Trees

The ADT tree

A tree is a finite set of elements or nodes. If the set is non-empty, one of the nodes is distinguished as the root node, while the remaining (possibly empty) set of nodes are grouped into subsets, each of which is itself a tree. This hierarchical relationship is described by referring to each such subtree as a child of the root, while the root is referred to as the parent of each subtree. If a tree consists of a single node, that node is called a leaf node.

Basic Tree Concepts

- A tree consists of a finite set of elements, called ‘nodes’, and a finite set of directed lines, called ‘branches’, that connect the nodes.
- The number of branches associated with a node is the degree of the node.
  - When the branch is directed toward a node, it is an indegree branch; when the branch is directed away from the node, it is an outdegree branch.
  - The sum of indegree and outdegree branches is the degree of the node.
  - The indegree of the root is by definition is zero.

- A leaf is any node with an outdegree of zero.
- A node that is not a root or a leaf is known as an internal node.
- A node is a parent if it has successor nodes – that is, if it has an outdegree greater than zero. Conversely, a node with a predecessor is a child. A child node has an indegree of one.
- Two or more nodes with the same parent are siblings.
A path is a sequence of nodes in which each node is adjacent to the next one.

Every node in the tree can be reached by following a unique path starting from the root.

The level of a node is its distance from the root. Because the root has a zero distance from itself, the root is at level 0. The children of the root are at the level 1.

The height or length of the tree is the level of the leaf in the longest path from the root plus 1. By definition, the height of an empty tree is -1.

A tree may be divided into subtrees. A subtree is any connected structure below the root.

The first node in a subtree is known as the root of the subtree and is used to name the subtree.

**Binary Trees**

A binary tree is a tree in which no node can have more than two subtrees.

These subtrees are designated as the left subtree and right subtree.

Each subtree is a binary tree itself.
The **height of the binary trees** can be mathematically predicted. The maximum height of the binary tree which has $N$ nodes,

$$H_{\text{max}} = N$$

A tree with a maximum height is rare. It occurs when the entire tree is built in one direction. The **minimum height of the tree**, $H_{\text{min}}$ is determined by,

$$H_{\text{min}} = \left\lfloor \log_2 N \right\rfloor + 1$$

Given a height of the binary tree, $H$, the minimum and maximum number of nodes in the tree are given as,

$$N_{\text{min}} = H \quad \text{and,} \quad N_{\text{max}} = 2^H - 1$$

If the height of the tree is less, then it is easier to locate any desired node in the tree.

To determine whether tree is balanced, the **balance factor** should be calculated.

If $H_L$ represents the height of the left subtree and $H_R$ represents the height of the right subtree then **Balance factor**,

$$B = H_L - H_R$$

A tree is balanced if its balance factor is 0 and its subtrees are also balanced.

A binary tree is balanced if the height of its subtrees differs by no more than one and its subtrees are also balanced.

A **complete tree** has the maximum number of entries for its height.

The maximum number is reached when the least level is full. The maximum number is reached when the last level is full.

A tree is considered **nearly complete** if it has the minimum height for its nodes and all nodes in the last level are found on the left.
A **binary tree** is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has just a right child; or
- has both a left and a right child.

A Complete binary tree of depth $K$ is a binary tree of depth $K$ having $2^k - 1$ nodes.

![Binary Tree Diagram](image.png)

A very simple representation for such binary tree results from sequentially numbering the nodes, starting with nodes on level 1 then those on level 2 and so on. Nodes on any level are numbered from left to right as shown in the above picture. This numbering scheme gives us the definition of a complete binary tree. A binary tree with $n$ nodes and of depth $K$ is complete if its nodes correspond to the nodes which are numbered one to $n$ in the full binary tree of depth $K$.

**Array Representation:**

Each node contains **info**, **left**, **right** and **father** fields. The left, right and father fields of a node point to the node’s left son, right son and father respectively.

