《数据结构与算法》课程组

重庆大学计算机学院

Data Structures & Algorithms

8

TREE AND BINARY TREE

Outline

8.1 Definitions and Terminology

8.2 Binary Tree Definitions and Properties

8.3 Binary Tree ADT and Implementations

8.4 Binary Tree Traversal

8.5 General Tree ADT

8.6 General Tree Implementations

8.7 Converting Forest to Binary Tree

8.1 Definitions & Terminology

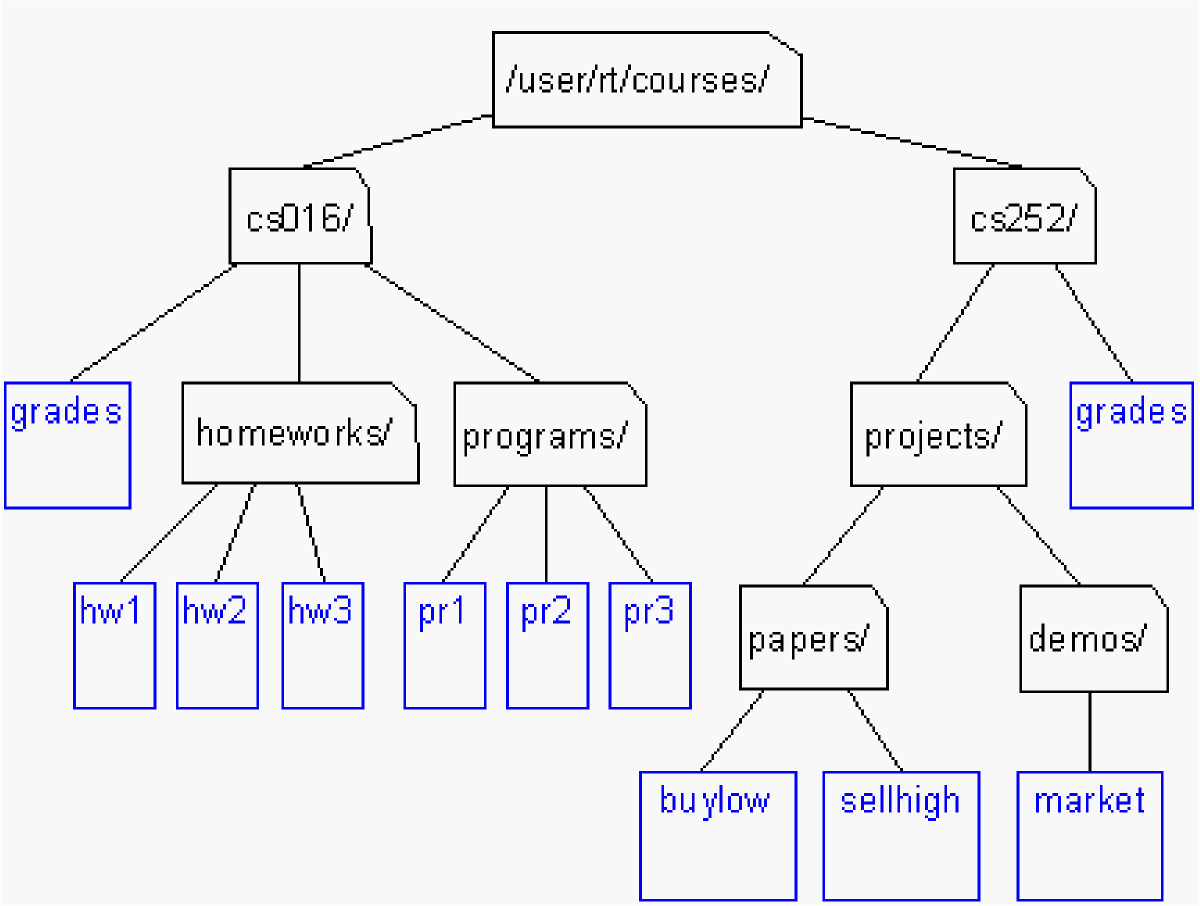
Tree Example

Representing File Structures:

- Consider the Unix file system
- Hierarchically arranged so that each file (including directories) belongs to some directory (except the / file which is the root)
- Each directory must be able to tell what files are in it

Tree Example

- **Unix / Windows file structure**

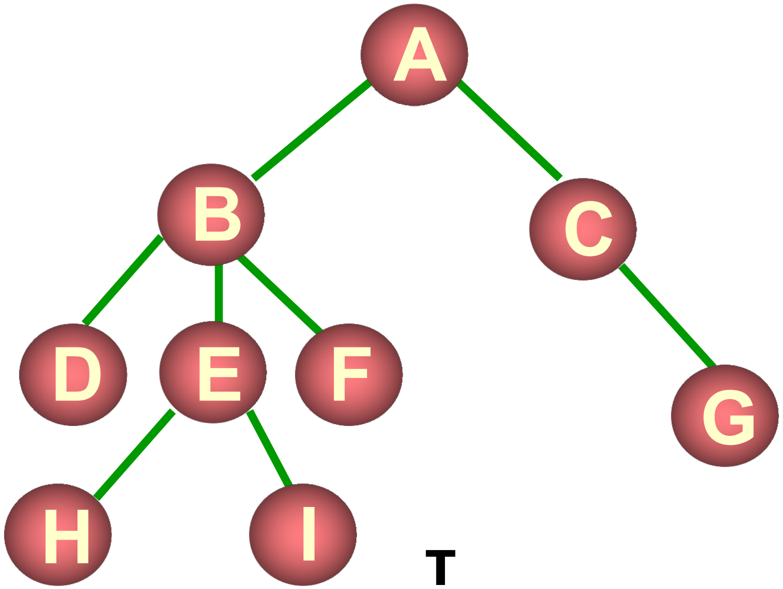


Other Trees

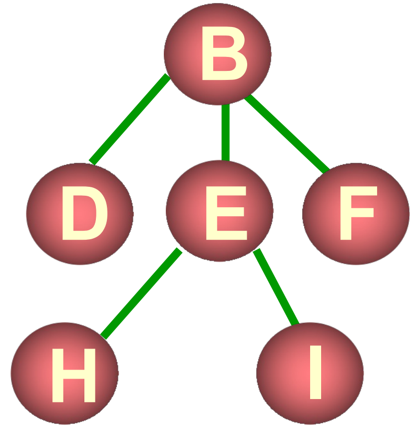
- **Family Trees**
- **Organization Structure Charts**
- **Program Design**
- **Structure of a chapter in a book**
- **……**

