
        do
            !! Result.make (n);
            Result.fill_blank
        end ;
end  -- class INORDER_ROOT
```