Using the array implementation, we may declare:

```c
#define NUMNODES 100
struct nodetype
{
    int info;
    int left;
    int right;
    int father;
};
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

Under this representation,

- **info(p)** would be implemented by reference `node[p].info`,
- **left(p)** would be implemented by reference `node[p].left`,
- **right(p)** would be implemented by reference `node[p].right`,
- **father(p)** would be implemented by reference `node[p].father` respectively.

The operations,

- **isleft(p)** can be implemented in terms of the operation **left(p)**
- **isright(p)** can be implemented in terms of the operation **right(p)**
The above trees can be represented in memory sequentially as follows:

A
B
- 
C 
- 
- 
D
- 
E

The above representation appears to be good for complete binary trees and wasteful for many other binary trees. In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.

Linked Representation:

The problems of sequential representation can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA, and RCHILD as represented below:

LCHILD  DATA  RCHILD

Fig (a)

Fig (b)
In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation, we may declare,

```c
struct nodetype
{
    int info;
    struct nodetype *left;
    struct nodetype *right;
    struct nodetype *father;
};
typedef struct nodetype *NODEPTR;
```

This representation is called dynamic node representation. Under this representation,

- `info(p)` would be implemented by reference `p->info`,
- `left(p)` would be implemented by reference `p->left`,
- `right(p)` would be implemented by reference `p->right`,
- `father(p)` would be implemented by reference `p->father`.

### PRIMITIVE OPERATION ON BINARY TREES

1. **maketree() function**
   Which allocates a node and sets it as the root of a single-node binary tree, may be written as follows;

   ```c
   NODEPTR maketree(x)
   int x;
   {
   NODEPTR p;

   p = getnode(); /* getnode() function get a available node */
   p->info = x;
   p->left = NULL;
   p->right=NULL;
   return(p);
   }
   ```

2. **setleft(p,x) function**
   Which sets a node with contents x as the left son of node(p)

   ```c
   setleft(p,x)
   NODEPTR p;
   int x;

   if(p == NULL)
       printf(“insertion not made”);
   else if ( p->left != NULL)
       printf(“invalid insertion “);
   else
       p->left = maketree (x);
   ```
Conversion of a General Tree to Binary Tree

General Tree:

- A General Tree is a tree in which each node can have an unlimited out degree.
- Each node may have as many children as is necessary to satisfy its requirements. Example: Directory Structure

![General Tree Diagram]

- It is considered easy to represent binary trees in programs than it is to represent general trees. So, the general trees can be represented in binary tree format.

Changing general tree to Binary tree:

- The binary tree format can be adopted by changing the meaning of the left and right pointers. There are two relationships in binary tree:
  - Parent to child
  - Sibling to sibling

  Using these relationships, the general tree can be implemented as binary tree.

Algorithm

1. Identify the branch from the parent to its first or leftmost child. These branches from each parent become left pointers in the binary tree
2. Connect siblings, starting with the leftmost child, using a branch for each sibling to its right sibling.
3. Remove all unconnected branches from the parent to its children

![Binary Tree Diagram]

Step 1: Identify all leftmost children

Step 2: Connect Siblings
A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.

The two general approaches to the traversal sequence are,
- **Depth first traversal**
- **Breadth first traversal**

In depth first traversal, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child. **In other words, in the depth first traversal, all the descendants of a child are processed before going to the next child.**

In a breadth-first traversal, the processing proceeds horizontally form the root to all its children, then to its children’s children, and so forth until all nodes have been processed. **In other words, in breadth traversal, each level is completely processed before the next level is started.**

**Depth-First Traversal**

There are basically three ways of binary tree traversals. They are:

1. **Pre Order Traversal**
2. **In Order Traversal**
3. **Post Order Traversal**

In C, each node is defined as a structure of the following form:

```c
struct node
{
    int info;
    struct node *lchild;
    struct node *rchild;
}
typedef struct node NODE;
```
**Binary Tree Traversals**  (Recursive procedure)

1. **Inorder Traversal**

Steps:
1. Traverse left subtree in inorder
2. Process root node
3. Traverse right subtree in inorder

**Algorithm**

Algorithm inorder traversal (Bin-Tree T)

Begin
If ( not empty (T) ) then

Begin
Inorder traversal (left subtree (T))
Print (info (T)) /* process node*/
Inorder traversal (right subtree (T))
End
End

**C Coding**