Definition of Tree

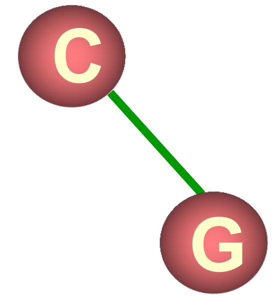
- A tree T is a finite set of one or more elements called **nodes** such that:
  - There is one designated node R, called the **root** of T.
  - If the set (T- {R}) is not empty, these nodes are partitioned into n > 0 **disjoint subsets** $T_0, T_1, \ldots, T_{n-1}$, each of which is a tree, and whose roots $R_1, R_2, \ldots, R_n$, respectively, are **children** of R
  - The subsets T_i ($0 < i < n$) are said to be subtrees of T.
- root: no predecessor
- leaf: no successor
- others: only one predecessor and one or more successor
- Definition is recursive



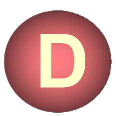
T



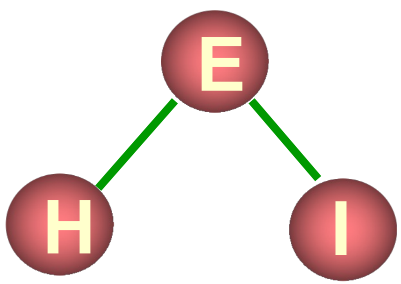
T_1



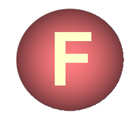
T_2



T_{11}



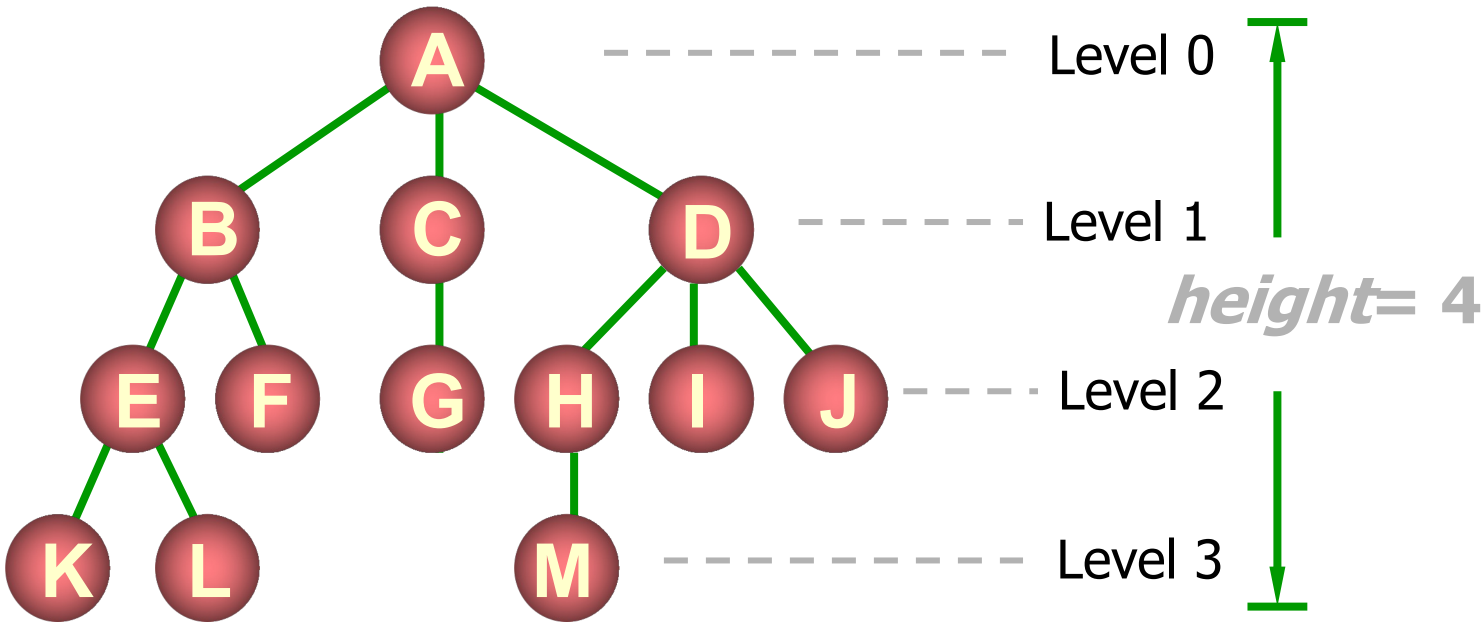
T_{12}



T_{13}

Tree: Level feature

- Root of subtree only have a direct previous, but can have 0 or more direct successor.



Terminology

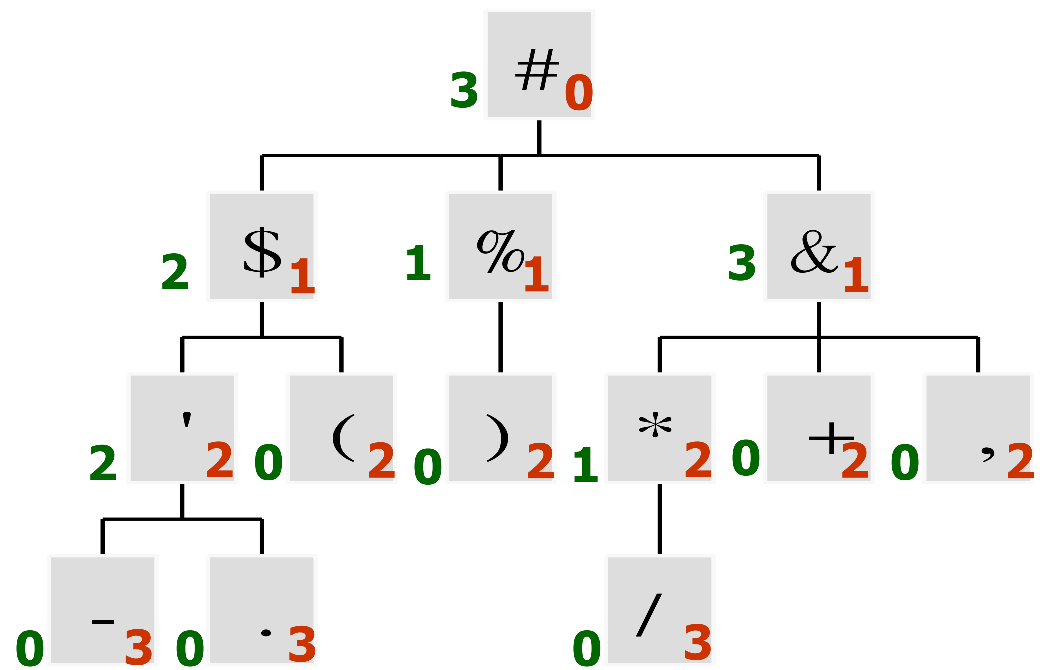
- There is an edge from a node to each of its children, and a node is said to be the parent of its children.
- If $n_1, n_2, \ldots, n_k$ is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a path from n_1 to n_k. The length of the path is $k - 1$.
- The depth of a node M in the tree is the length of the path from the root of the tree to M.
- All nodes of depth d are at level d in the tree.
- The height of a tree is one more than the depth of the deepest node in the tree.

