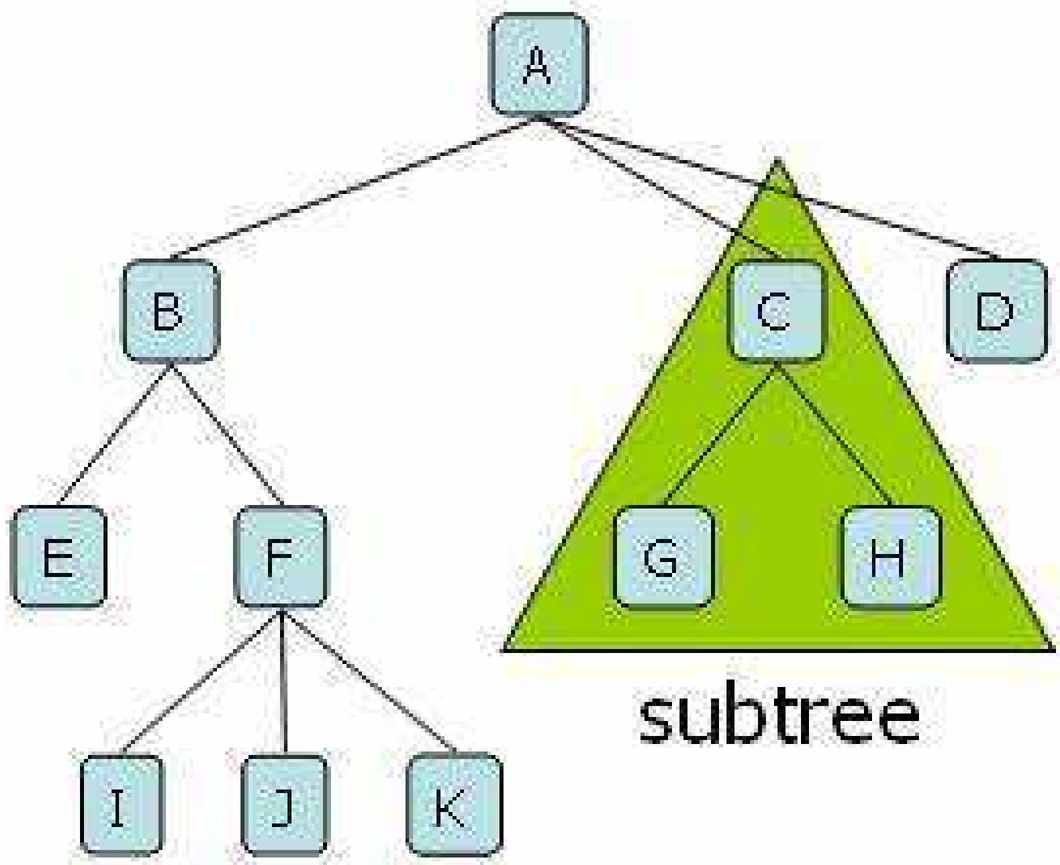
TREES

TREES

- A tree is a nonlinear data structure that is based on **hierarchical** tree structure with sets of nodes.
- A tree is a **acyclic connected graph** with zero or more children nodes and at most one parent nodes.
- A data structure accessed beginning at the **root** node.

- Each node is either a **leaf** or an **internal** node.
- An internal node has one or more child nodes and is called the **parent** of its child nodes.
- All children of the same node are **siblings**.

TREE



TREE TERMINOLOGY

- ***Root***: node without parent (A)
- ***Internal node***: node with at least one child (A, B, C, F)
- ***External node:*** (a.k.a. leaf) node without children (E, I, J, K, G, H, D)
- ***Ancestors of a node***: parent, grandparent, grand-grandparent, etc
- ***Depth of a node***: number of ancestors
- ***Height of a tree***: maximum depth of any node (3)

TREE TERMINOLOGY CONT.

- ***Descendant of a node***: child, grandchild, grand-grandchild, etc
- ***Degree of an element***: no. of children it has
- ***Subtree***: tree consisting of a node and its descendants.
- ***Path***: traversal from node to node along the edges that results in a sequence
- ***Root***: node at the top of the tree