```c
void inorder_traversal ( NODE * T)
{
    if ( T ! = NULL)
    {
        inorder_traversal(T->lchild);
        printf("%d \t", T->info);
        inorder_traversal(T->rchild);
    }
}
```

The Output is: C → B → D → A → E → F
2. Preorder Traversal

Steps: 1. Process root node
2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

Algorithm

Algorithm preorder traversal (Bin-Tree T)

Begin
If (not empty (T)) then
Begin
    Print (info (T)) / * process node */
    Preorder traversal (left subtree (T))
    Inorder traversal (right subtree (T))
End
End

C function
void preorder_traversal (NODE *T)
{
    if (T != NULL)
    {
        printf("%d", T->info);
        preorder_traversal(T->lchild);
        preorder_traversal(T->rchild);
    }
}

Output is: A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F
3. Postorder Traversal

Steps:  
1. Traverse left subtree in postorder  
2. Traverse right subtree in postorder  
3. process root node

Algorithm

Postorder Traversal

Algorithm postorder traversal (Bin-Tree T)

Begin
  If (not empty(T)) then
  Begin
    Postorder_traversal (left subtree (T))
    Postorder_traversal (right subtree(T))
    Print (Info(T)) /* process node */
  End
End

C function

void postorder_traversal (NODE *T)
{
  if (T != NULL)
  {
    postorder_traversal(T->lchild);
    postorder_traversal(T->rchild);
    printf("%d", T->info);
  }
}

The Output is: C → D → B → F → E → A
Non-Recursive algorithm: Inorder_Traversal

```c
#define MAXSTACK 100

inorder_traversal (tree)
NODEPTR tree;
{
    struct stack
    {
        int top;
        NODEPTR item[MAXSTACK];
    } s;
    NODEPTR p;