Terminology

- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The degree of a tree is the maximum degree of the nodes in the tree.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the parent of the roots of the subtrees.
- The roots of these subtrees are the children of the node.
- Children of the same parent are siblings.
- The ancestors of a node are all the nodes along the path from the root to the node.
- A forest is a collection of one or more trees.

Terminology

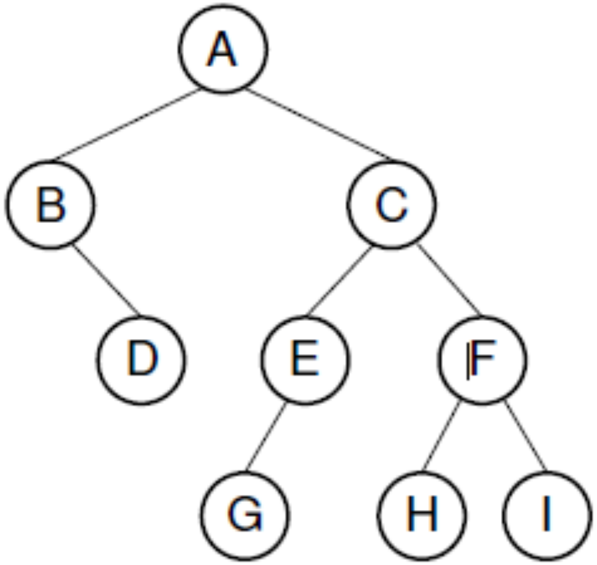
- node (13)
- degree of a node
- level of a node
- root
- leaf (terminal)
- internal node
- parent
- children
- sibling
- degree of a tree (3)
- ancestor
- descendant
- height of a tree (4)
- forest



8.2 Binary Tree Definitions and Properties

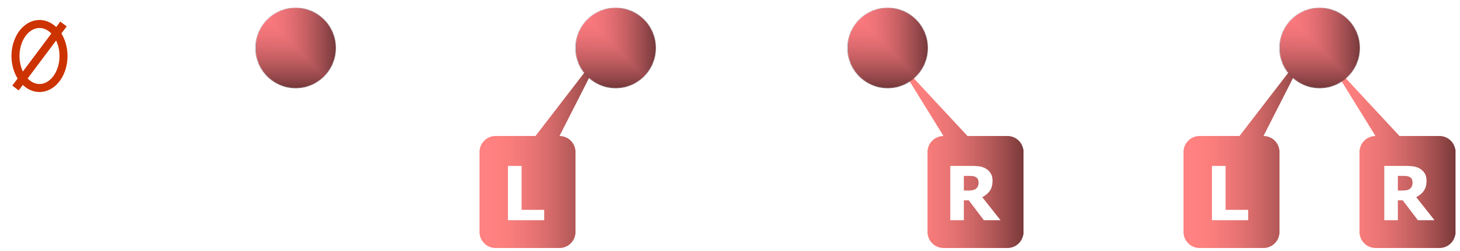
Definition of Binary Tree

- A **binary tree** is made up of a finite set of nodes. This set either is empty or consists of a node called the root together with two binary trees, called the **left subtree** and **right subtree**, which are disjoint from each other.
- A binary tree example

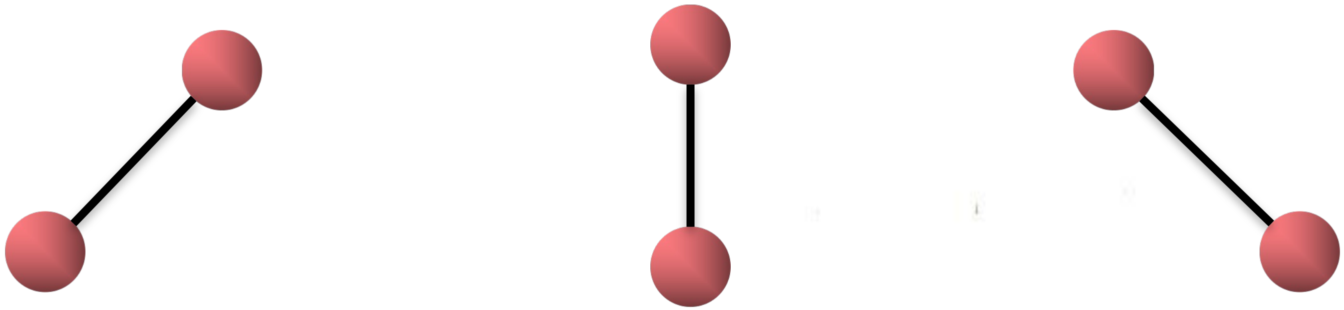


Shapes of Binary Tree

- **Binary Tree has five different shapes**

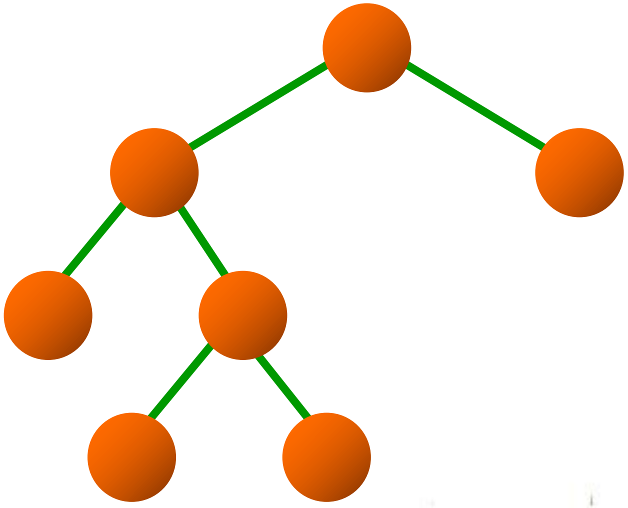


- left and right are important for binary trees
- Q: For general trees, the following three trees are the same or not?



