

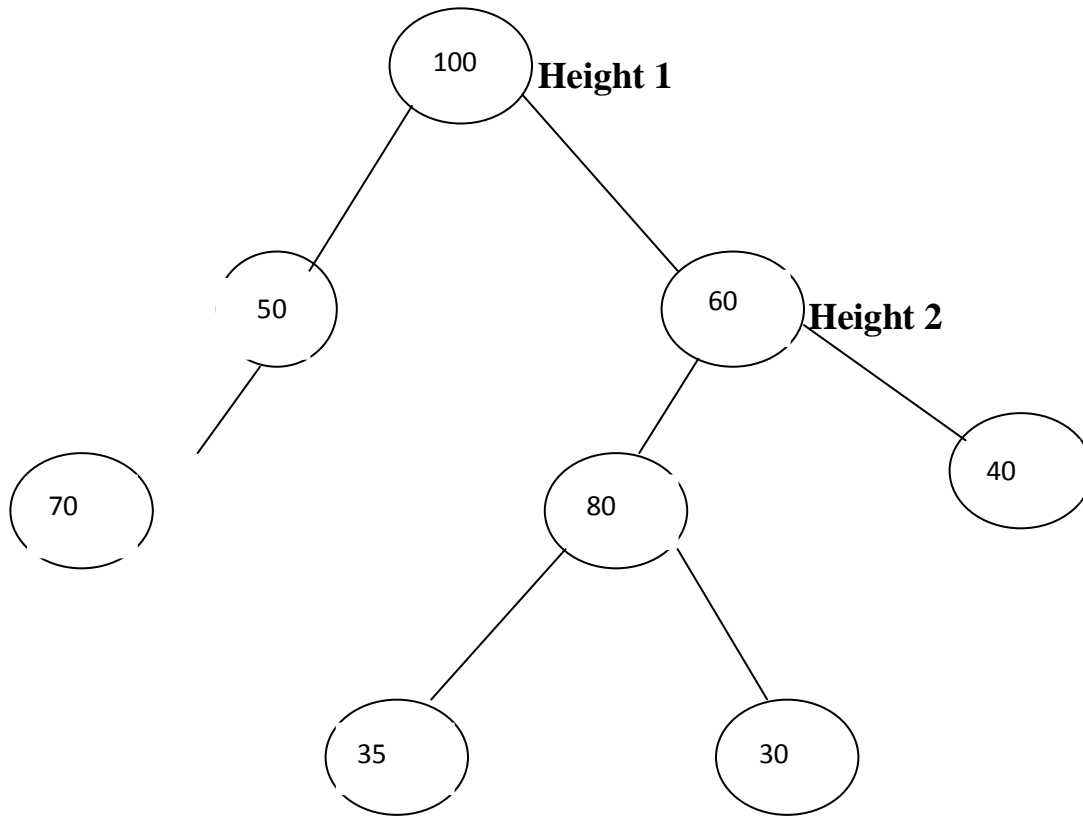
Binary Trees

Definitions

A tree is a finite set of one or more nodes that shows parent-child relationship such that

There is a special node called root

Remaining nodes are portioned into subsets $T_1, T_2, T_3 \dots T_N$ for $N > 0$ which are children of root. Consider the following tree



Terminologies

Root Node: The first node written at the top and does not have a parent

Child: Node obtained from the parent. A parent node can have zero or more child nodes

example 50 and 60 are children of 100

Siblings: Two or more nodes having the same parent

Ex 50 and 60 are siblings of 100

But 70 is NOT a sibling of 50

Ancestors: Nodes obtained in the path from a specific node X while moving upwards towards root

Ex 100 is the ancestor of 50 and 60

Descendants : Nodes in the path below the parent. ie nodes reachable from node X while moving downwards

ex all nodes below 100

Left descendent : Nodes to the left sub tree of X

ex 50 and 60 are left descendants of 100

Right descendent : Nodes to the right subtree of node X

ex right descendants of 100 are 60,80,40,35,30

Left sub tree: All nodes that are left descendants of X forms the left sub tree

ex Left sub tree of 100 is 50 and 70

Right Sub tree: All nodes that are right descendants of X is the right sub tree

ex Right sub tree of 100 is 60,80,40,35,30

Parent: A node having left sub tree or right sub tree or both is a parent for left and / or right subtree

ex parent of 50 and 60 is 100

Degree: Number of sub trees of a node

ex 100 has 2 sub trees and hence degree is 2

Leaf: A node in a tree whose degree is 0

ex 70,35,30

Internal nodes: Nodes except the leaf

ex 100,50,60

External nodes: They are leaf nodes

Level: Distance of a node from root

ex Distance of root is 0

Distance of 50 is 1

Height: Maximum level of any leaf

ex height is 4

Binary trees

A binary tree is a tree that has finite set of nodes that is either empty or consist of root and two subtrees –left and right. A binary tree has

