TREE : a Non-Linear Data Structure

****

A Tree is a finite set of one or more nodes such that -

- there is a specially designated node called ROOT, and
- the remaining nodes are partitioned into $N \geq 0$ disjoint sets, T_1, T_2, ..., T_N, where each of these sets is a tree. T_1 through T_N are called subtrees of the root.

A free tree is a connected, acyclic, undirected graph. If an undirected graph is acyclic but possibly disconnected, it is a forest.

In an undirected graph, $G=(V,E)$, the edge set E consists of unordered pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u,v\}$, where $u,v \in V$ and $u \neq v$. In an undirected graph, self-loops are forbidden, and so every edge consists of two distinct vertices .

Properties of Free Trees

Let G=(V,E) be an undirected graph. The following statements are equivalent

- G is a free tree.
- Any two vertices in G are connected by a unique simple path.
- G is connected, but if any edge is removed from E, resulting graph is dis connected.
- G is connected, and |E|=|V|-1
- G is acyclic, and |E|=|V|-1
- G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

Rooted Trees & Ordered Trees

A rooted tree is a free tree in which one of the vertices is distinguished (called the root) from the others. A vertex of a rooted tree is called node of the tree.

Consider a node x in a rooted tree T with root r. Any node y on the unique simple path from r to x an ancestor of x. If y is an ancestor of x, then x is a descendant of y. (Every node is both an ancestor and a descendant of itself.)

If y is an ancestor of x and $x \neq y$, then y is a **proper ancestor** of x and x is a **proper descendant** of y. The subtree rooted at x is the tree **induced** by descendants of x, rooted at x.

If the last edge on the simple path from the root r of a tree T to a node x is (y,x), then y is the **parent** of x, and x is a **child** of y.

The root is the only node in T with no parent. If two nodes have the same parent, they are **siblings**. A node with no children is a **leaf** or **external** node. A nonleaf node is an **internal** node.

The number of children of a node x in a rooted tree T equals the **degree of x**. The length of the simple path from the root r to a node x is the **depth** of x in T .

A **level** of a tree consists of <u>all nodes at the same depth</u>. The **height of a node** in a tree is the number of edges on the <u>longest simple downward path</u> from the node to a leaf, and the **height of a tree** is the <u>height of its root</u>. The height of a tree is also equal to the largest depth of any node in the tree.

An **ordered tree** is a rooted tree in which the children of each node are ordered.

Binary and Positional Trees

A binary tree T is a structure defined on a finite set of nodes that either -

- contains no nodes, or
- is composed of three disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

The binary tree that contains no nodes is called the empty tree or null tree .

In a positional tree, the children of a node are labeled with distinct positive integers. The i^{th} child of a node is absent if no child is labeled with integer i.

A complete k-ary tree is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k.

Binary Tree

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

Recursively, a (non-empty) binary tree is a tuple (L, S, R), where L and R are binary trees or the empty set and S is a singleton set.

Types of Binary Trees

- A rooted binary tree has a root node and every node has at most two children.
- A full binary tree (also a proper or plane binary tree) is a tree in which every node has either 0 or 2 children.
- A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
- In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes at the last level h. A complete binary tree can be efficiently represented using an array.A balanced binary tree has the minimum possible maximum height (a.k.a. depth) for the leaf nodes because, for any given number of leaf nodes, the leaf nodes are placed at the greatest height possible.

Properties of Binary Tree

- The number of nodes n in a full binary tree, is at least $n=2h+1$ and at most $n=2^{h+1}-1$, where h is the height of the tree. A tree consisting of only a root node has a height of 0.
- The number of leaf nodes l in a perfect binary tree, is $l=(n+1)/2$.
- A perfect binary tree with l leaves has $n=2l-1$ nodes.
- The number of internal nodes in a complete binary tree of n nodes is floor(n/2).
- In a balanced full binary tree,

$$h = \lceil \log_2(l) \rceil + 1 = \lceil \log_2((n+1)/2) \rceil + 1 = \lceil \log_2(n+1) \rceil$$

```
#define MAXOF(x, y) ((x) >= (y) ? (x) : (y))
```