    s.top = -1;
p = tree;
do
    {   /* travel down left branches as far as possible, saving
        pointers to nodes passed */
        while(p!=NULL)
        {
            push(s,p);
p = p->left;
        }   /* check if finished */
        if (!empty (s))
        {
            p = pop(s);
            printf("%d \n", p->info);
            p = p->right;
        }
    }while( !empty (s) || p = NULL );
```
Non-Recursive algorithm: Preorder_Traversal

#define MAXSTACK 100

preorder_traversal (tree)
{
    NODEPTR tree;
    {
        struct stack
        {
            int top;
            NODEPTR item[MAXSTACK];
        } s;
        NODEPTR p;
    }
    s.top = -1;
    p = tree;
    do
    {
        /* travel down left branches as far as possible, saving
           pointers to nodes passed */
        if(p!=NULL) {
            printf("%d\n", p->info);
            /* visit the root */
            if(p->right!=NULL) push(s,p->right); /* push the right subtree
                                                    on to the stack */
            p=p->left;
        }
        else p=pop(s);
    }while( ! empty(s) || p = NULL )
Binary Search Tree

Binary tree that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree of n are greater than or equal to the contents of n.

Uses: used in sorting and searching

Applications of Binary Trees  (find the duplication element from the list)

Binary tree is useful data structure when two-way decisions must be made at each point in a process. For example find the all duplicates in a list of numbers. One way doing this is each number compare with it’s precede it. However, it involves a large number of comparisons. The number of comparisons can be reduced by using a binary tree.

Step 1: from root, each successive number in the list is then compared to the number in the root.
Step 2: If it matches, we have a duplicate.
Step 3: If it is smaller, we examine the left subtree.
Step 4: If it is larger, we examine the right subtree.
Step 5: If the subtree is empty, the number is not a duplicate and is placed into a new node at that passion in the tree.
Step 6: If the subtree is nonempty, we compare the number of the contents of root of the subtree and entire process is repeated till all node completed.

/* read the first number and insert it into a sinlge-node binary tree */
scanf("%d", &number);
tree = maketree (number);
while (there are numbers left in the input)
{
    scanf("%d", &number);
p = q = tree;
    while (number != info(p) && q != NULL)
    {
        p = q;
if ( number < info(p) )
    q = left(p);
else
    q = right(p);
}

if(number == info(p) )
    printf(" %d %s ", number, "is a duplicate");
else if ( number < info(p) )
    setleft( p, number );
else
    setright(p, number);
}

(2) Application of Binary Tree - (Sort the number in Ascending Order)

If a binary search tree is traversed in inorder(left,root,right) and the contents of each node are printed as the node is visited, the numbers are printed in ascending order.

For convinence the above binary search tree if it is traversed inorder manner the result order is,

30, 46, 50, 58, 60, 70, 77 and 80 is ascending order sequence.

(3) Application Binary Tree – (Expression Tree)

Representing an expression containing operands and binary operators by a strictly binary tree. The root of the strictly binary tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees. A node representing an operator is a nonleaf, whereas a node representing an operand in a leaf.
BINARY SEARCH TREE OPERATIONS:

The basic operation on a binary search tree (BST) include, creating a BST, inserting an element into BST, deleting an element from BST, searching an element and printing element of BST in ascending order.

The ADT specification of a BST:

ADT BST
{
    Create BST() : Create an empty BST;
    Insert(elt) : Insert elt into the BST;
    Search(elt,x) : Search for the presence of element elt and
                    Set x=elt, return true if elt is found, else
                    return false.
    FindMin() : Find minimum element;
    FindMax() : Find maximum element;
    Ordered Output() : Output elements of BST in ascending order;
    Delete(elt,x) : Delete elt and set x = elt;
}

Inserting an element into Binary Search Tree

Algorithm InsertBST(int elt, NODE *T)
[ elt is the element to be inserted and T is the pointer to the root of the tree]

If (T = = NULL) then
    Create a one-node tree and return
Else if (elt<key) then
    InsertBST(elt, T->lchild)
Else if(elt>key) then
    InsertBST(elt, T->rchild)
Else
    "element is already exist
return T
End

C coding to Insert element into a BST

struct node
{
    int info;
    struct node *lchild;
    struct node *rchild;
};
typedef struct node NODE;
NODE *InsertBST(int elt, NODE *T)
{
    if(T = = NULL)
    {
        T = (NODE *)malloc(sizeof(NODE));
        if (T = = NULL)
printf ( "No memory error");

else
{
    t->info = elt;
    t->lchild = NULL;
    t->rchild = NULL;
}

else if (elt < T->info)
    t->lchild = InsertBST(elt, t->lchild);
