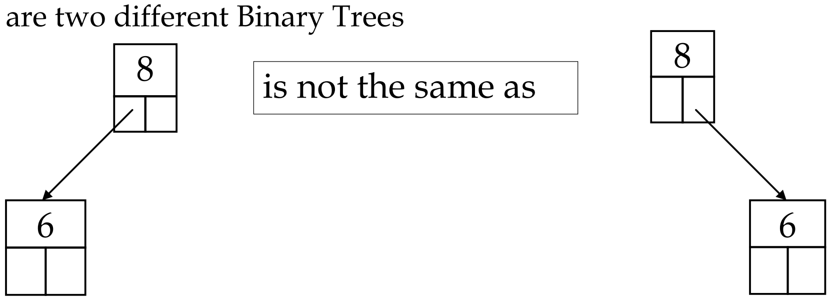
Binary Tree

The Abstract Data type, Binary Tree, is a non-linear data structure. We can define it by an inductive/recursive definition.

A Binary Tree is either empty or made from a Root node and two disjoint Binary (sub)Trees, called the Left and Right subtrees.

A Binary Tree is not quite the same as the notion of a n-ary tree (when n=2) in Graph Theory. In Graph Theory, a tree has no orientation or order, but a Binary Tree with just a Left subree is not the same as one with just a Right subtree.

These are two different Binary Trees



The root nodes of the Left and Right subtrees are called the Left and Right children. The root node is called the Parent of the children. A node with no children is called a Leaf. The root of a binary tree has no parent.

<u>Level</u> of a node:

The Level of the Root node is 1.

The level of a node (not the root) = Level of parent + 1.

<u>Height</u> of a Binary Tree.

= Max level of all the nodes.

Height of the empty tree is 0.

The Class Binary_Tree

A Binary Tree will be implemented in an way analoguos to Linked Lists. We first define the node class for Binary Tree.

```
class BIN_NODE[G]
feature
	value : G
	left, right : BIN_NODE[G]
	left_set  etc.
	right_set  etc.

	Build(v:G; L,R : BIN_NODE[G]) is
	do
		value := v; left := L; right := R
	end -- Build
end -- BIN_NODE
```

We can regard the class BIN_NODE as implementing a binary tree structure as we can view each node as the a subtree, with Left and Right nodes as the Left and Right subtrees. If bt:BIN_NODE then bt can be viewed either as a binary tree or a node.

Lists Again

In a similar way, in the LIST class we can regard a node as being a LIST. We can define a LIST recursively as; The empty list is a LIST and if x is a value and L is a LIST and then we construct a list from x and L

```
class LIST[G]
feature {NONE}
	first_node: NODE[G]

	search(x:G; p:NODE[G]) : NODE[G] is
		require
			p /= void
		do
			if equal(x, p.Value) then
				result := p
			elseif p.next /= void then
				result := search(x,p.next)
			end
		end -- search

feature
	count : INTEGER -- # nodes in the list

	is_empty : BOOLEAN is
		do
			result :=count = 0
		end -- is_empty


	has(x:G):Boolean is
		local
			here : NODE[G]
		do
			if not is_empty then
				here := search(x,first_node)
				result := here /= void
			end
		end -- has
```

```
add(x:G) is  -- repeated items allowed
        local
                n : NODE[G]
        do
                !!n
                n.set_value(x)
                n.set_next(first_node)
                first_node := n
                count := count + 1
end -- add

remove(x:G)  is
        local
                here,p : NODE[G]
        do
                 if equal(x, first_node.value) then
                        first_node := first_node.next
                        count := count-1
                else
                        here := search(x,first_node)
                        if here /= void then
                                from
                                        p := first_node
                                until
                                        p.next = here
                                loop
                                        p := p.next
                                end
                                p.set_next(here.next)
                                count := count-1
                        end
                 end
        end -- remove


remove (x:G) is
-- alternate version using a recursive function, remove_r
        do

                if not is_empty then

                        first_node := remove_r(x, first_node)

                end
        end -- remove
```

```
remove_r (x:G; p: NODE[G]) : NODE[G] is
        -- (recursive) function to remove x from list starting with p.
        local
                n:NODE[G]
        do
                        if p /= void  then
                                if equal(x, p.item) then
                                        Result := p.next
                                else
                                        !!n
                                        n.set_item(p.item)
                                        n.set_next(remove_r(x, p.next))
                                        Result := n
                                end
                        else
                                        Result := void
                        end
        end    -- remove_r

end -- LIST
```

The LIST features are implemented in terms of the class NODE. In searching for a value in the list we check is the value in the first node and recursively search the rest of the nodes. This also can be viewed as checking if the value the first item in the LIST and if not, searching the rest of the LIST. A list may contain repeated items.The class above still separates the NODE class from the LIST class.

To find the node with x, the routine remove uses search. Having found the node it traverses the list from the start to node just before the found node and then it removes the found node.We have a special case for removing 'first' node.