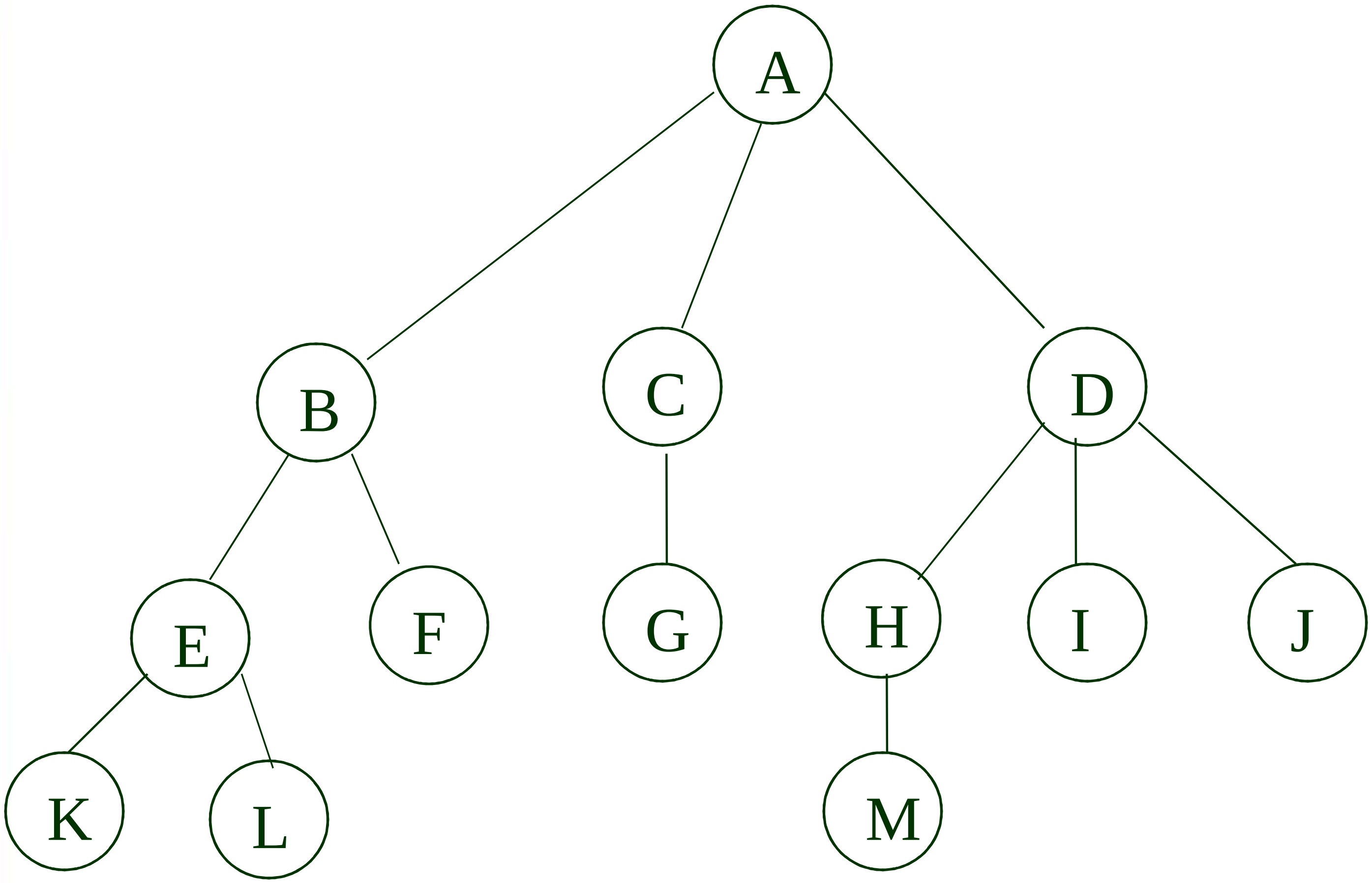
TREE TERMINOLOGY CONT.

- ***Parent***: any node, except root has exactly one edge running upward to another node. The node above it is called parent.
- ***Child***: any node may have one or more lines running downward to other nodes. Nodes below are children.
- ***Leaf***: a node that has no children
- ***Subtree***: any node can be considered to be the root of a subtree, which consists of its children and its children's children and so on.

TREE TERMINOLOGY CONT.

- ***Visiting***: a node is visited when program control arrives at the node, usually for processing.
- ***Traversing***: to traverse a tree means to visit all the nodes in some specified order.
- ***Levels***: the level of a particular node refers to how many generations the node is from the root. Root is assumed to be level 0.
- ***Keys***: key value is used to search for the item or perform other operations on it.
- ***Forest***: A set of n>= 0 disjoint trees.