The node structure for a binary tree is defined as -

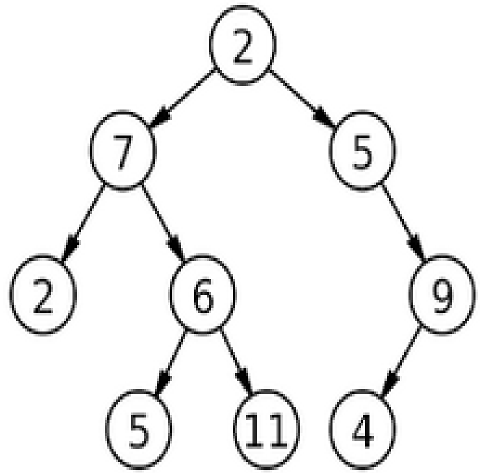
```
struct treeNode {
   int data;
   struct treeNode *lchild;
   struct treeNode *rchild;
};
typedef struct treeNode tNode;
typedef tNode* tree;
```

Consider a rooted binary tree of depth = 3.

Thus, the maximum nodes = $2^{3+1}-1 = 15$

This tree can be represented using an array as -

[2, 7, 5, 2, 6, <u>Null</u>, 9, Null, Null, 5, 11, <u>Null</u>, <u>Null</u>, 4, Null]



It should be noted that,

- No of internal nodes = floor(n/2)= floor(9/2) = 4
- For each non-leaf node (root and internal node) located at index i,
  - its left child is present at $left_{index}=2*i+1$, and
  - the right child is present at $right_{index}=2*i+2$, respectively.

Function LENGTH_LIST(A)

Given an array A containing keys, the last of which is a special value MXVAL, this function returns length (i.e., number of keys present) of A. NDX is a local parameter.

1. Initialize Local Variable

    NDX = 0

2. Iterate through the array

    While A[NDX] <> MXVAL

        NDX := NDX + 1

3. Return, length of the array

    Return NDX-1

Function GET_LEFT_CHILD(A,LEN,NDX)

Given an array A containing keys, the last of which is a special value MXVAL, this function returns the position of left child of the node whose index is NDX. LEN denotes the length of the list (i.e., returned by LENGTH_LIST(A)).

1. For leaf nodes at non-final level on right of last tree node.

    If 2*NDX+1 > LEN

        Return LEN

2. Otherwise, Return the left child position

    Return 2*NDX+1

```
Function GET_RIGHT_CHILD(A,LEN,NDX)
    Given an array A containing keys, the last of which is a special value MXVAL, this function
    returns the position of right child of the node whose index is NDX. LEN denotes length of A.
1.  For leaf nodes at non-final level on right of last tree node.
        If 2*NDX+2 > LEN
            Return LEN
2.  Otherwise, Return the right child position
        Return 2*NDX+2
```

```
Function BUILD_BIN_TREE(A,LEN,NDX)
    Given an array A containing keys, the last of which is a special value MXVAL, and length LEN,
    this function builds a binary tree originating at the node position denoted by NDX. T is a
    local tree pointer, NDT is a special key indicating a nullnode (say -1).
1.  Recurse, on an existing node
        If A[NDX] <> NDT
            T = Call CREATE_TNODE()
            LCHILD(T) = Call BUILD_BIN_TREE(A, LEN, Call GET_LEFT_CHILD(A,LEN,NDX))
            DATA(T) = A[NDX]
            RCHILD(T) = Call BUILD_BIN_TREE(A, LEN, Call GET_RIGHT_CHILD(A,LEN,NDX))
2.  Return the tree
        Return T
```

