

Data Structures

R. K. Ghosh

IIT Kanpur

Binary Tree Data Structures

Definition of a Node

A non-divisible unit of information of a large data structure such as a linked list, a tree, or a graph. A node would also contain links (pointers) to other nodes. The linked nodes are related in some sort of relations.

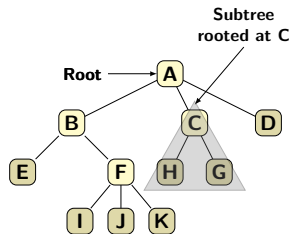
Definition of a Tree

A tree T can be empty (null tree having no node), or may consist of

- 1 A special node, designated as r , called the root.
- 2 A set of trees k trees T_1, T_2, \dots, T_k (some could possibly empty) with roots r_1, r_2, \dots, r_k respectively.

T is constructed by making r_1, r_2, \dots, r_k as children of r . Tree T_1, T_2, \dots, T_k are called subtrees of T .

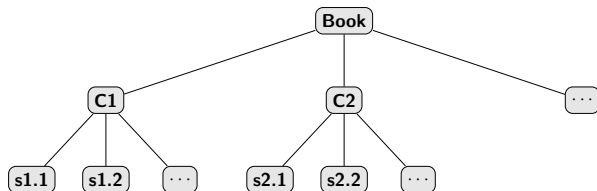
Tree



Farthest leaf nodes: I, J, K
Leaf node: D, E, G, H, I, J, K
Depth of I, J, K: 3
Height of tree: 3

Terminology

- ▶ Type of nodes: **internal** or **leaf**.
- ▶ A is an **ancestor** of all nodes including itself.
- ▶ All nodes including A are **descendants** of A.
- ▶ **Ordered tree**: **siblings** are ordered from left to right.
- ▶ Degree of a tree is k : if no node has more than k children.



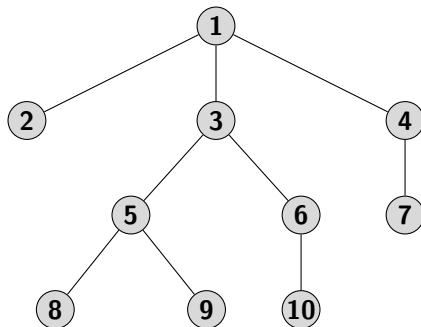
Tree Path

- ▶ Above figure depicts is an example of a tree.
- ▶ Parent-child relations shown by lines.
- ▶ A path: n_1, n_2, \dots, n_k , such that $n_i = \text{parent}(n_{i-1})$.
- ▶ Length of a path is 1 less than number of nodes.

Ordered Trees

- ▶ Ordered trees have special importance.
- ▶ Ordered trees are presented by drawing on a plane.
- ▶ Child nodes are listed using some specific order (counter clockwise).
- ▶ If children of each node is placed at a given distance down.
- ▶ Root is placed on the top.

Example of an Order Tree



- ▶ Children of node 1: 2-first child, 3-second child, 4-third child.
- ▶ Sometimes ordering is defined by left-to-right.

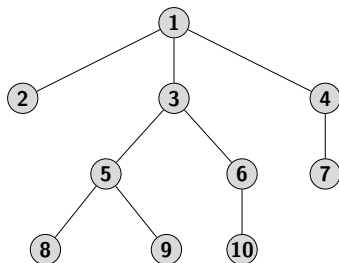
Tree Traversal

- ▶ We can systematically order nodes of a tree in many ways.
- ▶ Three most important ordering are: Preorder, Postorder and Inorder.
- ▶ Recursive definition of these orderings are as follows:
 - If a tree T is empty then empty list is the preorder, postorder and inorder listing of T .
 - If T consist of only one node, then the node by itself is the listing in all three orderings.
- ▶ Otherwise, let T be a tree with root r and k subtrees T_1, T_2, \dots, T_k .

Tree Traversal

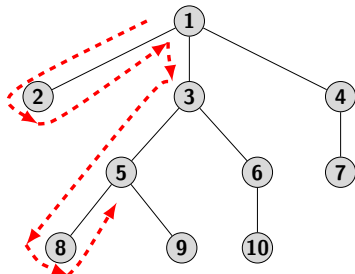
- ▶ Preorder listing of T : list root r of T , preorder list of all subtree T_1, T_2, \dots, T_k in left to right order.
- ▶ Postorder listing of T : postorder list of all subtree T_1, T_2, \dots, T_k in left to right order followed by root r of T .
- ▶ Inorder listing of T : inorder listing of T_1 followed by root r and then inorder listing of each group of nodes T_2, T_3, \dots, T_k in inorder.

Tree Traversal



- ▶ Preorder listing: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.
- ▶ Postorder listing: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1.
- ▶ Inorder listing: 2, 1, 8, 5, 9, 3, 10, 6, 7, 4.