GENERAL TREE



REPRESENTATION OF TREES

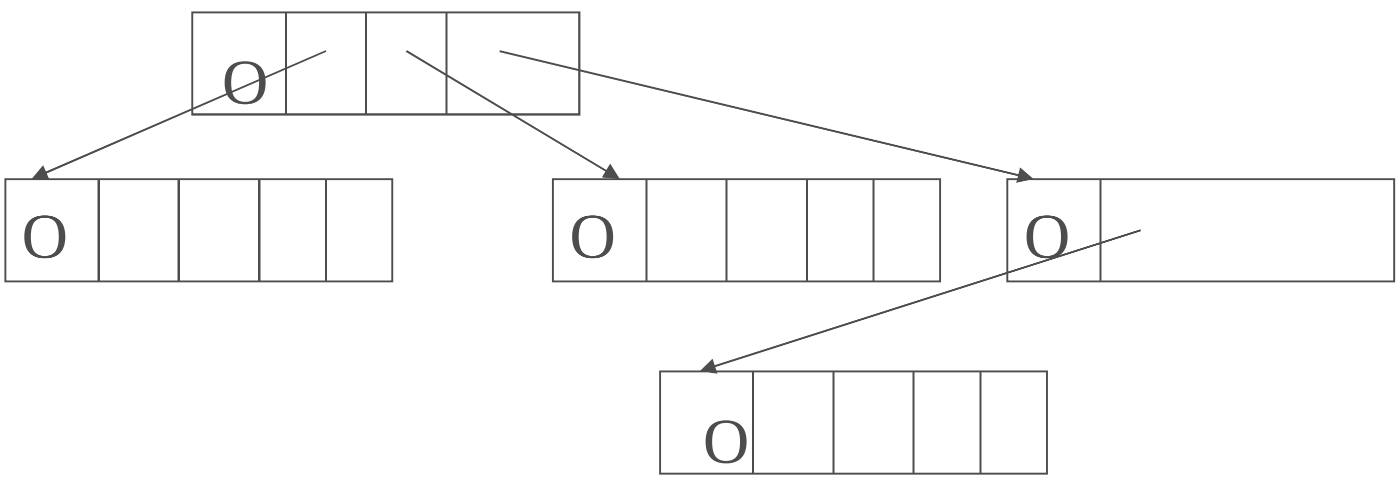
- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link n

How many link fields are
needed in such a representation?

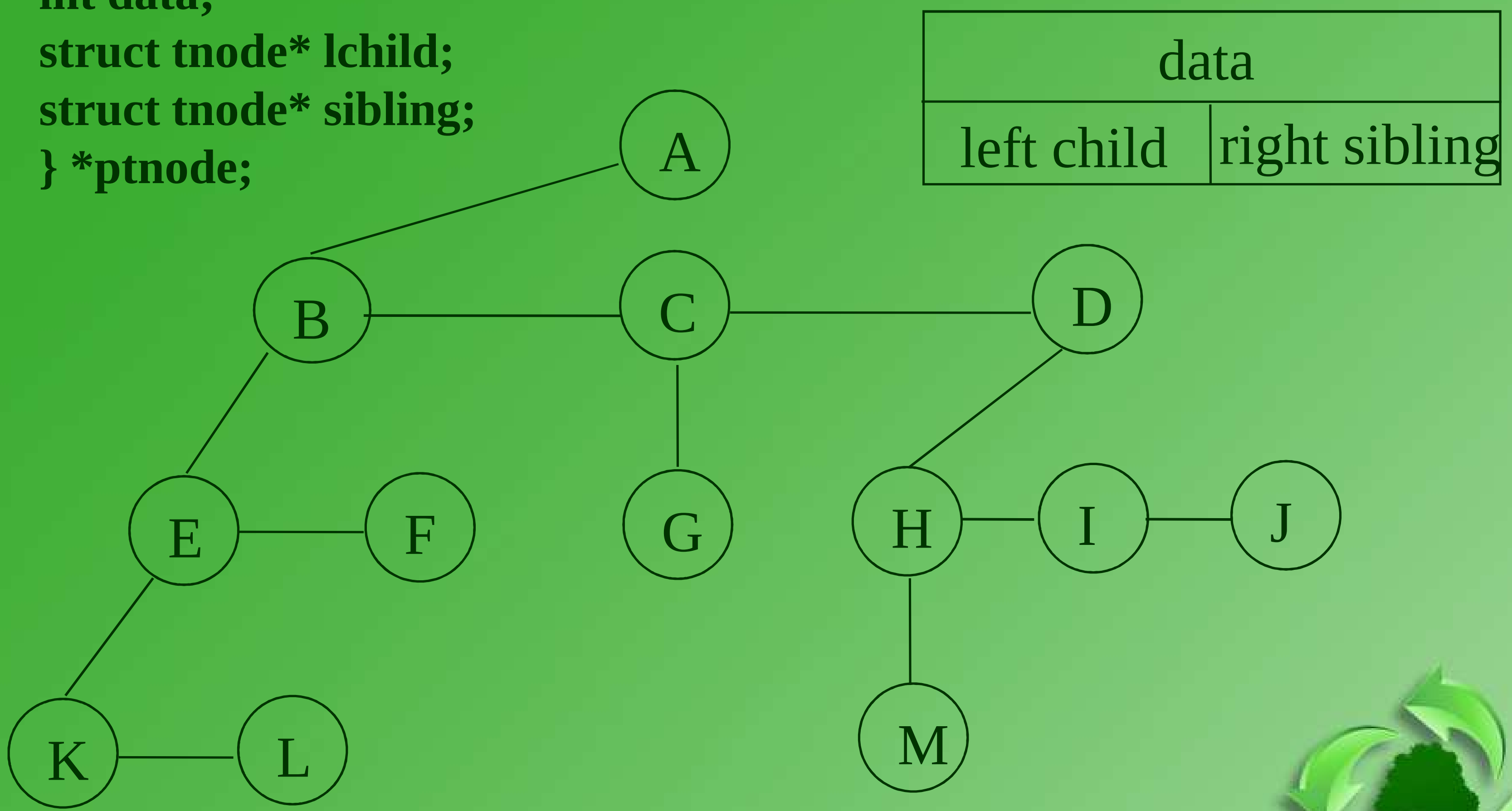
A TREE NODE

- Every tree node:
  - object – useful information
  - children – pointers to its children nodes