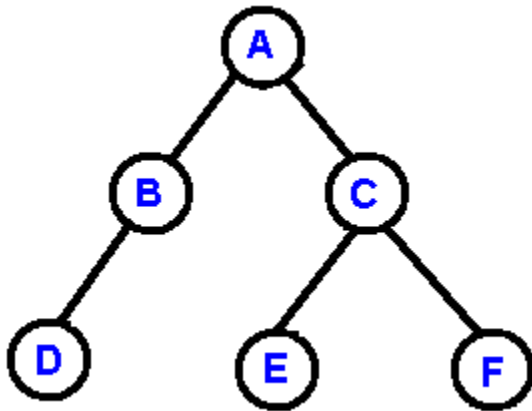
Root – If the tree is not empty then the first node is the root node

Left sub tree- It is tree connected to the left of the root

Right sub tree- It is the tree connected to the right of the root

A tree which has zero,one,or two subtrees is a binary tree

ex



Representation

Binary trees can be represented in two ways

Linked representation

Array representation

Linked representation

Here a node has 3 fields

- info – contains the actual information
- llink-contains the address of the left sub tree
- rlink- contains the address of right subtree

Hence a node can be represented as

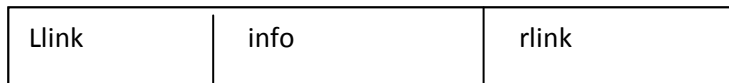
```

struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};

typedef struct node * NODE;

```

Pictorially a node is represented as



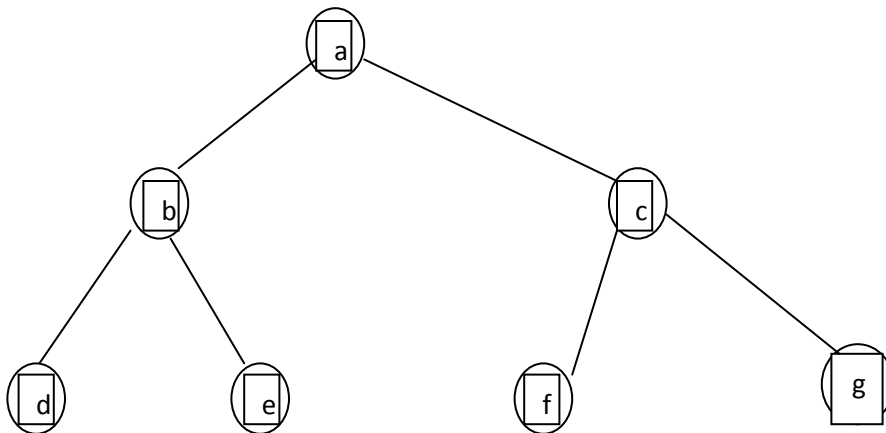
A pointer variable root can be used to point to the root.

Hence for a empty tree we have

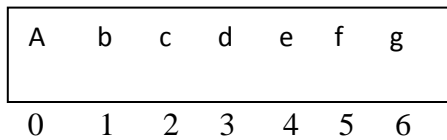
NODE root=NULL.

Array representation

A tree can be represented using a array. This method of representation is also called as sequential representation



The sequential representation of the above tree would be as follows



Note : Nodes are numbered sequentially from 0 (root)

Given the position of node I , $2*I + 1$ gives the position of the left child and $2*i+2$ gives the position of the right sub child.

If I is the position of the left child then $i+1$ gives the position of the right child

If I is the position of the right child then $i-1$ gives the position of left child

There are two ways of representing a tree using arrays

Method 1: Here some of the locations may be used and some may not be used. Hence we need to use a flag called **used which will indicate whether the location is used or not**. If the value is 0 then the location is not used and it indicates the absence of a node

Hence we have

```
#define MAX 200
```

```
struct node
```

```
{
```

```
int info;
```

```
int used;
```

```
};
```

```
typedef struct node *NODE;
```

A array of type NODE can be used now as

```
NODE a[MAX];
```

Method 2

Instead of using a separate flag field we initialize each location to 0 indicating that the node is not used. Non zero value indicates the presence of a node.