Walk Around the Tree



Euler Tour

- ▶ A walk around the tree treating edges as walls.
- ▶ Start the walk outside the tree, starting at the root,
- ▶ Stay as close to the tree as possible.
- ▶ Move anti clockwise till reaching back to the root.

Tree Traversal Terminology

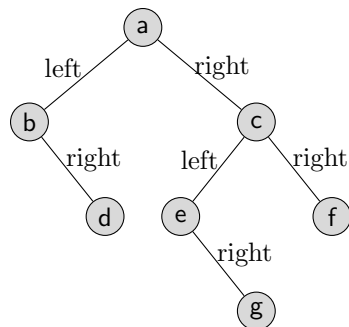


- ▶ Every node v is visited three times in the walk around a tree.
 - First time on the left before walking around left subtree of v .
 - Second time from below after having traversed around all nodes of left subtree of v .
 - Third time on the right after having traversed the right subtree of v .
- ▶ For distinguishing the three different visits to a node, use following terminology:
 - First time (before the Euler tour of v ' left subtree)
 - Second time (between the Euler tours of v ' two subtrees)
 - Third time (after the Euler tour of v ' right subtree)

Tree Traversal Terminology

- ▶ For preorder traversal: list the node when it is visited for the "first time" during Euler tour.
 - Listing will be: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.
- ▶ For inorder traversal: is not defined for general tree but only for binary tree. However, it may sometimes defined as visiting a node "second time" during Euler traversal.
 - Listing will be: 2, 1, 8, 5, 9, 3, 10, 6, 7, 4.
- ▶
- ▶ For postorder traversal: list the node when it is visited "third time" during Euler traversal.
 - Listing will be: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1.

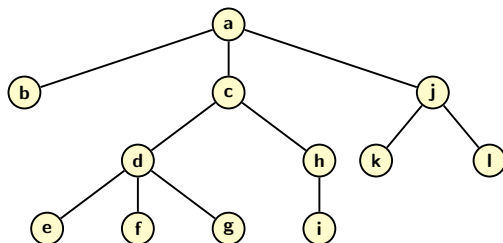
Binary Trees



Binary Trees

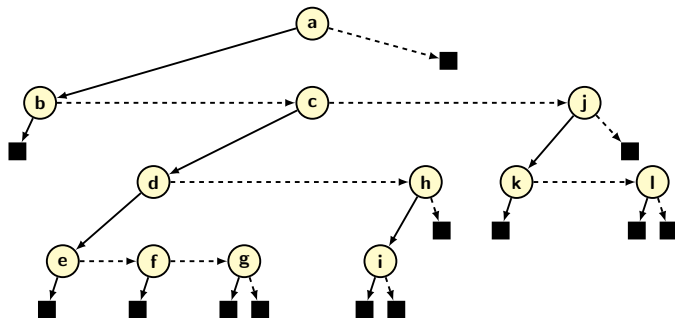
- ▶ If arity $k = 2$, then we have a binary tree.
- ▶ We distinguish between two children as left and right.
- ▶ Pictorial convention is to draw left child extended to the left and right child to right.

General Tree as Binary Tree



- ▶ Leftmost child is considered as the left child.
- ▶ A non empty right sibling of a node becomes as its right child.

Equivalent Extended Binary Tree

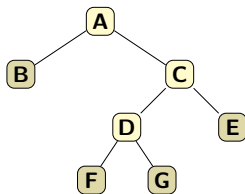


- ▶ Only internal nodes have information.
- ▶ All leaf nodes are external (null) nodes indicated by black squares.

Traversal of Equivalent Binary Tree

- ▶ When a general tree is represented as a binary tree, algorithms binary trees can be used process the general tree.
- ▶ But, inorder traversal of the equivalent binary tree does not make any sense.
- ▶ In a general tree a node may have more than two children.
- ▶ So, inserting a visit to parent node between children is difficult specially when there are odd number of children.

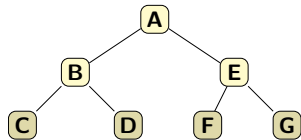
Important Binary Trees



A strictly Binary tree

Strictly Binary Tree

In a strictly binary tree, every internal node has two children.

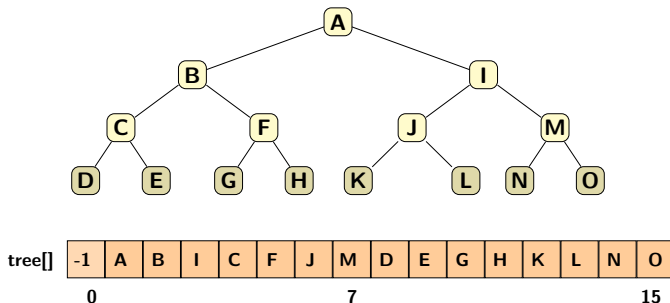


A Full Binary tree

A Full Binary Tree

A full binary tree is a strictly binary tree in which all leaves appear at the bottom most level.