else if (elt > T->info)
    t->rchild = InsertBST(elt, t->rchild);
return T;
}

Searching an element in BST

Searching an element in BST is similar to insertion operation, but they only return the pointer to the node that contains the key value or if element is not, a NULL is return:

Searching start from the root of the tree;
If the search key value is less than that in root, then the search is left subtree;
If the search key value is greater than that in root, then the search is right subtree;
This searching should continue till the node with the search key value or null pointer(end of the branch) is reached.
In case null pointer(NULL left/righ chile) is reached, it is an indication of the absence of the node.

Algorithm SearchBST(int elt, NODE *T)
[ elt is the element to be inserted and T is the pointer to the root of the tree]

1. If (T = = NULL) then
    Return NULL
2. If (elt < key) then
    /* elt is less than the key in root */
    return SearchBST(elt, T->lchild)
Else if(elt > key) then
    /* elt is greater than the key in root */
    return SearchBST(elt, T->rchild)
Else
    return T
End

NODE * SearchBST(int elt, NODE *T)
{
    if(T = = NULL)
        return NULL;
    if (elt < T->info)
        return SearchBST(elt, t->lchild);
    else if (elt > T->info)
        return SearchBST(elt, t->rchild);
    else
        return T;
}
Finding Minimum Element in a BST

Minimum element lies as the left most node in the left most branch starting from the root. To reach the node with minimum value, we need to traverse the tree from root along the left branch till we get a node with a null / empty left subtree.

Algorithm FindMin(NODE * T)

1. If Tree is null then
   return NULL;
2. If lchild(Tree) is null then
   return tree
   else
   return FindMin(T->lchild)
3. End

NODE * FindMin( NODE *T )
{
  if(T = = NULL)
    return NULL;
  if ( T->lchild = = NULL)
    return tree;
  else
    return FindMin(Tree->lchild);
}

Finding Maximum Element in a BST

Maximum element lies as the right most node in the right most branch starting from the root. To reach the node with maximum value, we need to traverse the tree from root along the right branch till we get a node with a null / empty right subtree.

Algorithm FindMax(NODE * T)

1. If Tree is null then
   return NULL;
2. If rchild(Tree) is null then
   return tree
   else
   return FindMax(T->rchild)
3. End

NODE * FindMax( NODE *T )
{
  if(T = = NULL)
    return NULL;
  if ( T->rchild = = NULL)
    return tree;
  else
    return FindMax(Tree->rchild);
DELETING AN ELEMENT IN A BST

The node to be deleted can fall into any one of the following categories:

1. Node may not have any children (ie, it is a leaf node)
2. Node may have only one child (either left / right child)
3. Node may have two children (both left and right)

Algorithm DeleteBST(int elt, NODE * T)

1. If Tree is null then
   print “Element is not found”
2. If elt is less than info(Tree) then
   locate element in left subtree and delete it
   else if elt is greater than info(Tree) then
   locate element in right subtree and delete it
   else if (both left and right child are not NULL) then     /* node with two children */
   begin
   Locate minimum element in the right subtree
   Replace elt by this value
   Delete min element in right subtree and move the remaining tree as its
   right child
   end
   else
   if leftsubtree is Null then
      /* has only right subtree or both subtree Null */
      replace node by its rchild
   else
      if right subtree is Null then
         replace node by its left child
   end
   free memory allocated to min node
   end
return Tree
End

NODE* DeleteBST(int elt, NODE * T)

NODE * minElt;
if(T == NULL)
   printf(“element not found\n”);
ext if (elt < T->info)
   T->lchild = DeleteBST(elt, T->lchild);
else if (elt > T->info)
    T->rchild = DeleteBST(elt, T->rchild);
else if(T->lchild && T->rchild)
    {
        minElt = FindMin(T->rchild);
        T->info = minElt->info;
        T->rchild = DeleteBST(T->info, T->rchild);
    }
else
    {
        minElt = Tree;
        if (T->lchild == NULL)
            T = T->rchild;
        else if (T->rchild == NULL)
            T = T->lchild;
        Free (minElt);
    }
return T;
}

**THREADED BINARY TREE:**

**Definition:** A **threaded binary tree** may be defined as follows:

A binary tree is **threaded** by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.

In binary trees, left and right child pointers of all leaf nodes (empty subtrees) are set to NULL while traversing the tree. Normally, a stack is maintained to hold the successor of the node in traversal. Replacing such NULL pointers with pointers to successor in traversal could eliminate the use of stack. These pointers are called as threads, and such a tree is called **threaded binary tree**.

If right child pointers nodes with empty right subtree, may be made to point its successor then the tree is called **right in-threaded binary trees**.