TREE Traversals

A rooted binary tree can be traversed the following most common ways –

- the inorder traversal [leftChild – root – rightChild]
- the postorder traversal [leftChild – rightChild – root] and
- the preorder traversal [root – leftChild – rightChild]

```
Procedure INORDER(ROOT)
    Given a rooted binary tree denoted by ROOT, this procedure prints the inorder traversal of the
    tree.
1.  Recurse, on an existing node
        If ROOT <> NULL
            Call INORDER(LCHILD(ROOT))         /* Traverse left subtree  */
            Write(DATA(ROOT))                  /* Print Key at Node      */
            Call INORDER(RCHILD(ROOT))         /* Traverse right subtree */
2. Otherwise,return routine call
        Return
```

Procedure PREORDER(ROOT)

Given a rooted binary tree denoted by ROOT, this procedure prints the preorder traversal of the tree.

```
1.  Recurse, on an existing node
        If ROOT <> NULL
            Write(DATA(ROOT))                 /* Print Key at Node      */
            Call PREORDER(LCHILD(ROOT))       /* Traverse left subtree  */
            Call PREORDER(RCHILD(ROOT))       /* Traverse right subtree */
2.  Otherwise,return routine call
        Return
```

Procedure POSTORDER(ROOT)

Given a rooted binary tree denoted by ROOT, this procedure prints the postorder traversal of the tree.

```
1.  Recurse, on an existing node
        If ROOT <> NULL
            Call POSTORDER(LCHILD(ROOT))      /* Traverse left subtree  */
            Call POSTORDER(RCHILD(ROOT))      /* Traverse right subtree */
            Write(DATA(ROOT))                 /* Print Key at Node      */
2.  Otherwise,return routine call
        Return
```

```
Function HEIGHT_BIN_TREE(ROOT)
    Given a rooted binary tree pointed by ROOT, this function recursively computes the height of
    the tree. A leaf node is assumed at height = 0.
1.  Empty Tree?? Return -1 [Terminating Case-1]
        If ROOT = NULL
            Return -1
2.  Leaf Node? Return 0 [Terminating Case-2]
        If LCHILD(ROOT)= NULL AND RCHILD(ROOT)= NULL
            Return 0
3.  Recursion?? Compute height
        Return MAX(Call HEIGHT_BIN_TREE(LCHILD(ROOT), Call HEIGHT_BIN_TREE(RCHILD(ROOT)) + 1
```

```
Function PARENTS_BT(ROOT)
    Given a rooted binary tree pointed by ROOT, this function recursively finds the parent
    [internal] nodes of the tree. The function returns the count of parent nodes.
1.  Is the tree empty or the node has no children??
        If ROOT = NULL OR (LCHILD(ROOT) = NULL AND LCHILD(ROOT) = NULL)
            Return 0
2.  Print the Node
        Write(DATA(ROOT))
3.  Return the result
        Return ( Call PARENTS_BT(LCHILD(ROOT)) + Call PARENTS_BT(RCHILD(ROOT)) + 1 )
```

Function ALL_NODES_BT(ROOT)

Given a rooted binary tree pointed by ROOT, this function recursively prints all nodes of the tree. The function returns the count of nodes. KOUNT is a local integer variable.

```
1.  Is the tree empty??
        If ROOT = NULL
            Return 0
2.  Initiate KOUNT and Print the Node
        KOUNT := 1
        Write(DATA(ROOT))
3.  Traverse left- and right- subtrees
        KOUNT = KOUNT + Call ALL_NODES_BST(LCHILD(ROOT)) + Call ALL_NODES_BST(RCHILD(ROOT))
4.  Return the result
        Return KOUNT
```

```
Function LEAVES_BT(ROOT)
    Given a rooted binary (search) tree pointed by ROOT, this function recursively finds the leaf
    [terminal] nodes of the tree. The function returns the count of leaf nodes.
1.  Is the tree empty??
        If ROOT = NULL
            Return 0
2.  Is this a leaf??
        If LCHILD(ROOT) = NULL AND RCHILD(ROOT) = NULL
            Write(DATA(ROOT))   /* Print the Node */
            Return 1

3.  Return the result
        Return ( Call LEAVES_BT(LCHILD(ROOT) + Call LEAVES_BT(RCHILD(ROOT) )
```

Procedure EMPTY_BT(ROOT)