LEFT CHILD - RIGHT SIBLING

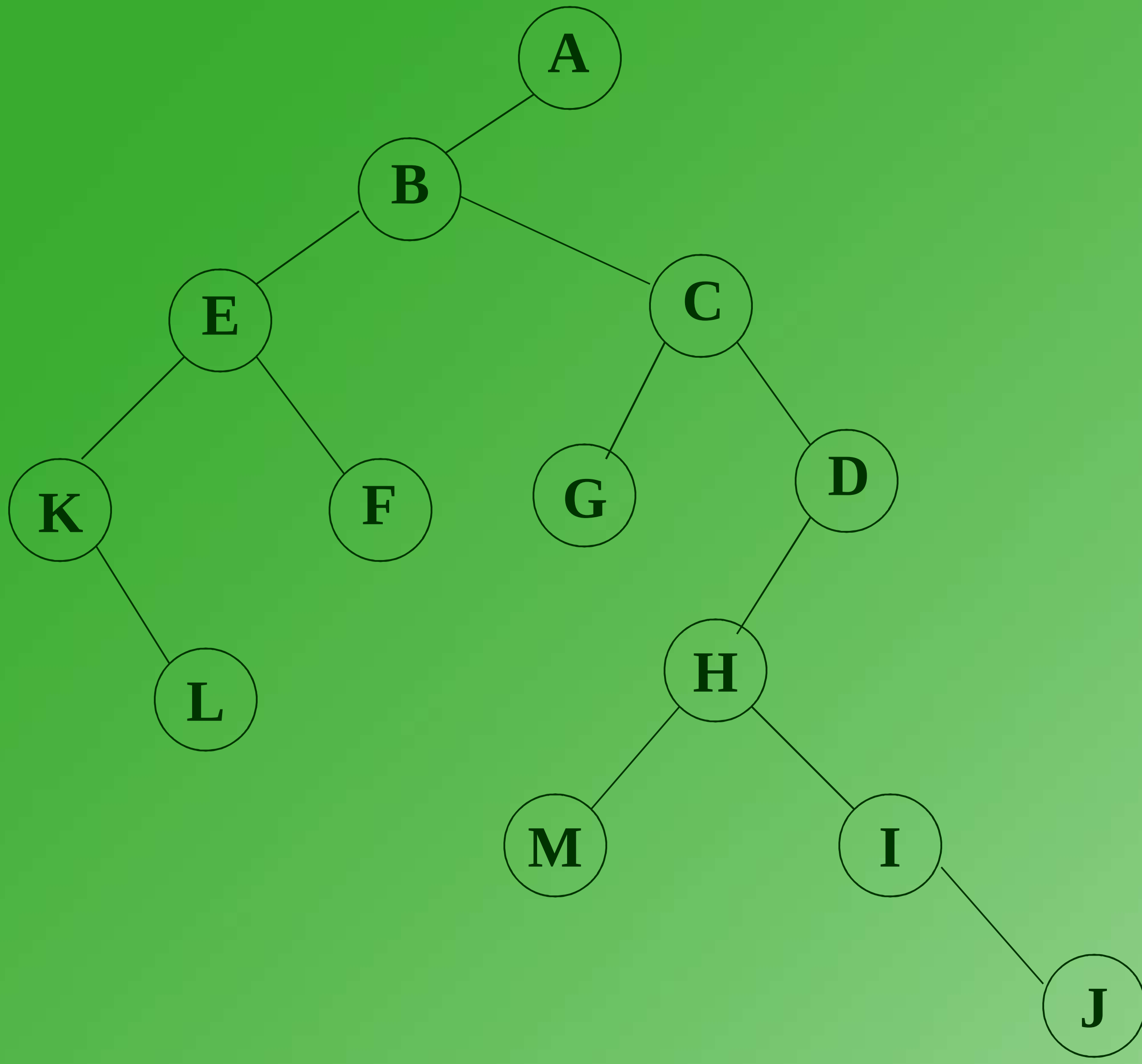
```
typedef struct tnode {
int data;
struct tnode* lchild;
struct tnode* sibling;
} *ptnode;
```