Array Representation

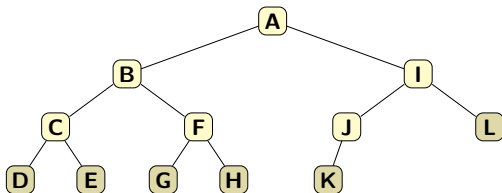


- ▶ Children of node i are at $2i$ and $2i + 1$.
- ▶ All nodes except node 1 has a parent.
- ▶ Array representation also works for complete binary tree.
 - Let child of i is at $2i$ unless $2i > n$.
 - If $2i > n$ then node i has no left child.

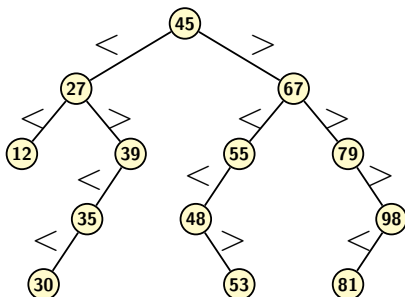
Important Binary Trees

A Complete Binary Tree

A complete binary of height h consists of a full binary tree of height $h - 1$, in which all internal nodes except at most one at height $h - 1$ have two children and appear as far left as possible.



Important Binary Trees



Binary Search Tree

- ▶ All key values stored at any node belonging to left subtree is less than the key stored at the root.
- ▶ All key values stored at any node belonging to right subtree is greater than the key stored at the root.
- ▶ The above two properties hold at any internal node.

Properties of Binary Trees

- ▶ It is assumed height an empty tree is undefined.
- ▶ The height of a tree with one node is 0.
- ▶ A tree branch contributes 1 to height.
- ▶ A non empty binary tree with height $h \geq 0$, has at least $h + 1$ nodes and at most $2^{h+1} - 1$ nodes.
- ▶ The height of a binary tree with n nodes is at most $n - 1$ and at least $\lceil \log_2(n + 1) \rceil - 1$.
 - Let h be the height, then $h + 1 \leq n \leq 2^{h+1} - 1$.
 - Implies that $\log(n + 1) \leq h + 1$.

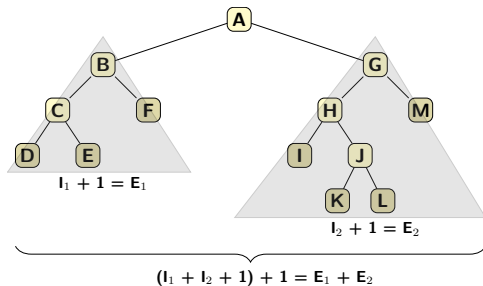
Note: You may assume either the number of nodes or the number edges not both.

Properties of Binary Trees

Leaf nodes and Internal Nodes

Number of leaf nodes in a strictly binary tree is one more than the number internal nodes.

- ▶ Every internal node has 2 children.
- ▶ A tree of height 2 has $I = 1$, $E = 2$.
- ▶ A height 3 has $I = 2$, $E = 3$ or $I = 3$, $E = 4$.

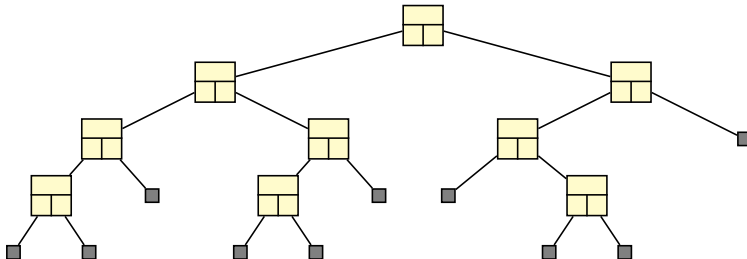


Common Operations on ADT Tree

- ▶ **newTree()**: Create a new empty tree.
- ▶ **findRoot()**: Given a tree returns its root.
- ▶ **isRoot()**: Returns true if the given node is the root.
- ▶ **findParent()**: Given a node returns its parent.
- ▶ **children()**: Given a node returns the list of its children.
- ▶ **isInternal()**: Returns true if the given node is an internal node.
- ▶ **isLeaf()**: Returns true if the given node is a leaf node.
- ▶ **exchangeNode()**: Given two nodes swaps the information held by them.
- ▶ **replaceValue()**: Given a node and a value replaces old value by the new value.