Given a rooted binary tree pointed by ROOT, this procedure recursively restores the nodes of the tree to availability stack, AVAIL and returns a NULL tree pointer.

```
1.  If tree exists, set up the recursion
        If ROOT <> NULL
            /**Traverse to left subtree */
            Call EMPTY_BT(LCHILD(ROOT))
            /**Traverse to right subtree */
            Call EMPTY_BT(RCHILD(ROOT))
            /**Print the deleted node data*/
            Write('Released Node with Key ', DATA(ROOT))
            /**Restore the node*/
            Restore(ROOT)
```

BINARY SEARCH TREE [BST]

A binary search tree is a rooted ordered binary tree with following properties -

- At each node, its left subtree contains nodes having keys lesser than node's key
- At each node, its right subtree contains nodes having keys greater than node's key
- Both the left- and right- subtree is a BST
- Duplicate keys are not allowed.

BST has been amongst the first indexing structures used for external memory.

In a BST implementation the inorder traversal always results in keys sorted in ascending order.

BST supports an efficient mechanim for searching a key, taking a maximum of $\log_2(N)$ probes.

BST is a well-known and frequently implemented library routine.

In an efficient implementation of a BST, the cost of Insert(), Delete() and LookUp() can be $O(\log_2(N))$. Here, N is the number of nodes in the tree.

BST are best suited for the order statistics (i.e., Nth smallest, Nth largest) as it builts a sorted array of keys.

```
Function INSERT_BST(ROOT, KEY)
    This function creates a Binary Search Tree node and returns a pointer of the created node. If
    the node cacnot be allocated from AVAIL, the function return NULL. NEWW is a local tree
    pointer.
1.  Is the tree is empty?? If YES, create a tree node
        If ROOT = NULL
            ROOT := Call CREATE_TNODE()
            /**Is the node usable?? If NO, initialize the node */
            If ROOT = NULL
                Write('AVAIL Underflow, Insert Failed...')
            Else
                DATA(ROOT) = KEY
                LCHILD(ROOT) = RCHILD(ROOT) = NULL
2.  Otherwise recursively add a node
        Else If KEY < DATA(ROOT)
                LCHILD(ROOT) = Call INSERT_BST(LCHILD(ROOT), KEY) /**Add to left subtree */
            Else If KEY < DATA(ROOT)
                RCHILD(ROOT) = Call INSERT_BST(RCHILD(ROOT), KEY) /**Add to right subtree */
            Else /**Report Duplicate KEY */
                Write('Duplicate key')
3.  Return the tree
        Return ROOT
```

Function CREATE_BST()

This function creates a Binary Search Tree (excluding the duplicate keys) from the set of keys entered by the user. For a better implementattion the keys can be populated in an array, LIST, which may be passed as input parameter to this function. The function returns a TREE pointer to the generated tree. KEY is the local variable denoting data value of the tree node. ROOT denotes the head node of the created tree.

1. Initialize ROOT

    ROOT = NULL

2. Set up the iteration for insertion of a node

    Repeat Step 3 thru 4 until KEY = STOP

3. Acquire the node value

    read(KEY)

4. Insert the Node

    If KEY <> STOP

        Call INSERT_BST(ROOT, KEY)