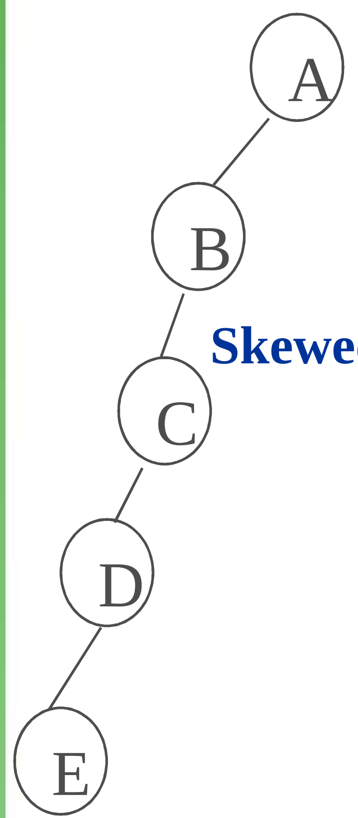
BINARY TREES

- A special class of trees: max degree for each node is 2
- Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation

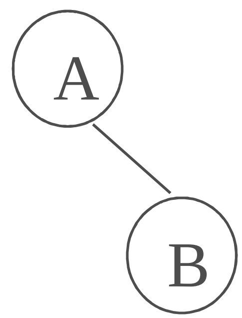
EXAMPLE



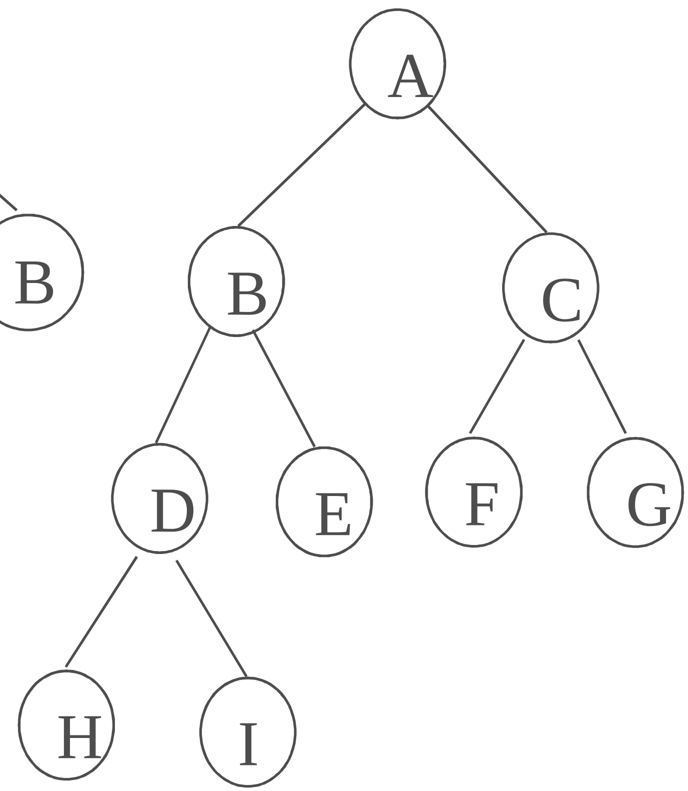
SAMPLES OF TREES



Skewed Binary Tree



Complete Binary Tree



MAXIMUM NUMBER OF NODES IN BT

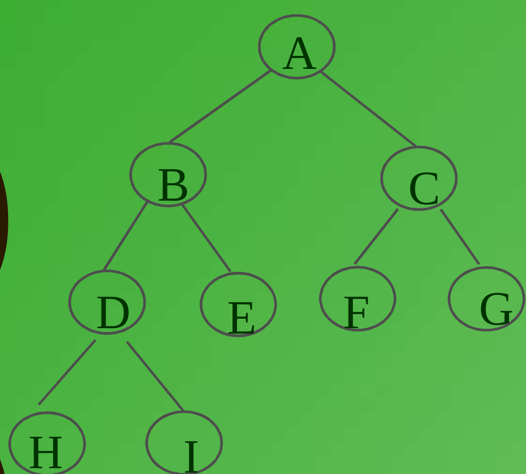
- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

RELATIONS BETWEEN NUMBER OF LEAF NODES AND NODES OF DEGREE 2

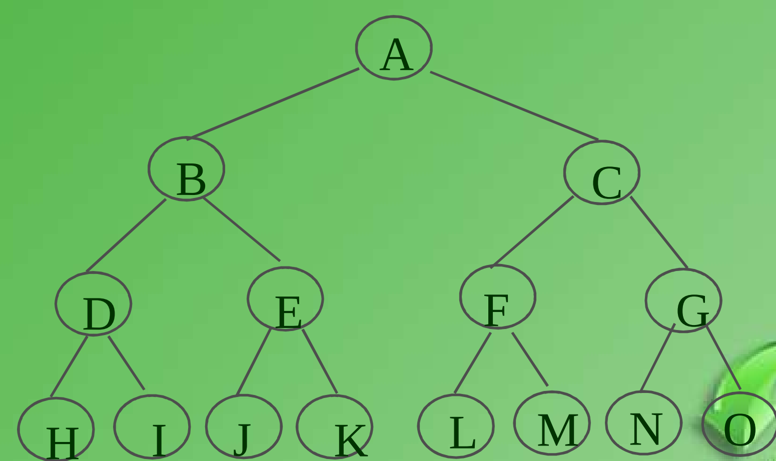
- For any nonempty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0=n2+1

FULL BT VS. COMPLETE BT

- A full binary tree of depth *k* is a binary tree of depth *k* having $2^k - 1$ nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.