Types of binary trees

We have the following types of binary trees

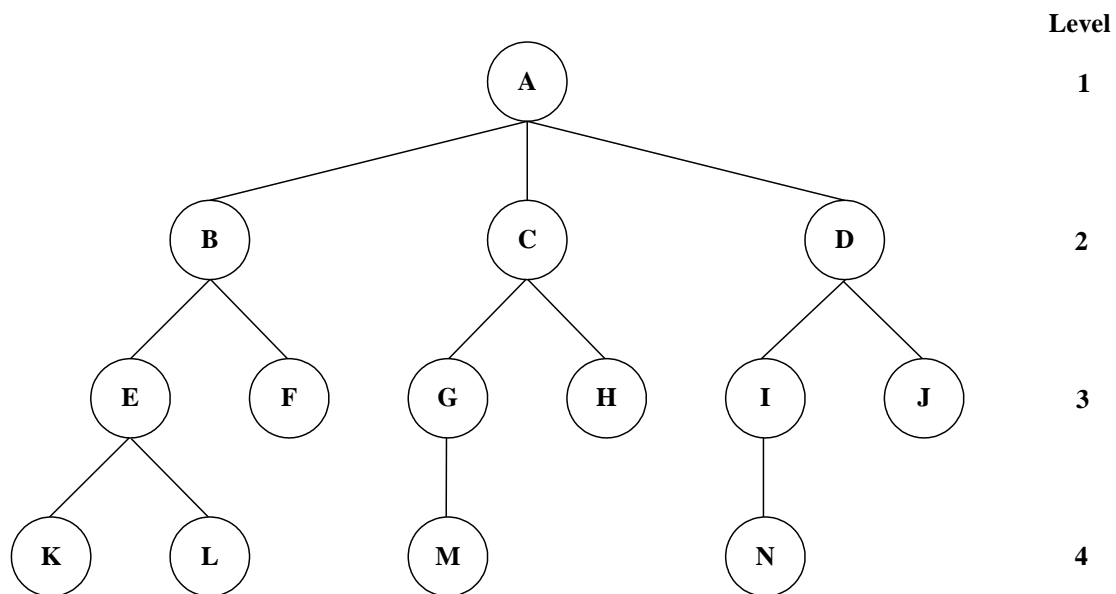
Strictly binary tree(Full binary tree)

Skewed binary tree

Complete binary tree

Expression tree

Binary search tree



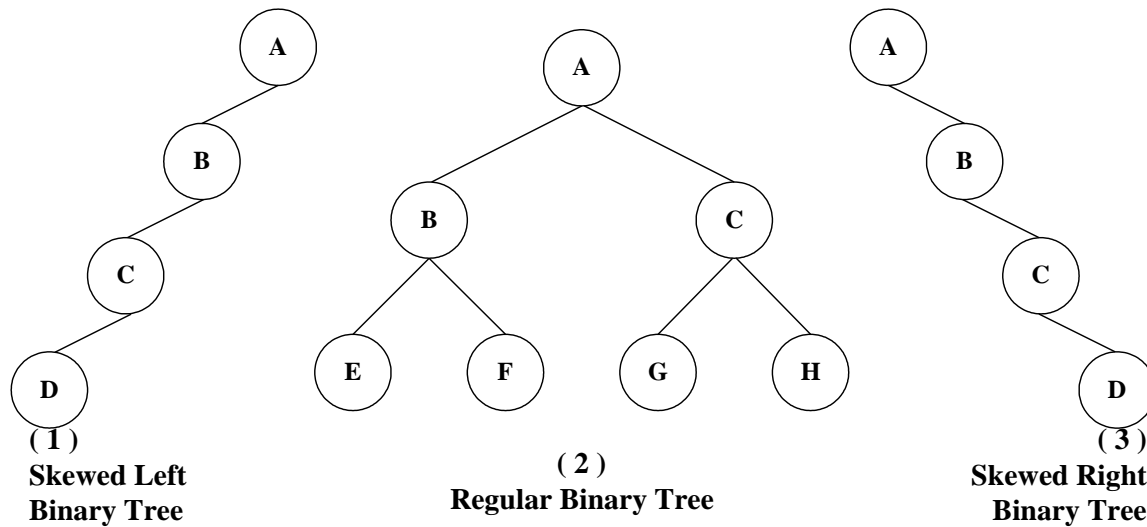
Binary Tree Types:

Regular Binary Tree (2)

Skewed Left Binary Tree (1)

Skewed Right Binary Tree (3)

Three Graphical Pictures of the Binary Tree:



Properties of Binary Trees

In particular, we want to find out the maximum number of nodes in a binary tree of depth k , and the number of leaf nodes and the number of nodes of degree two in a binary tree.

Binary Tree Traversals

There are many operations that we can perform on tree, but one that arises frequently is traversing a tree, that is, visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree.

Binary Tree Traversals Types

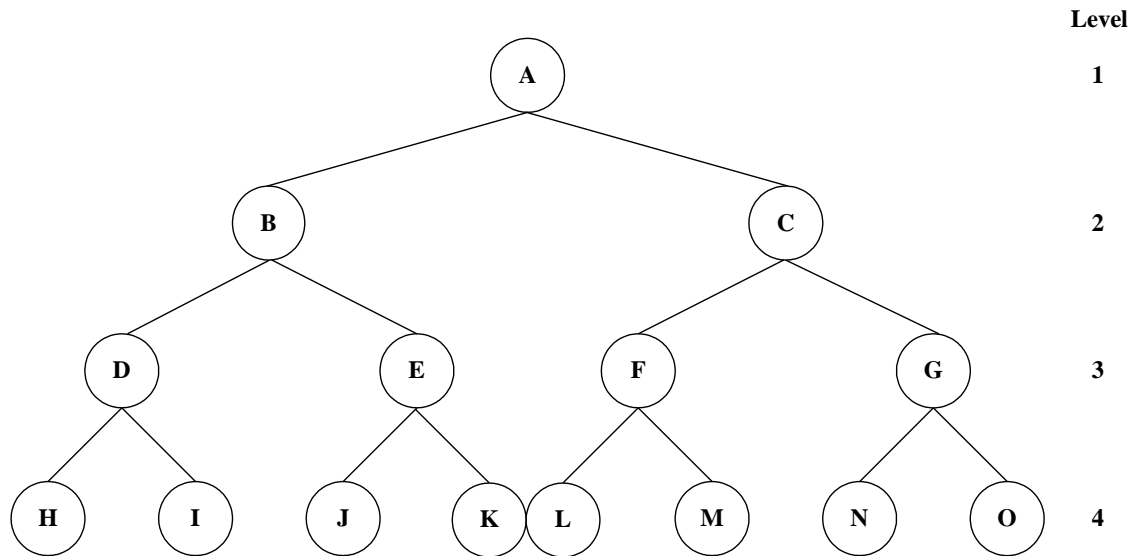
Inorder Traversal (Left, Parent, Right)

Preorder Traversal (Parent, Left, Right)

Postorder Traversal (Left, Right, Parent)

Level Order Traversal (Top to Bottom, Left to Right)

Example of the Binary Tree:



Binary Tree Traversals Functions

Inorder Tree Traversal

Recursive function:

```
void inorder (ptr_node ptr)
{
    if (ptr)
    {
        inorder (ptr->left_child);
        cout << ptr->data;
        inorder (ptr->right_child);
    }
}
```

Result of binary tree example:

H, D, I, B, J, E, K, A, L, F, M, C, N, G, O

Preorder Tree Traversal

Recursive function:

```
void preorder (ptr_node ptr)
{
    if (ptr)
    {
        cout << ptr->data;
        preorder (ptr->left_child);
        preorder (ptr->right_child);
    }
}
```

Result of binary tree example:

A, B, D, H, I, E, J, K, C, F, L, M, G, N, O

Postorder Tree Traversal

Recursive function:

```
void postorder (ptr_node ptr)
{
    if (ptr)
    {
        postorder (ptr->left_child);
        postorder (ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

```
    }  
}
```

Result of binary tree example:

H, I, D, J, K, E, B, L, M, F, N, O, G, C, A

Level Order Tree Traversal

Using queue:

```
void level_order (ptr_node ptr)  
{  
    int front = rear = 0;  
    ptr_node queue[max_queue_size];  
  
    if (!ptr) // empty tree;  
        return;  
    addq(front, &rear, ptr);  
    for ( ; ; )  
    {  
        ptr = deleteq (&front, rear);  
        if (ptr)  
        {  
            cout << ptr->data;  
            if (ptr->left_child)  
                addq (front, &rear, ptr->left_child);  
            if (ptr->right_child)  
                addq (front, &rear, ptr->right_child);  
        }  
    }  
}
```

```
    }  
    else  
        Break;  
    }  
}
```

Result of binary tree example:

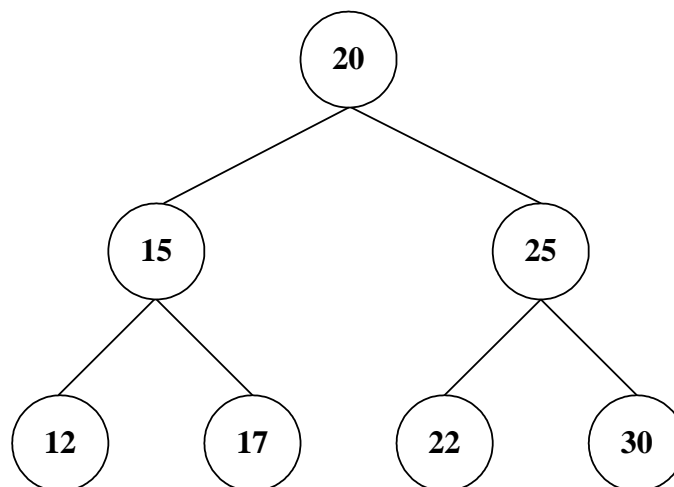
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O

Binary Search Tree

Definition: A binary search tree is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

- (1) Every element has a key, and no two elements have the same key, that is, the keys are unique.
- (2) The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
- (3) The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
- (4) The left and right subtrees are also binary search trees.

Example of the Binary Search Tree:



Searching A Binary Search Tree

Suppose we wish to search for an element with a key. We begin at the root. If the root is NULL, the search tree contains no elements and the search is unsuccessful. Otherwise, we compare key with the key value in root. If key equals root's key value, then the search terminates successfully. If key is less than root's key value, then no elements in the right subtree subtree can have a key value equal to key. Therefore, we search the left subtree of root. If key is larger than root's key value, we search the right subtree of root.

Recursive Function for Binary Search Tree:

```
tree_ptr search ( tree_ptr root, int key )
{
    if ( !=root )
        return NULL;
    if ( key == root->data )
        return root;
    if ( key < root->data )
        return search ( root->left_child, key );
    return search ( root->right_child, key );
}
```

Inserting Into A Binary Search Tree

To insert a new element, key, we must first verify that the key is different from those of existing elements. To do this we search the tree. If the search is unsuccessful, then we insert the element at the point the search terminated.

Insert Function:

```
void insert_node ( tree_ptr *node, int num )
{
    tree_ptr ptr, temp = modified_search ( *node, num ); // **
    if ( temp || ! ( *node ) )
    {
        ptr = new node;
        if ( ptr == NULL )
        {
            cout << "The memory is full \n";
            exit ( 1 );
        }
        ptr->data = num;
        ptr->left_child = ptr->right_child = NULL;
        if ( *node )
        {
            if ( num < temp->data )
                temp->left_child = ptr;
            else
                temp->right_child = ptr;
        }
        else
            *node = ptr;
    }
}
```

Deletion from a Binary Search Tree

- Deletion of a leaf node is easy. For example, if a leaf node is left child, we set the left child field of its parent to NULL and free the node.
- The deletion of a nonleaf node that has only a single child is also easy. We erase the node and then place the single child in the place of the erased node.
- When we delete a nonleaf node with two children, we replace the node with either the largest element in its left subtree or the smallest elements in its right subtree. Then we proceed by deleting this replacing element from the subtree from which it was taken.

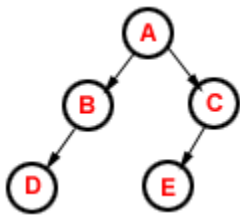
THREADED BINARY TREE

A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

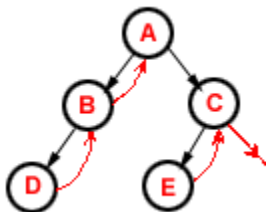
The node structure for a threaded binary tree varies a bit and its like this --