5. Return the tree

    Return ROOT

```
Function HAS_LEFT_CHILD(ROOT)
    This function determines whether a BST node pointed by ROOT has a left-child and returns TRUE
    (i.e., value = 1) if the left-child exists, otherwise returns FALSE (i.e., value = 0).
1.  Is there a left child??
        If LCHILD(ROOT) <> NULL
            Return 1
        Return 0
```

```
Function HAS_RIGHT_CHILD(ROOT)
    This function determines whether a BST node pointed by ROOT has a right-child and returns TRUE
    (i.e., value = 1) if the right-child exists, otherwise returns FALSE (i.e., value = 0).
1.  Is there a right child??
        If RCHILD(ROOT) <> NULL
            Return 1
        Return 0
```

```
Function IS_LEAF(ROOT)
    This function determines whether a BST node pointed by ROOT is a leaf node.
1.  Is there a right child??
        If ( NOT (Call HAS_LEFT_CHILD(ROOT)) AND NOT (Call HAS_RIGHT_CHILD(ROOT)) )
            Return 1
        Return 0
```

```
Function INTERNAL_NODES(ROOT)
    This function returns the internal (parent|non-terminal) nodes of a BST node pointed by ROOT.
1.  Is empty tree or tree is a single node??
        If ROOT = NULL OR (Call IS_LEAF(ROOT))
            Return 0
        Return 1 + Call INTERNAL_NODES(LCHILD(ROOT)) + Call INTERNAL_NODES(RCHILD(ROOT))

Function FIND_MIN_NODE(ROOT)
    This function returns the node with minimum data value of a rooted BST pointed by ROOT.
1.  Is empty tree??
        If ROOT = NULL
            Return NULL
2.  The rooted tree has no left child
        If LCHILD(ROOT) = NULL
            Return ROOT
3.  Recursively traverse the left sub-tree
        Return ( Call FIND_MIN_NODE(LCHILD(ROOT)) )
```

```
Function DELETE_NODE_BST(ROOT, KEY)
    Given a rooted Binary Search Tree pointed by ROOT, this function deletes a node with data value
    equal to KEY from the tree. TEMP is the local tree pointer.
1.  Is empty tree??
        If ROOT = NULL
            Write('Delete Failed, Empty Tree')
            Return NULL
2.  Traverse the subtrees to locate the node to be deleted
        If KEY < DATA(ROOT)
            LCHILD(ROOT) = Call DELETE_NODE_BST(LCHILD(ROOT), KEY) /* Traverse Left Subtree  */
        Else If KEY > DATA(ROOT)
            RCHILD(ROOT) = Call DELETE_NODE_BST(RCHILD(ROOT), KEY) /* Traverse Right Subtree */
        Else                                                       /* The intended node      */
            If LCHILD(ROOT)<> NULL AND RCHILD(ROOT)<> NULL  /* Case-1: Node has both children*/
                TEMP = Call FIND_MIN_NODE(RCHILD(ROOT))
                DATA(ROOT) = DATA(TEMP)
                RCHILD(ROOT) = Call DELETE_NODE_BST(RCHILD(ROOT), DATA(ROOT))
            Else
                TEMP = ROOT
```

```
            If LCHILD(ROOT) = NULL                  /* Case-2: Node has only right child*/
                ROOT = RCHILD(ROOT)
            Else If RCHILD(ROOT) = NULL             /* Case-3: Node has only left child */
                ROOT = LCHILD(ROOT)
            Restore TEMP                            /* Case-4: Its a Leaf node, Restore */
3.  Return the tree
        Return (ROOT)
```

Procedure LEVEL_ORDER (ROOT)

Given a rooted binary tree denoted by ROOT, this procedure traverses the tree level-by-level and prints the data content of the nodes from left-to-right on the said level. LEVEL is a local integer variable. The procedures NODES_AT_LEVEL() and HEIGHT() prints data contents of nodes at specified level and height of the tree respectively.

```
1.  Initialize Local Variable
        LEVEL := 0
2.  Iterate through tree
        While LEVEL <= (Call HEIGHT(ROOT))
            Call NODES_AT_LEVEL(ROOT, LEVEL)
            LEVEL := LEVEL+1
```

```
Procedure NODES_AT_LEVEL (ROOT, LEVEL)
    Given a rooted binary tree denoted by ROOT, this procedure prints data content of the nodes
    from left-to-right on the given level, LEVEL.
1.  Terminating Condition [Empty Tree]
        If ROOT = NULL
            Return
2.  Set up Recursion
        If LEVEL = 0
            Write( DATA(ROOT) )
3.  Otherwise, traverse left- and right- subtrees
        Else
            Call NODES_AT_LEVEL( LCHILD(ROOT), LEVEL-1)
            Call NODES_AT_LEVEL( RCHILD(ROOT), LEVEL-1)
```