Complete binary tree



Full binary tree of depth 4

BINARY TREE REPRESENTATIONS

- A binary tree can be represented using two methods:
  - Sequential array representation
  - Linked list representation

Sequential Representation

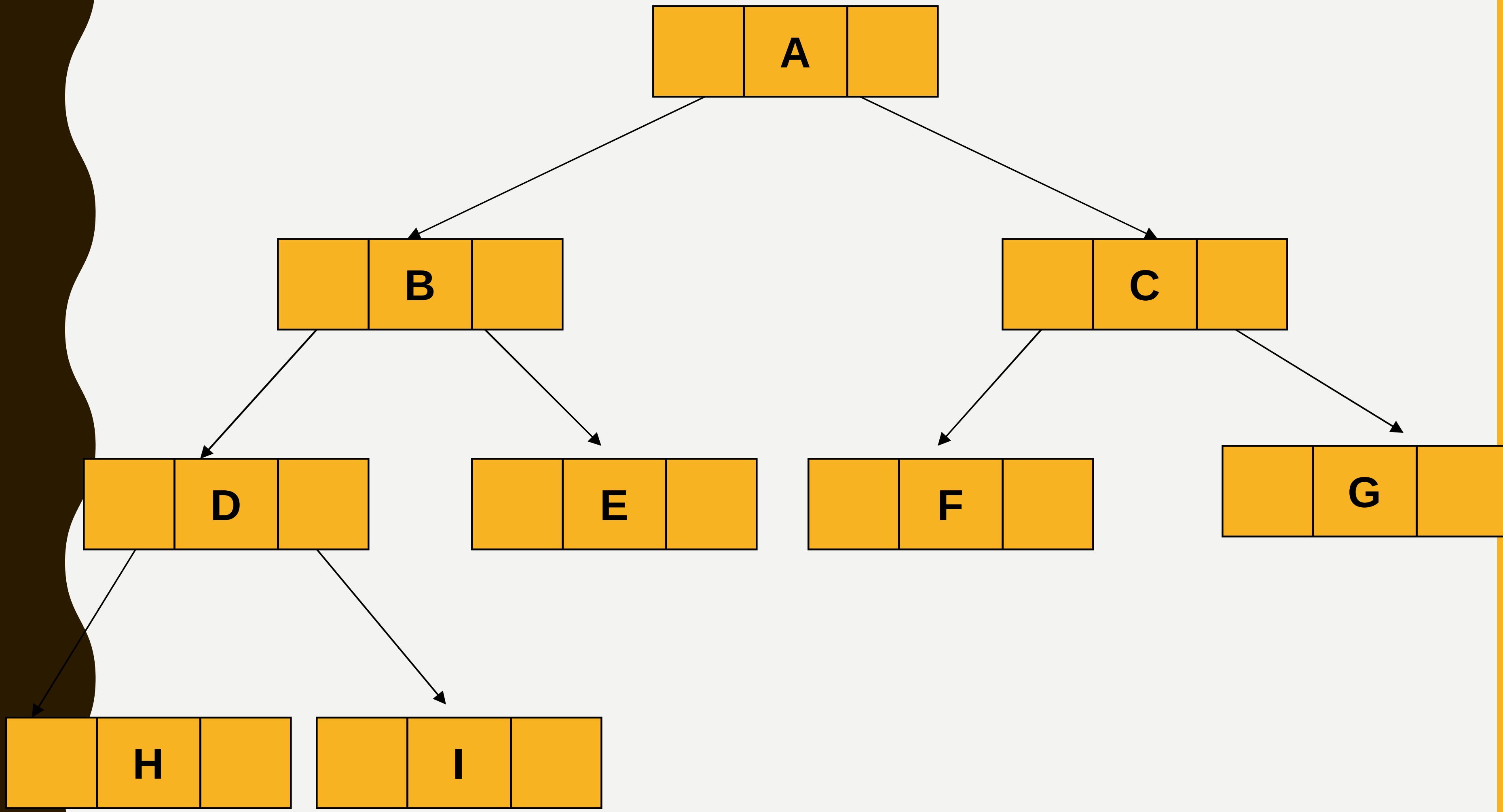
1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I