```
struct NODE
{
  struct NODE *leftchild;
  int node_value;
  struct NODE *rightchild;
  struct NODE *thread;
}
```

Let's make the Threaded Binary tree out of a normal binary tree...



The INORDER traversal for the above tree is -- D B A E C. So, the respective Threaded Binary tree will be --



B has no right child and its inorder successor is A and so a thread has been made in between them. Similarly, for D and E. C has no right child but it has no inorder successor even, so it has a hanging thread

A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

The node structure for a threaded binary tree varies a bit and its like this --

```
struct NODE
{
struct NODE *leftchild;
int node_value;
struct NODE *rightchild;
struct NODE *thread;
```

Tree traversal

In computer science, tree traversal refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Depth-first traversal

To traverse a non-empty binary tree in preorder, perform the following operations recursively at each node, starting with the root node:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

To traverse a non-empty binary tree in inorder (symmetric), perform the following operations recursively at each node:

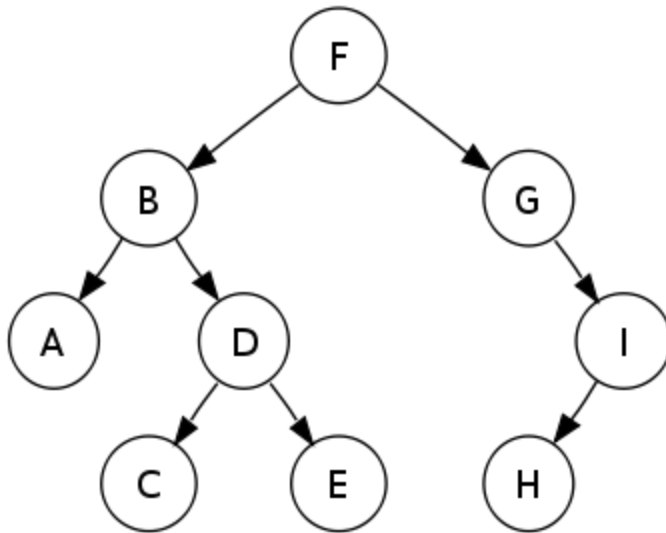
1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

To traverse a non-empty binary tree in postorder, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Breadth-first traversal

Trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level.



Depth-first


- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right); note how this produces a sorted sequence
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)


Breadth-first

- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

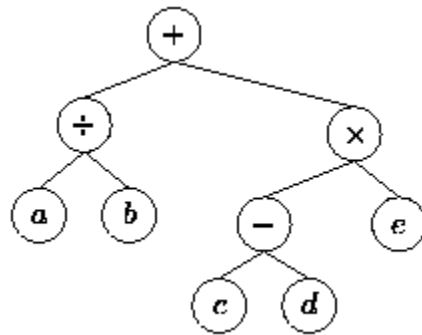
Algebraic expressions such as

$$a/b+(c-d)$$

have an inherent tree-like structure. For example, Figure  is a representation of the expression in Equation . This kind of tree is called an *expression tree* .

The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (a , b , c , d , and e). The non-terminal nodes of an expression tree are the operators ($+$, $-$, \times , and \div). Notice that the parentheses which appear in Equation  do not appear in the tree. Nevertheless,

the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.



The common algebraic operators are either unary or binary. For example, addition, subtraction, multiplication, and division are all binary operations and negation is a unary operation. Therefore, the non-terminal nodes of the corresponding expression trees have either one or two non-empty subtrees. That is, expression trees are usually binary trees.

What can we do with an expression tree? Perhaps the simplest thing to do is to print the expression represented by the tree. Notice that an inorder traversal of the tree visits the nodes in the order

$a, /, b, +, c, -, d, e$

Except for the missing parentheses, this is precisely the order in which the symbols appear in Equation .

This suggests that an *inorder* traversal should be used to print the expression. Consider an inorder traversal which, when it encounters a terminal node simply prints it out; and when it encounters a non-terminal node, does the following:

1. Print a left parenthesis; and then
2. traverse the left subtree; and then
3. print the root; and then
4. traverse the right subtree; and then
5. print a right parenthesis.

Application of trees

One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

```

  / <-- root
 /  \
...  home
     /  \
   ugrad  course
    /  / | \
...  cs101 cs112 cs113

```

- 2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays)
- 3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists).
- 4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

The following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms