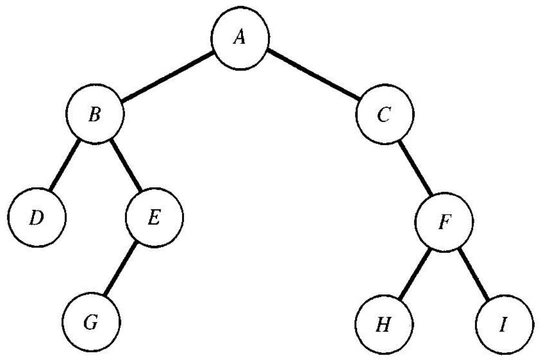
Binary Tree

A binary tree is a finite set of elements that is either empty or is partitioned into 3 disjoint subsets:

- a single element called the root
- two subsets which are themselves binary tree, called the left subtree and right subtree of the root

Each element of a binary tree is called a *node* of the tree.

An example binary tree:



Parent-child (father-son) relation

- The root of the subtree of node x are the children of x
- x is the parent of its children
- Two nodes are siblings (brothers) if they are left and right child of the same parent.
- In the above example, A is the parent of B and C.

Ancestor-descendant relation

- A simple path is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that the nodes are all distinct and there is an edge between each pair of nodes (n_1, n_2), (n_2, n_3), …, (n_{k-1}, n_k)
- The nodes along the simple path from the root to node x are the ancestors of x.
- The descendants of a node x are the nodes in the subtrees of x.

Degree of a node = number of subtrees of a node

In a binary tree, all nodes have degree 0, 1, or 2.

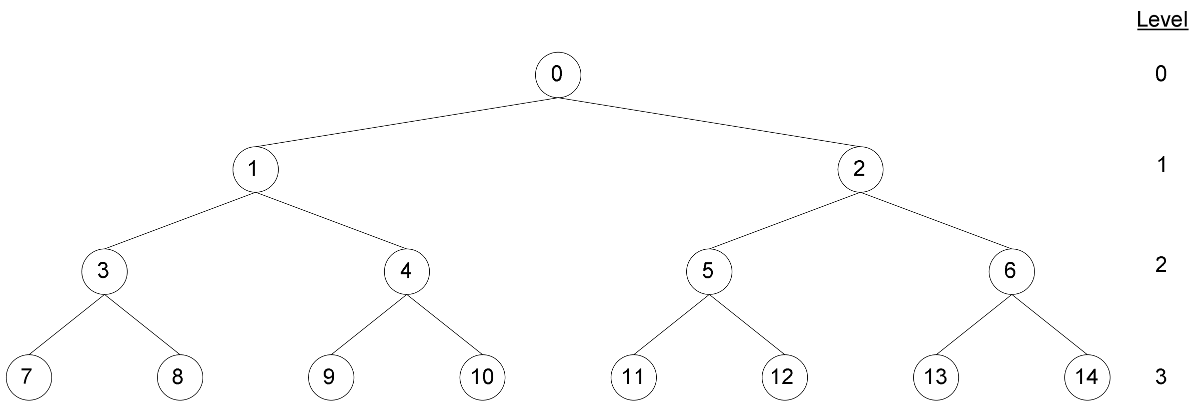
Nodes that have degree zero are called leaf nodes (terminal nodes)

Nodes that are not leaf nodes are called non-leaf nodes (branch nodes, internal nodes or non-terminal nodes)

If every internal node in a binary tree has nonempty left and right subtrees, the tree is termed strictly binary tree.

Level of a node

- root is at level 0 (some books define the root at level 1)
- if a node is at level i, then its children are at level i+1



Numbering of nodes of a *complete binary tree* (root is labelled 0)

1. The depth (or height) of a binary tree is the maximum level of any leaf in the tree (this is equal to the length of the longest path, i.e. number of branches, from the root to any leaf).

   According to this definition, depth of a tree with only 1 node is equal to zero.

2. In the textbook, the height of a binary tree is defined to be the number of nodes on the longest path from the root to a leaf.

   According to this definition, the height of a tree with only 1 node is equal to 1. (This definition is used in the study of height-balanced tree).

The maximum number of nodes in a binary tree of depth k (definition 1) is $2^{k+1} - 1$.

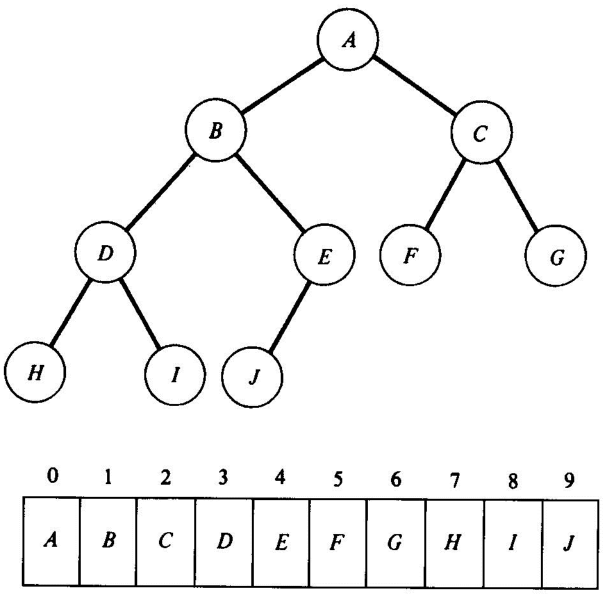
A binary tree with n nodes and depth d is *almost complete* iff its nodes corresponds to the nodes which are numbered 0 to n–1 in the complete binary tree.

Some books use the term *full binary tree* for complete binary tree, and *complete binary tree* to mean almost complete binary tree.

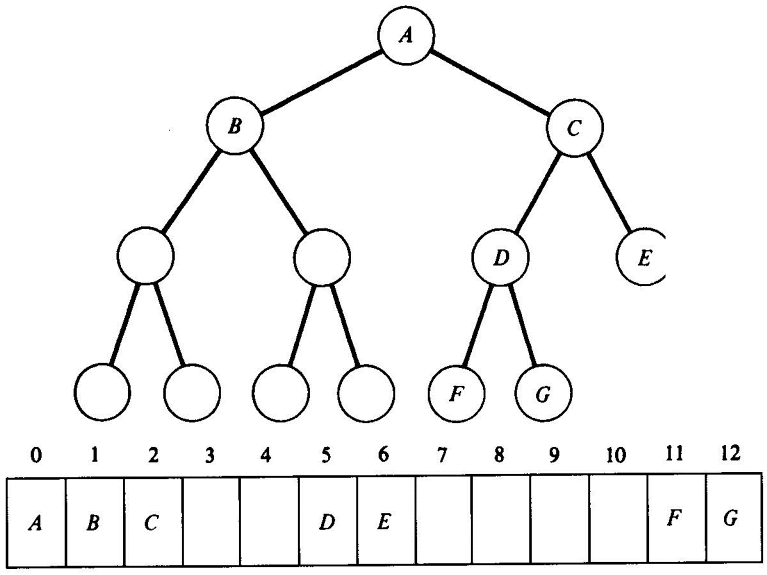
Array representation of an almost complete binary tree with n nodes

The nodes are numbered in level order.

The root is assigned the label 0 (nodes are labelled 0 to n-1)
(a) parent(i) is at $\lfloor (i-1)/2 \rfloor$ if $i \neq 0$. When $i = 0$, i is the root and has no parent.
(b) Lchild(i) is at $2i+1$ if $2i+1 < n$. If $2i+1 \geq n$, then i has no left child.
(c) Rchild(i) is at $2i+2$ if $2i+2 < n$. If $2i+2 \geq n$, then i has no right child.



In general, array representation of binary tree is not memory efficient.



Linked representation of binary tree

```
template<class Type>
struct treeNode
{  Type info;
   treeNode<Type> *left, *right;
};
```

<u>Traversals of binary trees</u>
To visit systemically the nodes and process each of them in some manner.

Four basic traversal orders

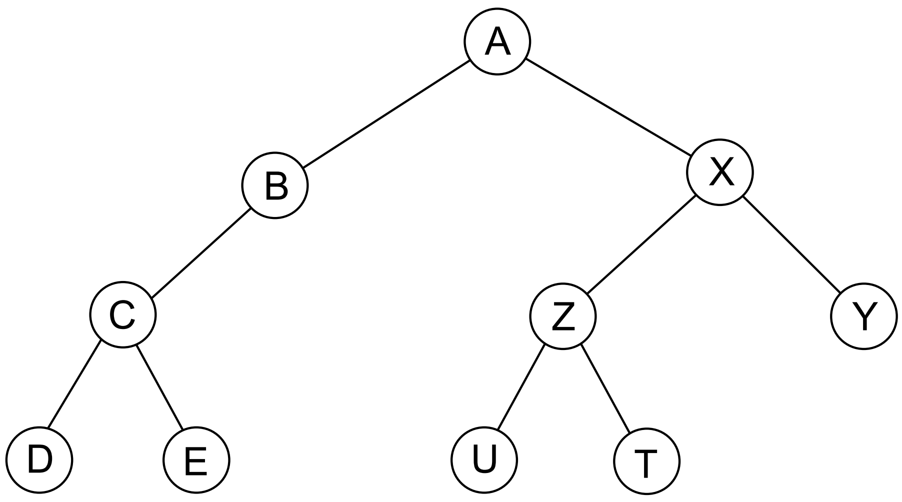
1. Preorder
   - visit the root
   - visit the left subtree in preorder
   - visit the right subtree in preorder

2. Inorder
   - visit the left subtree in inorder
   - visit the root
   - visit the right subtree in inorder

3. Postorder
   - visit the left subtree in postorder
   - visit the right subtree in postorder
   - visit the root

4. Level order
   - visit the nodes level by level starting from the root, and from left to right on each level

Example:



```
Preorder traversal:


Inorder traversal:


Postorder traversal:


Level order traversal:
```

Recursive algorithms to traverse a binary tree and print the node label

```
template<class Type>
void preorder(treeNode<Type> *p)
{
   if (p != NULL)
   {
      cout << p->info << " ";  //visit the node
      preorder(p->left);
      preorder(p->right);
   }
}
```

```
template<class Type>
void inorder(treeNode<Type> *p)
{
   if (p != NULL)
   {
      inorder(p->left);
      cout << p->info << " ";  //visit the node
      inorder(p->right);
   }
}
```

```
template<class Type>
void postorder(treeNode<Type> *p)
{
   if (p != NULL)
   {
      postorder(p->left);
      postorder(p->right);
      cout << p->info << " ";  //visit the node
   }
}
```

Algorithm to traverse a binary tree in level order

```
#include <queue>

template<class Type>
void levelTrav(treeNode<Type> *tree)
{
   queue<treeNode<Type>*> Q;

   if (tree != NULL)
      Q.push(tree);

   while (!Q.empty()) //there are nodes not yet visited
   {
      treeNode<Type>* p = Q.front();
      Q.pop();

      cout << p->info << " ";

      if (p->left != NULL)
         Q.push(p->left);

      if (p->right != NULL)
         Q.push(p->right);
   }
}
```

Non-recursive inorder traversal using a stack

```
#include <stack>

template<class Type>
void traverseLeft(treeNode<Type> *p,
                  stack<treeNode<Type>*>& S)
{
   while (p != NULL)
   {
      S.push(p);
      p = p->left;
   }
}

template<class Type>
void inorder_2(treeNode<Type> *tree)
{
   Stack<treeNode<Type>*> S;

   traverseLeft(tree, S);

   //there are nodes not yet visited
   while (!S.empty())
   {
      treeNode<Type>* p = S.top();
      S.pop();

      cout << p->info << " ";
      traverseLeft(p->right, S);
   }
}
```

Reconstruction of Binary Tree from its Preorder and Inorder sequences.

Find the binary tree whose preorder traversal is

A B C D E X Z U T Y

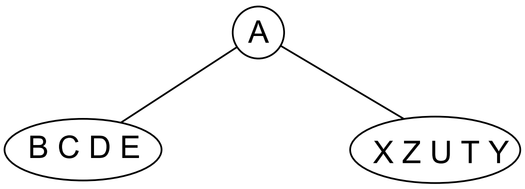
and whose inorder traversal is

D C E B A U Z T X Y

From Preorder, we know that A is the root.
From Inorder, we know that BCDE are in the left subtree and XZUTY are in the right subtree

Preorder **A** B C D E *X Z U T Y*

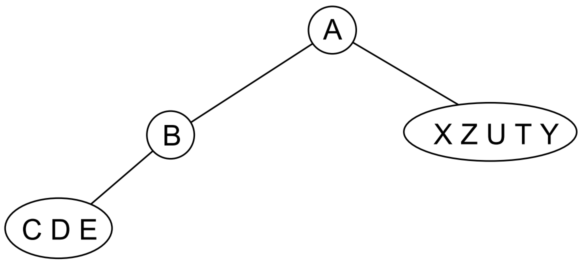
Inorder D C E B **A** *U Z T X Y*



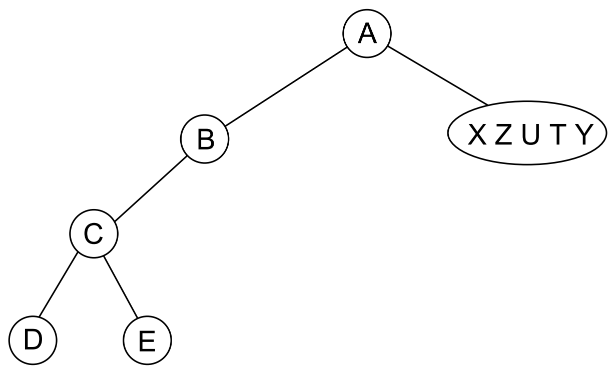
Consider the left subtree BCDE:
From preorder, B is the root;
From inorder, CDE are in the left subtree of B

Preorder **A B** C D E *X Z U T Y*

Inorder D C E **B A** *U Z T X Y*



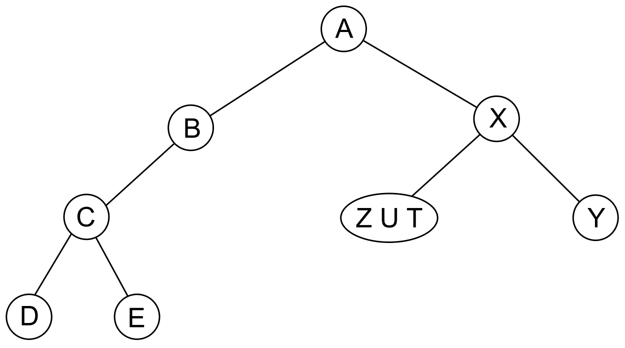
Consider the left subtree CDE:
C is the root; D is in the left subtree of C and E is in the right subtree of C



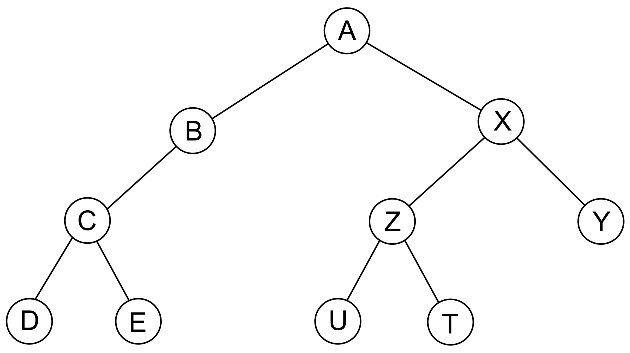
Consider the right subtree XZUTY:
X is the root; ZUT are in the left subtree of X; Y is in the right subtree of X;

Preorder *A B C D E* $\overline{\mathbf{X}\ \mathrm{Z\ U\ T\ Y}}$

Inorder *D C E B A* $\overline{\mathrm{U\ Z\ T}\ \mathbf{X}\ \mathrm{Y}}$



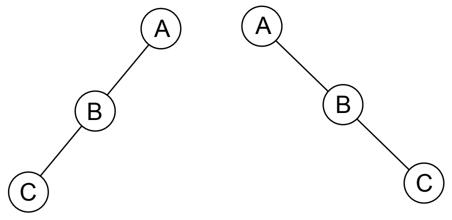
And the final result is



A binary tree may not be uniquely defined by its preorder and postorder sequences.

Example:

Preorder sequence:	ABC
Postorder sequence:	CBA



2 different binary trees can give you the same preorder and postorder sequences.

Algorithm to determine the height of a binary tree

```
template<class Type>
int height(treeNode<Type> *tree)
{
   if (tree == NULL)
      return 0;

   if ((tree->left == NULL) && (tree->right == NULL))
      return 0; // return 1 (if definition 2 is used)
                // or simply delete this if-statement

   int HL = height(tree->left);
   int HR = height(tree->right);

   if (HL > HR)
      return 1+HL;
   else
      return 1+HR;
}
```

Algorithm to count the number of nodes

```
template<class Type>
int nodeCount(treeNode<Type> *tree)
{
  if (tree == NULL)
    return 0;

  return 1 + nodeCount(tree->left) + nodeCount(tree->right));
}
```

Algorithm to count the number of leaf nodes

```
template<class Type>
int leavesCount(treeNode<Type> *tree)
{
  if (tree == NULL)
    return 0;

  if ((tree->left == NULL) && (tree->right == NULL))
    return 1;
  else
    return leavesCount(tree->left) + leavesCount(tree->right);
}
```

Algorithm to copy a binary tree

```
template<class Type>
void copyTree(treeNode<Type>*& copiedTree,
              treeNode<Type> *other)
{
   if (other == NULL)
      copiedTree = NULL;
   else
   {
      copiedTree = new treeNode<Type>;
      copiedTree->info = other->info;

      //copy the left subtree
      copyTree(copiedTree->left, other->left);

      //copy the right subtree
      copyTree(copiedTree->right, other->right);
   }
}
```

```
//alternative version

template<class Type>
treeNode<Type>* copyTree_2(treeNode<Type> *other)
{
   if (other == NULL)
      return NULL;

   treeNode<Type> *p = new treeNode<Type>;
   p->info = other->info;
   p->left = copyTree_2(other->left);
   p->right = copyTree_2(other->right);

   return p;
}
```

Algorithm to compare two binary trees

```
template<class Type>
bool equal(treeNode<Type> *tree1, treeNode<Type> *tree2)
{
  if ((tree1 == NULL) && (tree2 == NULL))
    return true;

  if ((tree1 != NULL) && (tree2 != NULL))
  {
     if ((tree1->info == tree2->info) &&
          equal(tree1->left, tree2->left)&&
          equal(tree1->right, tree2->right))
        return true;
  }
  return false;
}
```

Algorithm to print a binary tree in a table format

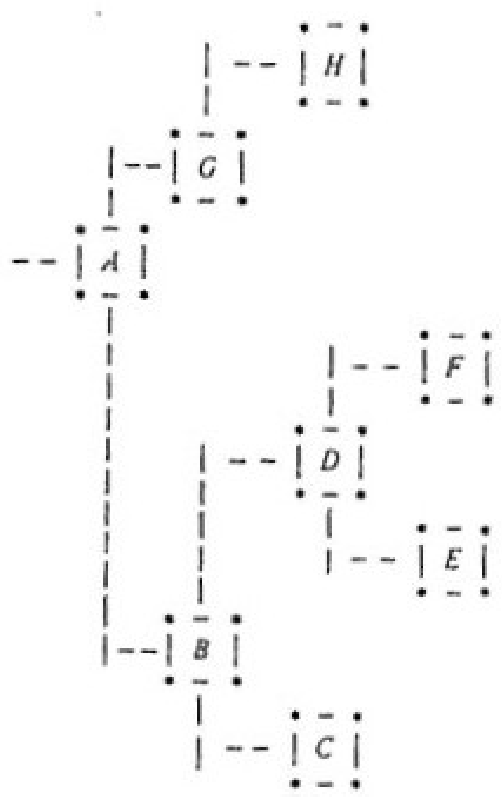
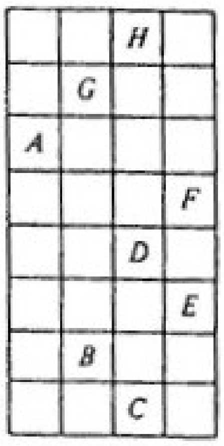


Figure 5.16
A "computer printed" version of Figure 5.5(a)



		H	
	G		
A			
			F
		D	
			E
	B		
		C	

Figure 5.17
A simplified version of Figure 5.16

```
#include <iomanip> //setw(), set width

template<class Type>
void printTree(treeNode<Type> *p, int indent)
{
    if (p != NULL)
    {
        //print right subtree, root, and then left subtree

        printTree(p->right, indent+3);
        cout << setw(indent) << p->info << endl;
        printTree(p->left, indent+3);
    }
}
```