The alternate version of remove, uses a recursive function, remove_r, to remove x from a list starting with node, p. The function, remove_r, has a side-effect of creating new nodes.

<u>B</u>inary <u>S</u>earch <u>T</u>rees (abbreviate to BST)

In analogy with our version of LIST we consider a tree class in which one can Add and Remove items. It is not clear what Add would mean in a non-linear structure, where does one Add in a binary tree; in front of the root (and so make the tree linear again) or to the left or to the right (how is one to decide). The decision is made by only allowing items that conform to COMPARABLE, i.e. the binary trees will only contain items that can be compared. A key can be added to an item is there is no compare operator.

In a Binary Search Tree (BST), all the values in the Left subtree are (strictly) less than the value at the root which is (strictly) less than the values in the right subtree. In Adding an item into a tree we insert it into the left subtree if it is less than the root value and into the right subtree if it is greater.

If the value of the item is equal to the root value, then no insertion takes place. The binary tree contains a 'set' of items, there are no repeated items.

In implementing lists, the LIST class was a client of the CLASS node, and operations we essentially operations on NODE items. In a similar way we will regard the BST class as a binary_tree of BIN_NODE. Like the class LIST, the operations will appear as being operations on BIN_NODE's. We still have the advantage of separating the 2 classes, the class BST and the class BIN_NODE.

An elementary test class tests the features Add and Has.

Also the items in the tree are printed out by an <u>Inorder</u> traversal

```
class BST_TEST
creation  make
feature
	make is
		local
			bt : BST[STRING]
		do
			io.put_string("%NEnter word: -quit- to quit ")
			!!bt
			from
				io.read_word
			until
				io.last_string = “quit”
			loop
				bt.add(io.last_string)
				io.put_string("%NEnter another word : ")
				io.read_word
			end
			io.put_string("%N Looking for word : ")
			io.get_string
			if bt.has(io.last_string) then
				io.put_string("%NWord was found%N")
			else
				io.put_string("%NWord was not found%N")
			end
			Inorder(bt.root) -- Prints out the tree
	end -- make
```

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

Add an Item to a BST

We add an item just once to the tree; there are no repeated items. The Add routine is similar to routine Remove in the class LIST, in that we have to search to the location in the tree where the item is to be inserted. The auxillary feature Insert does the real work. The feature Insert is non-recursive. It needs 2 entities, c to find the location to insert and p tracking 'behind' c to facilitate insertion of a new node.

```
Insert(x:G; t:BIN_NODE[G]) is
        local
                p,c, new : BIN_NODE[G]
        do
                from
                        c := t
                until
                        c = void or else equal(x, c.value)
                loop
                        if x < c.value then
                                p := c
                                c := c.left
                        elseif x > c.value then
                                p := c
                                c := c.right
                        end
                end

                if c = void then
                        if x < p.value then
                                !!new
                                new.build(x,void,void)
                                p.Left_Set(new)
                                count := count+1
                        elseif x > p.value then
                                !!new
                                new.build(x,void,void)
                                p.Right_Set(new)
                                count:= count+1
                        end
                end
        end -- Insert
```

Adding x to the tree, considers the special case of an empty tree.

```
Add(x:G) is
    do
        if root /= void then
            Insert(x,root)
        else
            !!root
            root.build(x,void,void)
            Count := 1
        end
    end -- Add
```

Binary Tree Traversal

In the class LIST we may iterate or traverse the list in a forward direction. If the LIST is kept sorted then the traversal would in effect give an ascending sort or descending sort.

In a Binary Tree we can traverse the tree in either of 3 main ways; Preorder, Inorder or Postorder. Let Left and Right be the left and right subtrees.

- Preorder:
  Root first then Preorder Left then Preorder Right
- Inorder:
  Inorder Left then the Root then Inorder Right.
- Postorder:
  Postorder Left then Postorder Right then the Root

We already have used Inorder in printing out a Binary Tree. If the tree is a BST, then Inorder traverses the tree in ascending order.

Let us assume a procedure Process, that processes an item in some way, e.g. print it.

Let us rename BIN_NODE to BIN_TREE

```
Preorder(bt:BIN_TREE[G]) is
    do
        If bt /= void then
            Process(bt.value)
            Preorder(bt.left)
            Preorder(bt.right)
        end
    end -- Preorder
```

```
Postorder(bt:BIN_TREE[G]) is
	do
		If bt /= void then
			Postorder(bt.left)
			Postorder(bt.right)
			Process(bt.value)
		end
	end -- Postorder
```

Each time these routines are called, a check is made for the empty tree. To avoid a recursive call on an empty or void tree we rewrite Inorder as follows:
(Preorder & Postorder could also be rewritten in this way)

```
Inorder(bt: BIN_TREE[G]) is
	do
		If bt /= void then
			Inord(bt)
	end
end -- Inorder
```

where

```
Inord(bt:BIN_TREE[G]) is
	require
		Not_Void: bt /= void
	do
		if t.left /= void then
			Inord(t.left)
		end

		Process(t.value)
		if t.right /= void then
			Inord(t.right)
		end
	end --Inord
```

Recursive Version of BST routines

In implementing Add we use a recursive function Update and for Remove we use a recursive function Delete.