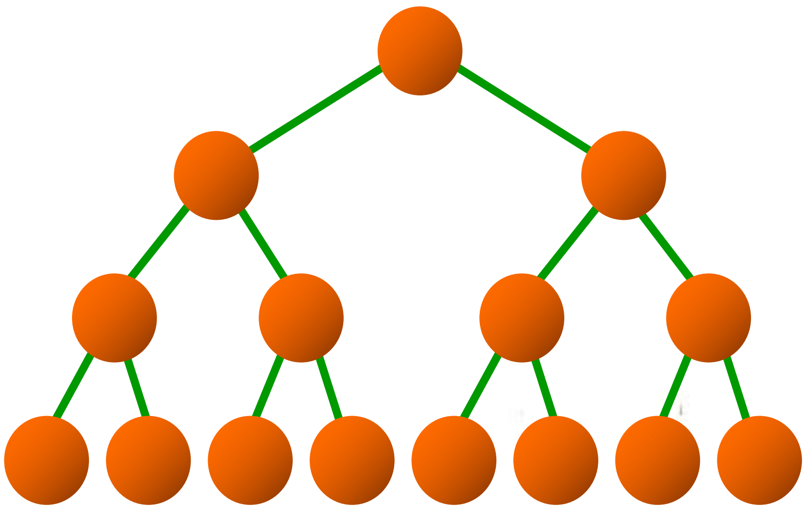
Full Binary Tree

- **Each node in a full binary tree is either a leaf or an internal node with exactly two non-empty child nodes**



Complete Binary Tree

- A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height d, all levels except possibly level d-1 are completely full. The bottom level has its nodes filled in from the left side.



Properties of Binary Tree

(1) The maximum number of nodes on i-th level of a binary tree is 2^{i-1} $(i \geq 1)$.

- **Proof:** The proof is by induction on i.
  - **Induction base**: The root is the only node on first level. The maximum number of nodes on 1th level is $2^{i-1} = 2^0 = 1$.
  - **Induction hypothesis**: For all j, $1 \leq j < i$, the maximum number of nodes on jth level is 2^{j-1}.
  - **Induction step**: Since each node has a maximum degree of 2, the maximum number of nodes on ith level is two times the maximum number of nodes on

    (i–1)th level or $2 * 2^{(i-1)-1} = 2^{i-1}$

Properties of Binary Tree

(2) The maximum number of nodes in a binary tree of height k is 2^k-1 (k≥1).

- **Proof:** **The maximum number of nodes in a binary tree of height k is:**

$$\sum_{i=1}^{k}(\text{max number of nodes on i-th level})$$

$$=\sum_{i=1}^{k}2^{i-1}=2^k-1$$

Properties of Binary Tree

(3) For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0=n_2+1$

- Let n be the total number of nodes in the tree, and let n_0, n_1, n_2 be numbers of nodes with 0 child (leaf), one child and two children, respectively.
- Thus, $n_0 + n_1 + n_2 = n$
- Each node except the root must be a child of another node, and two distinct internal node must have their children disjoint.
- As a result, we have $n_1 + 2*n_2 = n - 1$

Properties of Binary Tree

(3) For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0=n_2+1$

- proof:
  - Let n and B denote the total number of nodes & branches in T.
  - Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.
  - $n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$
  - $n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

Properties of Binary Tree

Full Binary Tree Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

- **Proof:**

  By definition, all internal nodes of full binary tree are nodes of degree 2.

  And $n_0=n_2+1$

8.3 Binary Tree Node ADT and Implementations

Binary Tree Node ADT

```
// Binary tree node abstract class
template <typename E> class BinNode {
public:
    virtual ~BinNode() {} // Base destructor
    // Return the node's value
    virtual E& element() = 0;
    // Set the node's value
    virtual void setElement(const E&) = 0;
    // Return the node's left child
    virtual BinNode* left() const = 0;
```

Binary Tree ADT

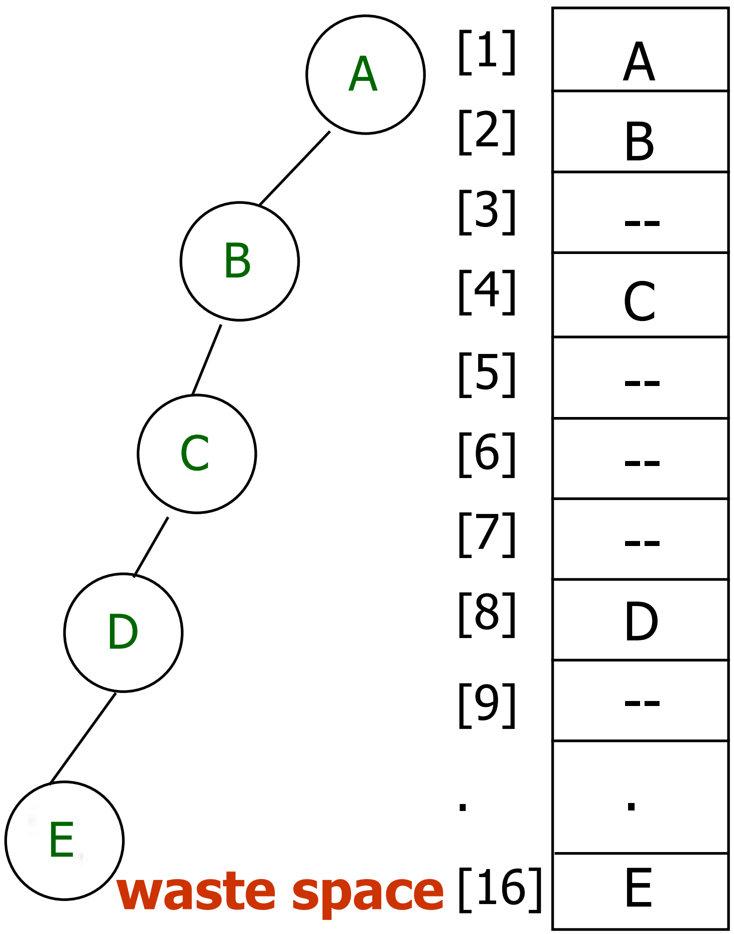
```
// Set the node's left child
virtual void setLeft(BinNode*) = 0;
// Return the node's right child
virtual BinNode* right() const = 0;
// Set the node's right child
virtual void setRight(BinNode*) = 0;
// Return true if the node is a leaf, false otherwise
virtual bool isLeaf() = 0;
};
```

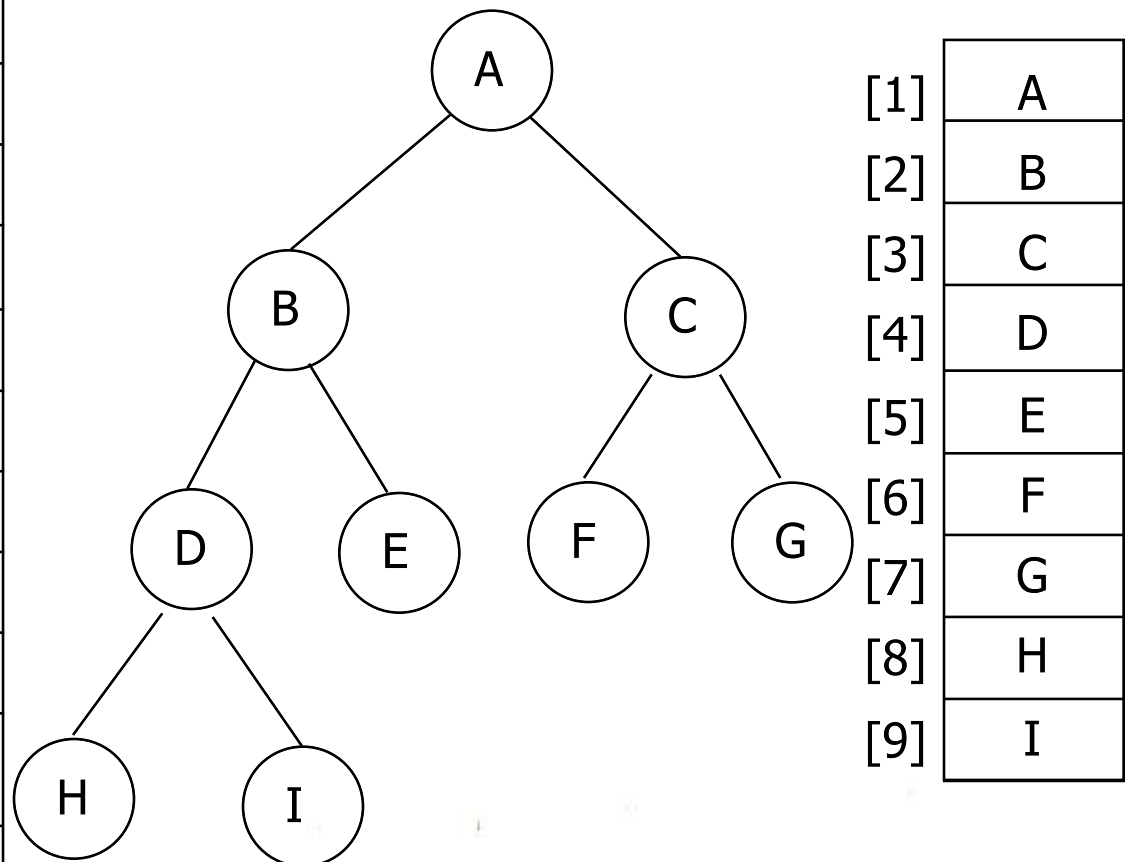
Binary Tree Implementations

- **There are two implementations:**
  - **array-based implementation**
  - **pointer-based implementation**

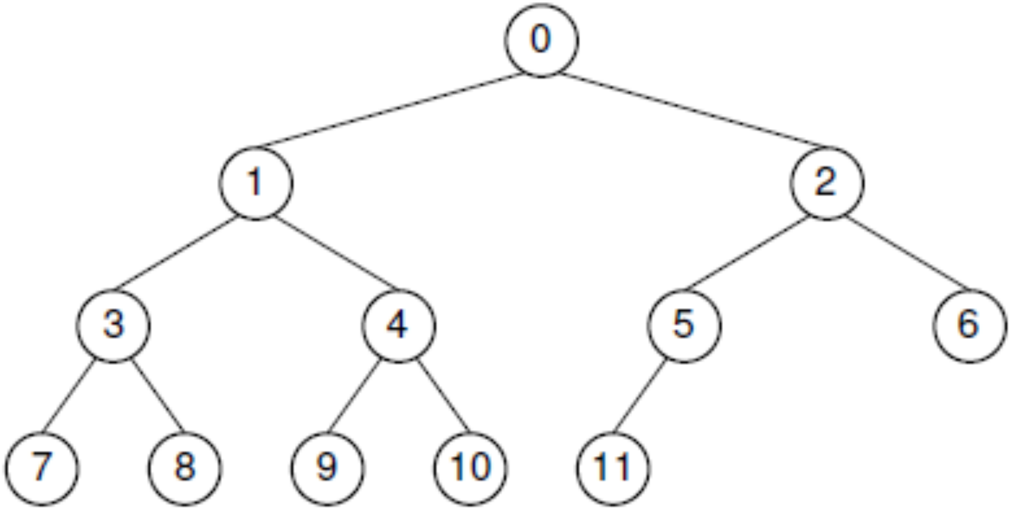
Array-Based Implementation

- **How to use an array representation for binary trees?**





Array-Based Implementation for Complete Binary Trees



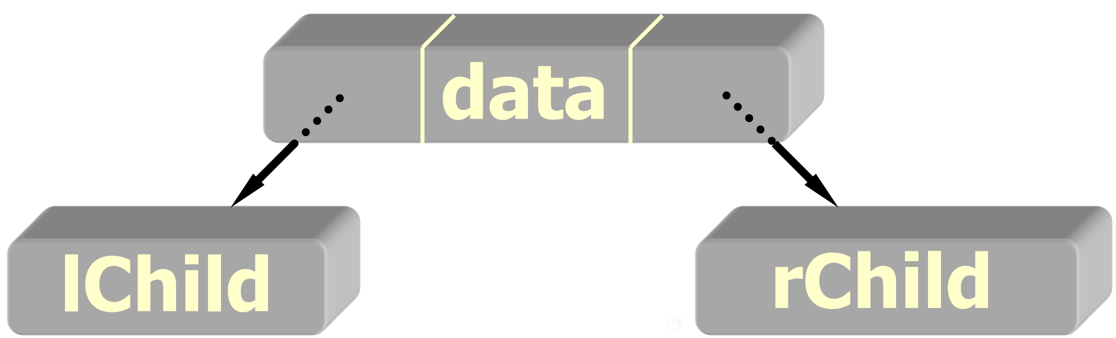
Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

Array-Based Implementation for Complete Binary Trees

- The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to n-1.
  - Parent(r) = $\lfloor (r - 1) / 2 \rfloor$ if $r \neq 0$.
  - Left child(r) = $2r + 1$ if $2r + 1 < n$.
  - Right child(r) = $2r + 2$ if $2r + 2 < n$.
  - Left sibling(r) = $r - 1$ if r is even.
  - Right sibling(r) = $r + 1$ if r is odd and $r + 1 < n$.