If left child pointers nodes with empty left subtree, may be made to point its successor then the tree is called **left in-threaded binary trees**.
Note: The last node in the tree not threaded. In example F is at rightmost and not threaded.

Implementing a right in-threaded binary tree:

To implement right in-threaded binary tree we need info, left, right and an extra logical field rthread is included within each node to indicate whether or not its right pointer is a thread.

Thus a node is defined as follows:
struct nodetype
{
    int info;
    struct nodetype *left;    /* pointer to left son */
    struct nodetype *right;    /* pointer to right son */
    int rthread;                 /* rthread is TURE if right is NULL (or) a non-NULL thread */
}
typedef struct nodetype *NODEPTR;

We present a routine to implement inorder traversal of a right in-threaded binary tree;

intrav2(tree)
NODEPTR tree;
{
    NODEPTR q, p;
p = tree;
do
    { q=NULL;
        while(p!=NULL)            /* traverse left branch */
        {
            q = p;
            p = p->left;
            if(q ! = NULL)
            {
                printf("%d \n", q->info);
                p = q->right;
                while(q->rthread && p!=NULL)
                {
                    printf("%d \n", p->info);
                    q = p;
                    p = p->right;
                }
            }
        }
    }
}
while(q != NULL);
}
intrav3(tree)
NODEPTR tree;
{
NODEPTR q, p;
q = NULL;
p = tree;
do
{
    while(p!=NULL) /* traverse left branch */
    {
        q = p;
p = p \rightarrow left;
    }
    if(q ! = NULL)
    {
        printf("%d \n", q\rightarrow info);
p = q\rightarrow right;
    }
    while( q ! = NULL && p == NULL )
    {
        do
        {
            /* node(q) has no right son. Back up until a left son or the tree root is encountered */

            p = q;
q = p\rightarrow father;
}while( ! isleft(p) && q ! = NULL);

    if( q ! = NULL)
    {
        printf("%d \n", q\rightarrow info);
p = q\rightarrow right;
    }
}/* end while */
} while (q ! = NULL);
}

Heterogeneous Binary Trees

Information contained in different nodes of binary tree is of different type. Such a tree is called Heterogeneous Binary Tree. To represent such a tree in C we may use a union as follows :
#define OPERATOR 0
#define OPERAND 1
struct nodetype
{
    short int utype; /* operator or operand */
union
float evalbintree (tree)
NODEPTR tree;
{
    float opnd1, opnd2;
    char symb;
    if (tree->utype == OPERAND)
        return (tree->numinfo);
    /* tree->utype == OPERATOR */
    opnd1 = evalbintree (tree->left);  /* evaluate the left subtree */
    opnd2 = evalbintree (tree->right);  /* evaluate the right subtree */

    symb = tree->chinfo;  /* extract the operator */
    /* apply the operator and return the result */
    return oper (symb, opnd1, opnd2);
}

HUFFMAN TREE

Huffman tree are used for developing an optimal encoding scheme for the given frequency of occurrence of each symbol in a message (string). Find the two symbols that appear least frequency and assign them with different bit using 0’s and 1’s. And combine these two symbols whose code represents the knowledge that a symbol is either 0 or 1 frequency. Such a way the Huffman tree is construct for remaining symbols in the given message.

Once the Huffman tree is constructed, the code of any symbol in the alphabet can be constructed by starting at the leaf representing that symbol and Climbing Up to the root. Initially the code is assigned to NULL. Each time a left branch is climbed 0 is appended to the beginning of the code; each time a right branch is climbed 1 is appended to the beginning of the code.

To read the codes from a Huffman tree, start from the root and add a '0' every time you go left to a child, and add a '1' every time you go right.

Then, repeat these steps until there is only one node left:

- Find the two nodes with the lowest weights.
- Create a parent node for these two nodes. Give this parent node a weight of the sum of the two nodes.
- Remove the two nodes from the list, and add the parent node.

A Huffman encoding tree

HUFFMAN ALGORITHM

/* initialize the set of root nodes */
rootnodes = the empty ascending priority queue;

/* construct a node for each symbol */
for(i=0; i<n; i++)
{
  p = makeTree(frequency[i]);
  position[i] = p;  /* a pointer to the leaf containing the ith symbol */
  pqinsert(rootnodes,p);
}

while ( rootnodes contains more than one item)
{
  p1 = pqmindelete(rootnodes);
  p2 = pqmindelete(rootnodes);
  /* combine p1 and p2 as branches of a single tree */
p = maketree( info(p1) + info (p2) );
setleft(p,p1);
setright(p,p2);
pqinsert( rootnodes, p);
}

/* the tree is now constructed; use it to find codes */

root = pqmindelete(rootnodes);

for( i=0; i<n; i++)
{
    p = position [i];
    code[i] = the null bit string;
    while( p != root )
    {
        /* travel up the tree */
        if( isleft(p) )
            code[i] = 0 followed by code[i];
        else
            code[i] = 1 followed by code[i];
        p = father(p);
    }
}