Recursive Update of a BST

Update is a recursive version of Insert. We have used the procedure Insert to define a procedure Add that adds an item to a BST. Similarly we can define Add in terms of Update.

Warning:
Update is implemented as a recursive function. Update is not a procedure but a function with side-effects

Side-Effects
Eiffel strongly advises against side-effects in functions.
In "Object Oriented Software Construction" p139, Meyer states

> "*Side-effects are prohibited in functions,*
> *except if they only affect the concrete state*"

Side-Effects

Allowable Side-Effects:

An example of an allowable side-effect function would be a function that returns the i^{th} item in a list by moving a 'hidden' cursor to the i^{th} item. The cursor would not be an exported attribute and so would be part of the concrete state. The abstract state is defined in terms of the exported attributes.

Disallowed Side-Effects:

As an example of a disallowed side-effect function consider a C-like function 'get_int' that moves the file pointer as well as returning a value. We would have in this case the undesirable consequence that

get_int + get_int = 2*get_int

would most likely be false.

Parameters in Eiffel Routine

No VAR (In Out) parameter in Eiffel:

In our implementation of Update we take advantage of a recursive formulation. We need a side-effect function as Eiffel does not (for good reasons) have the Modula-2 equivalent of a VAR parameter or the Ada equivalent of an 'in out' parameter.

Eiffel has 'Value' parameters only:

Eiffel uses only 'value' parameters in all its routines (functions & procedures). So if p is a parameter in an Eiffel routine it is an error to have in the body of the routine something like

p := q -- wrong or !!p -- wrong

Both of these statements change the reference of p.

Update

The function Update uses the recursive structure of a Binary Tree to

Update the Left subtree is x < root value

or Update the Right subtree is x > root value.

If x is in the Binary Tree then Update returns the tree as it was and if x is not in the tree, a new node is created with value x and attached to the tree.

Update creates new structure:

The Update function, in effect, creates a new structure with the same items. The old structure is 'garbage collected' by the Eiffel system.

```
Update(x:G; bt:BIN_NODE[G]) : BIN_NODE[G] is
local
        t : BIN_NODE[G]
do
   if x = bt.value then
        result := bt
   else
        !!t -- create new BIN_NODE
        if bt = void then
              t.build(x,void,void)
              Count := Count+1
        elseif x < bt.value then
              t.build(bt.value, Update(x, bt.left), bt.right)
        elseif x > bt.value then
              t.build(bt.value, bt.left, Update(x,bt.right))
        end
        result := t
   end
end -- Update
```

The Procedure Add

To Add an item to a BST we Update starting at the root.

```
Add_Rec(x:G) is
    do
        root := Update(x,root)
    end -- Add_Rec
```

The Procedure Remove

```
Remove(x:G)  is
    do
        root := Delete(x,root)
    end --Remove
```

Recursive Delete

The pattern for recursive Delete is similar to that of Update except for the special case of deleting a 'root' node. If x is the value of a root node that has a non-void left subtree then to delete x we replace the value x of the root with the value of its predecessor in the tree, and delete the predecessor node. In a BST, the predecessor value is found in the Right_Most node of the left subtree.

```
Right_Most(bt:BIN_NODE[G]) : G is
    require
        Not_Void: bt /= void
    local
            t : BIN_NODE[G]
    do
        from
            t := bt
        until
            t.right = void
        loop
            t := t.right
        end
        result := t.value
    end -- Right_Most
```

The Delete function uses the auxillary function Delete_Root for the special case of deleting a root node.

```
Delete(x:G; bt:BIN_NODE[G]) : BIN_NODE[G] is
      local
            t : BIN_NODE[G]
      do
            if bt /= void then
                  if equal(x, bt.value) then
                        result := Delete_Root(x, bt)
                  else
                  !!t
                  if x < bt.value then
                        t.build(bt.root, Delete(x,bt.left), bt.right)
                  elseif x > bt.value then
                        t.build(bt.root, bt.left, Delete(x,bt.right))
                  end
                  result := t
                  end
            end
   end -- Delete


Delete_Root(x:G; bt:BIN_NODE[G]) : BIN_NODE[G] is
      require
            Not_Void : bt /= void
      local
            rm : G
            t : BIN_NODE[G]
      do
            if bt.left = void then
                  result := bt.right
            elseif bt.right = void then
                  result := bt.left
            else
                  rm := Right_Most(bt.left)
                  !!t
                  t.build(rm, Delete(rm,bt.left), bt.right)
                  result := t
            end
      end -- Delete_Root
```