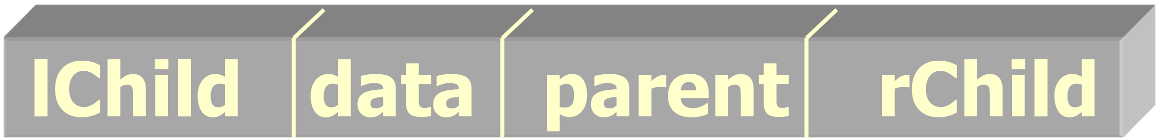
Pointer-Based Implementation

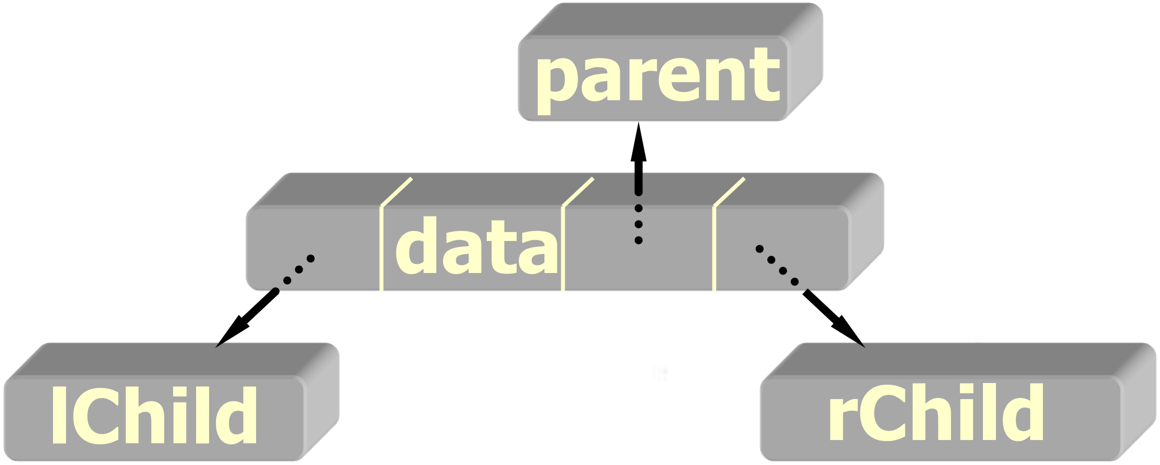
- All binary tree nodes have two children.
- The most common node implementation includes a value field and pointers to the two children



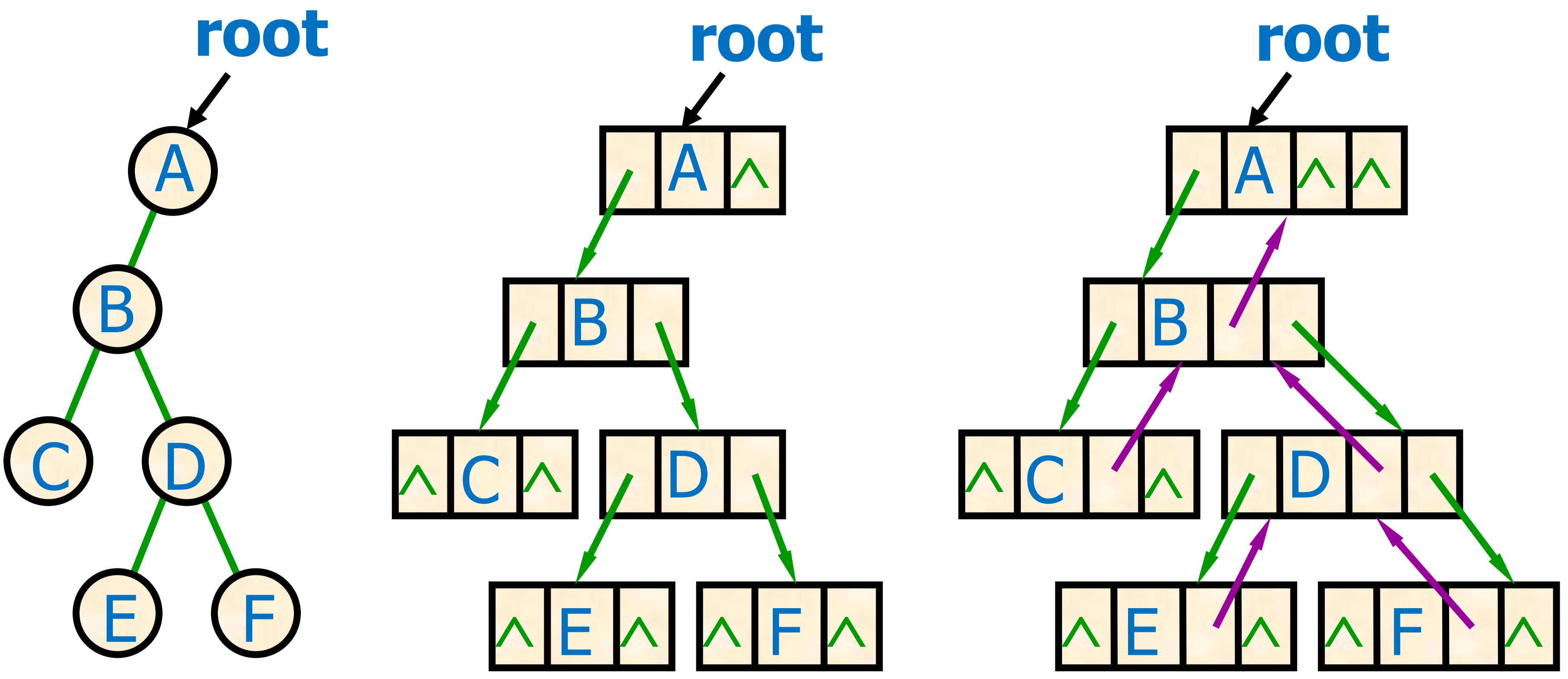
Pointer-Based Implementation

- **All binary tree nodes have two children and one parent, except root.**
- **An alternate node implementation includes a value field and pointers to the two children and one parent.**





Example



Pointer-Based Node Implementation

```
                    // Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
   Key k;               // The node's key
   E it;                // The node's value
   BSTNode* lc;    // Pointer to left child
   BSTNode* rc;    // Pointer to right child

public:
   // Two constructors -- with and without initial values
   BSTNode() { lc = rc = NULL; }
   BSTNode(Key K, E e, BSTNode* l =NULL, BSTNode* r =NULL)
      { k = K; it = e; lc = l; rc = r; }
   ~BSTNode() {} // Destructor
```