Parent (i) = i/2

Left child (i) = 2i

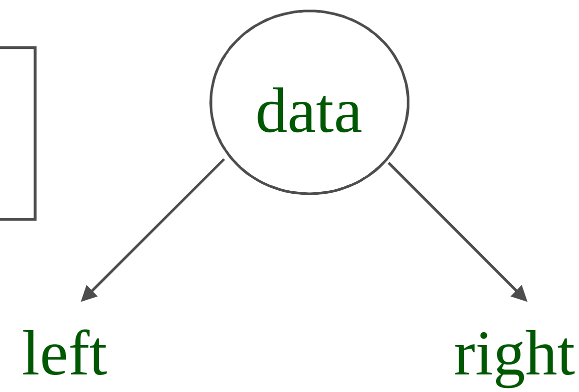
Right child (i) = 2i + 1

Linked List Representation



LINKED REPRESENTATION

```
typedef struct tnode *ptnode;
typedef struct tnode {
 int data;
 ptnode left, right;
};
```



OPERATION ON BINARY TREE

- Traversal / Display
- Insertion
- Deletion

TREE TRAVERSALS

- A binary tree is defined recursively: it consists of a **root**, a **left subtree**, and a **right subtree**

To **traverse** (or **walk**) the binary tree is to visit each node in the binary tree exactly once

Tree traversals are naturally recursive

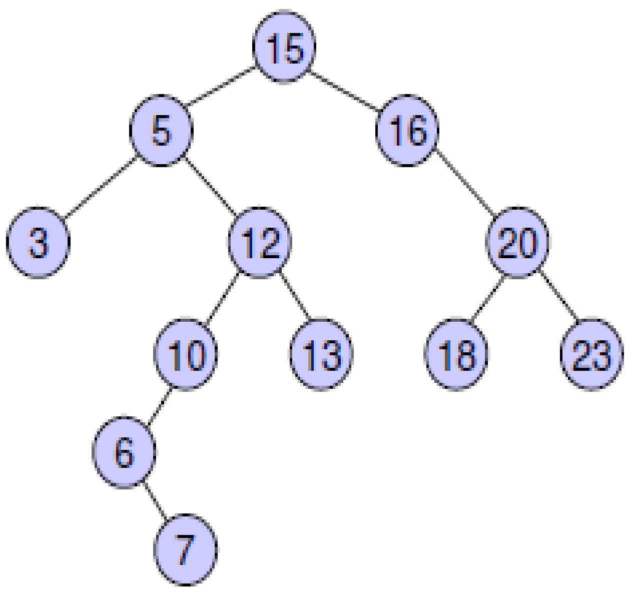
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

BINARY TREE TRAVERSAL

- **INORDER**
  - Traverse the left sub-tree in inorder.
  - Visit the node.
  - Traverse the right sub-tree in inorder
- **POSTORDER**
  - Traverse the left sub-tree in postorder.
  - Traverse the right sub-tree in postorder
  - Visit the node.
- **PREORDER**
  - Visit the node.
  - Traverse the left sub-tree in preorder.
  - Traverse the right sub-tree in preorder.

Tree Traversal Example

- Let's do an example first...



- in-order: (left, root, right)
  3, 5, 6, 7, 10, 12, 13,
  15, 16, 18, 20, 23
- pre-order: (root, left, right)
  15, 5, 3, 12, 10, 6, 7,
  13, 16, 20, 18, 23
- post-order: (left, right, root)
  3, 7, 6, 10, 13, 12, 5,
  18, 23, 20, 16, 15

INORDER TRAVERSAL

```
void inorder(ptnode ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left);
        printf(“%d”, ptr->data);
        indorder(ptr->right);
    }
}
```

PREORDER TRAVERSAL

```
void preorder(ptnode ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf(“%d”, ptr->data);
        preorder(ptr->left);
        predorder(ptr->right);
    }
}
```

POSTORDER TRAVERSAL

```
void postorder(ptnode ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left);
        postdorder(ptr->right);
        printf(“%d”, ptr->data);
    }
}
```

NON RECURSIVE INORDER TRAVERSAL

In this method we use stack explicitly.

Algorithm NR_INORDER (ptr)

Steps:

1. TOP=0
2. While (TOP != -1 || ptr != NULL)
    1. If (ptr != NULL)
        1. STACK [++TOP] = ptr
        2. Ptr = ptr->LCHILD
    2. Else
        1. Ptr = STACK[TOP--]
        2. Print ptr->DATA
        3. Ptr = ptr->RCHILD
    3. End if
3. End while
4. End

NON-RECURSIVE PRE-ORDER TRAVERSAL

Initially push NULL onto STACK and then set PTR=ROOT. Then repeat the following steps until PTR=NULL or equivalently while PTR <> NULL

a. Proceed down the left-most path rooted at PTR, processing each node N on the path and pushing each right child R(N), if any, onto STACK. The traversing ends after a node N with no left child L(N) is processed.

b. [Backtracking]Pop and assign to PTR the top element on STACK. If PTR<>NULL, then return to Step (a); otherwise Exit.

PRE-ORDER TRAVERSAL

1. [initialize]

   if T=NULL

   then Write('Empty Tree')

   Return

   else TOP<- 0

   Call PUSH(S,Top,T)
2. Repeat Step3 while TOP>0
3. P<-POP(S,Top)

   Repeat while P<>NULL

   Write(P->DATA)

   If RPTR(P)<>NULL

   then call PUSH(S,TOP,RPTR(P))

   P<-LPTR(P)
4. Return

NON RECURSIVE POST-ORDER TRAVERSAL

Initially push NULL onto STACK and then set PTR=ROOT. Then repeat the following steps until NULL is popped from STACK

a. Proceed down the left-most path rooted at PTR. At each node N of the path, push N onto the STACK and, if N has a right child R(N), push –R(N) onto the STACK.
b. [Backtracking] Pop and process positive nodes on STACK. If NULL is popped, then Exit. If a negative node is popped, that is, if PTR=-N for some node N, set PTR=N(PTR=-PTR) and return to Step(a)

POST-ORDER TRAVERSAL(I)

1. If T=NULL

   then Write('Empty')

   Return

   else PTR<-T

   TOP<-0

2. Repeat steps 3 thru 5 while PTR<>NULL
3. Call PUSH(STACK,TOP,PTR)
4. if RIGHT[PTR]<>NULL, then Call PUSH(STACK,TOP,-PTR)
5. PTR<-LPTR(p)
6. PTR<-POP(STACK,TOP)
7. Repeat while PTR>0
   a. Write DATA(p)
   b. PTR<-POP(STACK,TOP)
8. if PTR<0 then
   a. PTR=-PTR
   b. Go to Step2

OPERATIONS ON BINARY TREES

- Insertion
- Deletion

SEARCHING

Algorithm SEARCH (PTR0, KEY)

Steps:

```
Ptr=ptr0
If (Ptr->data != KEY)
        If (Ptr->LCHILD != NULL)
                SEARCH (Ptr->LCHILD,KEY)
        Else
                Return (0)
        End if
        If (Ptr->RCHILD !=NULL)
                SEARCH (Ptr->RCHILD,KEY)
        Else
                Return (0)
        End if
else
        return (Ptr)
end if
end
```

INSERTION

Algorithm INSERT_BIN_TREE (KEY,ITEM)

Input: KEY, the data content of the key node after which a new node is to be inserted and ITEM is the data content of the new node that has to be inserted.

Steps:

```
Ptr=SEARCH (Root, KEY)
if (ptr == NULL)
    print "Search is Unsuccessful: No insertion"
    exit
end if
If (ptr->LCHILD=NULL or ptr->RCHILD=NULL)
    read option to insert as left or right child
    if (option =L)
        if (ptr->LCHILD =NULL)
            create();
            nn->data=ITEM
            ptr->LCHILD=nn
```

```
        else
            print “insertion is not possible as left  child”
            exit
    else
        If (Ptr->RCHILD=NULL)
            create()
            nn->data=ITEM
            Ptr->RCHILD=nn
        else
            print “insertion is not possible as right child”
            exit
        end if
else
    print “key node already has 2 child nodes”
end if
end
```

DELETION

Algorithm DELETE_BIN_TREE (ROOT,ITEM)
***Steps*:**

```
Ptr=ROOT
If ptr =NULL
            Print "tree is empty"
            Exit
End if
Parent=SEARCH_PARENT (ROOT, ITEM)
If Parent != NULL
      Ptr1=parent->LCHILD
      Ptr2=parent->RCHILD
      If ptr1->DATA = ITEM
            If ptr1->LCHILD=NULL and ptr1->RCHILD=NULL
               Parent->LCHILD = NULL
            Else
                Print "Node is not a leaf node: NO deletion"
            End if
```

```
        Else
            If ptr2->LCHILD=NULL and ptr2->RCHILD=NULL
                Parent->RCHILD = NULL
            Else
                Print "Node is not a leaf node: NO deletion"
            End if
        End if
Else
    print "node with data ITEM does not exist: Deletion fails"
End if
End
```

SEARCHPARENT

```
search_parent(struct node *ptr0,int item)
 {
   parent=ptr0;
   ptr1=ptr0->lchild;
   ptr2=ptr0->rchild;
   if(ptr1!=NULL)
   {
    if(ptr1->data!=item)
    {
      search_parent(ptr1,item);
    }
    else
    {
      return(parent);
    }
   }
```

```
else if(ptr2!=NULL)
    {
     if(ptr2->data!=item)
     {
       search_parent(ptr2,item);
     }
     else
     {
       return (parent);
     }
    }
    else
    {
      return(NULL);
    }
  }
```