Linked List Implementation

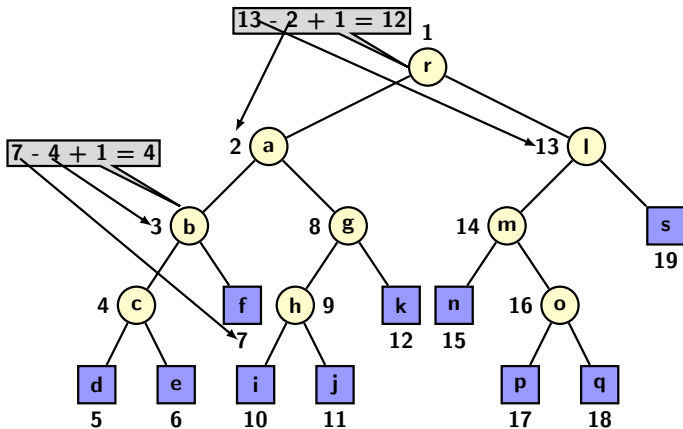
```
typedef struct treenode NODE;  
    struct treenode{  
        int info;  
        treenode *left;  
        treenode *right;  
    } TreeNode;
```



Tree Traversal

- ▶ The walk around the tree or "Euler tour" besides generating three traversals, can also be used other types of traversals.
 - Initialize a counter to 0 to start Euler tour.
 - Increment counter first time a node is visited.
 - In $C_{left}(v)$ store the counter value before traversing v 's left subtree.
 - In $C_{right}(v)$ store the counter value after traversing right subtree v .
 - Find the difference $C_{right}(v) - C_{left}(v)$ and add 1.
- ▶ The above value gives the number of descendants of v .

Descendant Computation



Preorder Traversal

```
void preorder(NODE *t) {  
    if (t!=NULL) {  
        printf("%d\t",t->data); // visit the root  
        preorder(t->left);      // left subtree  
        preorder(t->right);     // right subtree  
    }  
}
```

- Postorder and inorder traversals can be also be performed in similar way.

Creating Tree

```
NODE *create() {  
    NODE *p;  
    int x;  
    printf("Enter data(-1 for no data):");  
    scanf("%d",&x);  
    if(x== -1)  
        return NULL;  
    p=(NODE *)malloc(sizeof(NODE));  
    p->data=x;  
    printf("Enter left child of %d:\n",x);  
    p->left=create();  
    printf("Enter right child of %d:\n",x);  
    p->right=create();  
    return p;  
}
```

Membership Search

```
NODE * search(NODE *t, int x) {  
    NODE *p;  
    if ((t == NULL) || (t->data == x))  
        return t;  
    p = search(t->left, x);  
    if (p != NULL)  
        return p;  
    p = search(t->right, x);  
    if (p != NULL)  
        return p;  
    else  
        return NULL;  
}
```

Number of Descendants

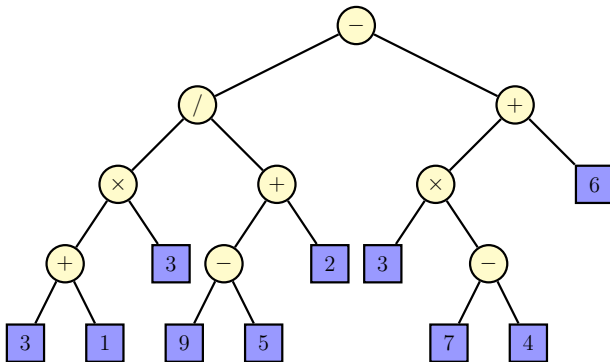
- ▶ First find the pointer to element in the tree.
- ▶ Then use Euler tour to compute number of descendants.

```
int EulerTour(NODE *t, int x) {  
    int no_descendants = 0; // Initialization  
    if (t==NULL)  
        return no_descendants;  
    else {  
        counter++;  
        if (isInternal(t)) {  
            no_descendants += EulerTour(t->left, x);  
            no_descendants += EulerTour(t->right, x);  
        }  
        return no_descendants;  
    }  
}
```

Printing Expression

```
Algorithm PrintExpression( $T, v$ ) {  
    if ( $T.isExternal(v)$ )  
        print "value" stored in  $v$ ;  
    else {  
        print "(";  
        PrintExpression( $T, T.leftChild(v)$ );  
        print "operator" stored in  $v$   
        PrintExpression( $T, T.rightChild(v)$ );  
        print ")";  
    }  
}
```


Example of Printing Expression



- ▶ **Infix:** ((((((3+1) * 3)))/((9-5)+2))+((3 * (7-4)) + 6))
- ▶ **Prefix:** - / * + 3 1 3 + - 9 5 2 + * 3 - 7 4 6
- ▶ **Postfix:** 3 1 + 3 * 9 5 - 2 + / 3 7 4 - * 6 + -