Pointer-Based Node Implementation

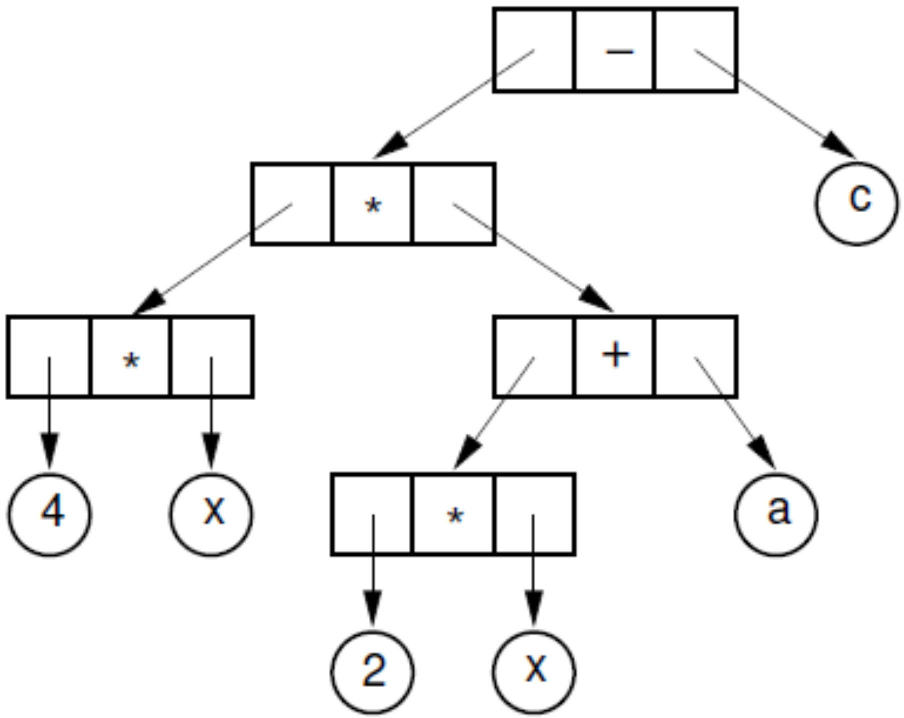
```
    // Functions to set and return the value and key
  E& element() { return it; }
  void setElement(const E& e) { it = e; }
  Key& key() { return k; }
  void setKey(const Key& K) { k = K; }

    // Functions to set and return the children
  inline BSTNode* left() const { return lc; }
  void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
  inline BSTNode* right() const { return rc; }
  void setRight(BinNode<E>* b) { rc = (BSTNode*)b; }

    // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};
```

Distinguish leaf and internal nodes

- **Why?**
  - Some applications require data values only for the leaves.
  - Other applications require one type of value for the leaves and another for the internal nodes.
- By definition, only internal nodes have non-empty children. It can save space to have separate implementations for internal and leaf nodes.

Example: Expression Tree

- **Expression : 4*(2*x + a) - c**



Internal nodes store operators, while the leaves store operands.

Approach

- Approach 1: to use class inheritance
  - A base class can be declared for binary tree nodes in general, with subclasses defined for the internal and leaf nodes.
- Approach 2: to use the composite design pattern
  - using a virtual base class and separate node classes for the two types.

Approach1 --leaf node representation

```
// Node implementation with simple inheritance
class VarBinNode { // Node abstract base class
public:
    virtual ~VarBinNode() {}
    virtual bool isLeaf() = 0; // Subclasses must implement
};

class LeafNode : public VarBinNode { // Leaf node
private:
    Operand var; // Operand value

public:
    LeafNode(const Operand& val) { var = val; } //
     Constructor
    bool isLeaf() { return true; } // Version for LeafNode
    Operand value() { return var; } // Return node value
};
```

Approach1 --internal node representation

```
class IntlNode : public VarBinNode { // Internal node
private:
   VarBinNode* left; // Left child
   VarBinNode* right; // Right child
   Operator opx; // Operator value

public:
   IntlNode(const Operator& op, VarBinNode* l,
    VarBinNode* r)
      { opx = op; left = l; right = r; } // Constructor
   bool isLeaf() { return false; } // Version for IntlNode
   VarBinNode* leftchild() { return left; } // Left child
   VarBinNode* rightchild() { return right; } // Right child
   Operator value() { return opx; } // Value
};
```

Approach1--Function traverse

```
void traverse(VarBinNode *root) { // Preorder traversal
    if (root == NULL) return; // Nothing to visit
    if (root->isLeaf()) // Do leaf node
        cout << "Leaf: " << ((LeafNode *)root)->value() << endl;
    else { // Do internal node
        cout << "Internal: "
             << ((IntlNode *)root)->value() << endl;
        traverse(((IntlNode *)root)->leftchild());
        traverse(((IntlNode *)root)->rightchild());
    }
}
```

Approach2 --leaf node implementation

```
// Node implementation with the composite design pattern
class VarBinNode { // Node abstract base class
public:
    virtual ˜VarBinNode() {} // Generic destructor
    virtual bool isLeaf() = 0;
    virtual void traverse() = 0; //virtual base class function
};
class LeafNode : public VarBinNode { // Leaf node
private:
    Operand var; // Operand value
public:
    LeafNode(const Operand& val) { var = val; } // Constructor
    bool isLeaf() { return true; } // isLeaf for Leafnode
    Operand value() { return var; } // Return node value
    void traverse() { cout << "Leaf: " << value() << endl; }
};
```

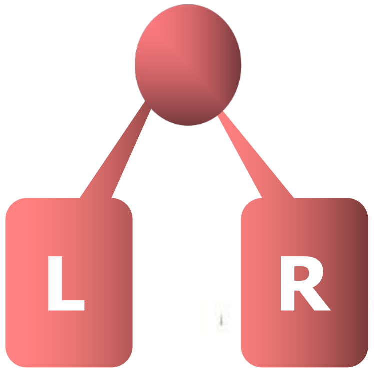
Approach2 --internal node implementation

```
class IntlNode : public VarBinNode { // Internal node
private:
   VarBinNode* lc; // Left child
   VarBinNode* rc; // Right child
   Operator opx; // Operator value
public:
   IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
      { opx = op; lc = l; rc = r; } // Constructor
   bool isLeaf() { return false; } // isLeaf for IntlNode
   VarBinNode* left() { return lc; } // Left child
   VarBinNode* right() { return rc; } // Right child
   Operator value() { return opx; } // Value
   void traverse() { // Traversal behavior for internal nodes
      cout << "Internal: " << value() << endl;
      if (left() != NULL)   lc->traverse();
      if (right() != NULL) rc->traverse();
   }
};
```

8.4 Binary Tree Traversals

Binary Tree Traversals

- Traversal: Each node is visited once and can only be visited once.
- Traversal is easy to Linear structure. But to nonlinear structure, it is needed to linearize nonlinear structure according to certain rules
- The binary tree consists of three basic units: root, left subtree and right subtree.



Binary Tree Traversals

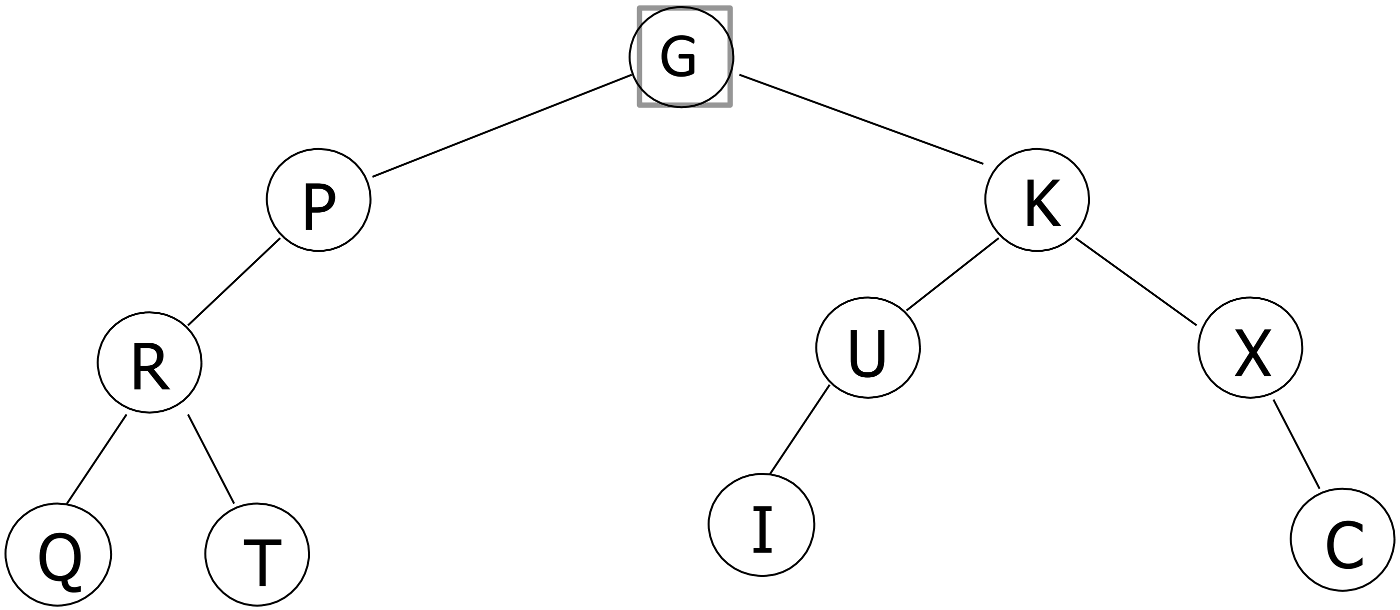
- Let L, D, and R stand for moving left, visiting the root node, and moving right.
- There are six possible combinations of traversal
  - DLR, LDR, LRD, DRL, RDL, RLD
- Adopt convention that we traverse left before right, only 3 traversals remain
  - DLR, LDR, LRD
  - preorder, inorder , postorder

Binary Tree Traversals

- Suppose that we need to visit all of the nodes in a binary tree. In what order can this be done? The most common:
  - Preorder
  - Inorder
  - Postorder
  - Level Order
- Level order is breadth-first traversal, the other three are depth-first traversal.

Preorder Traversal

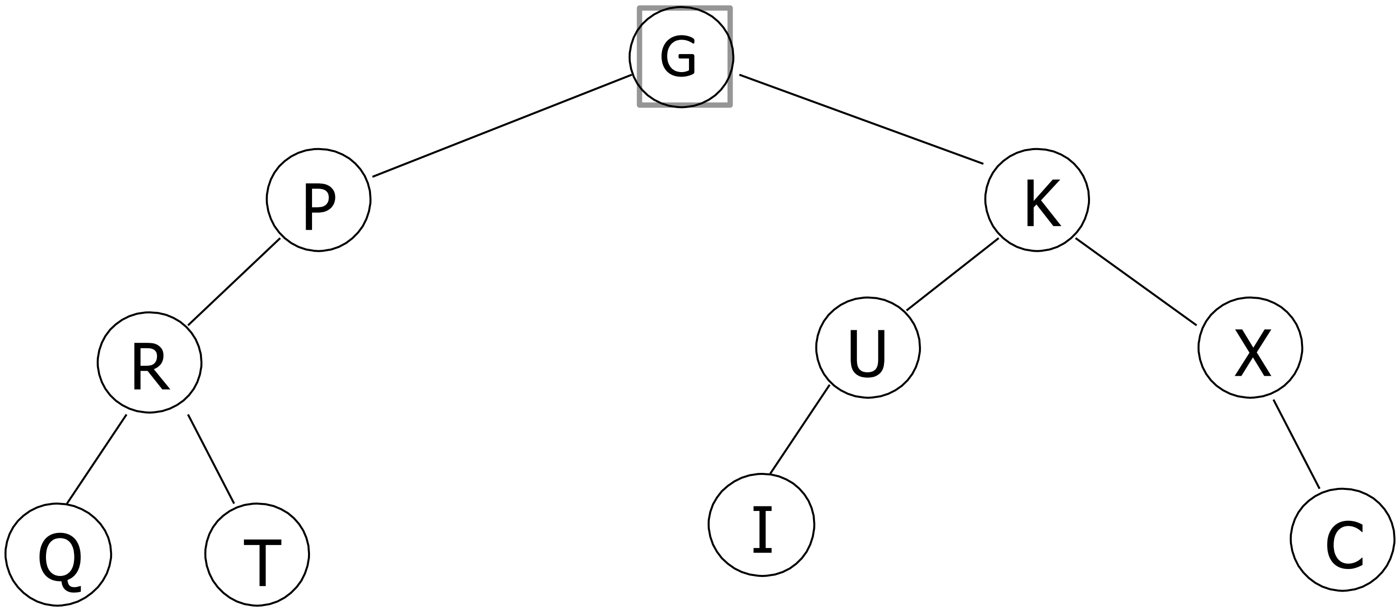
- **Visit the root node.**
- **Traverse the left subtree.**
- **Traverse the right subtree.**



G, P, R, Q, T, K, U, I, X, C

Inorder Traversal

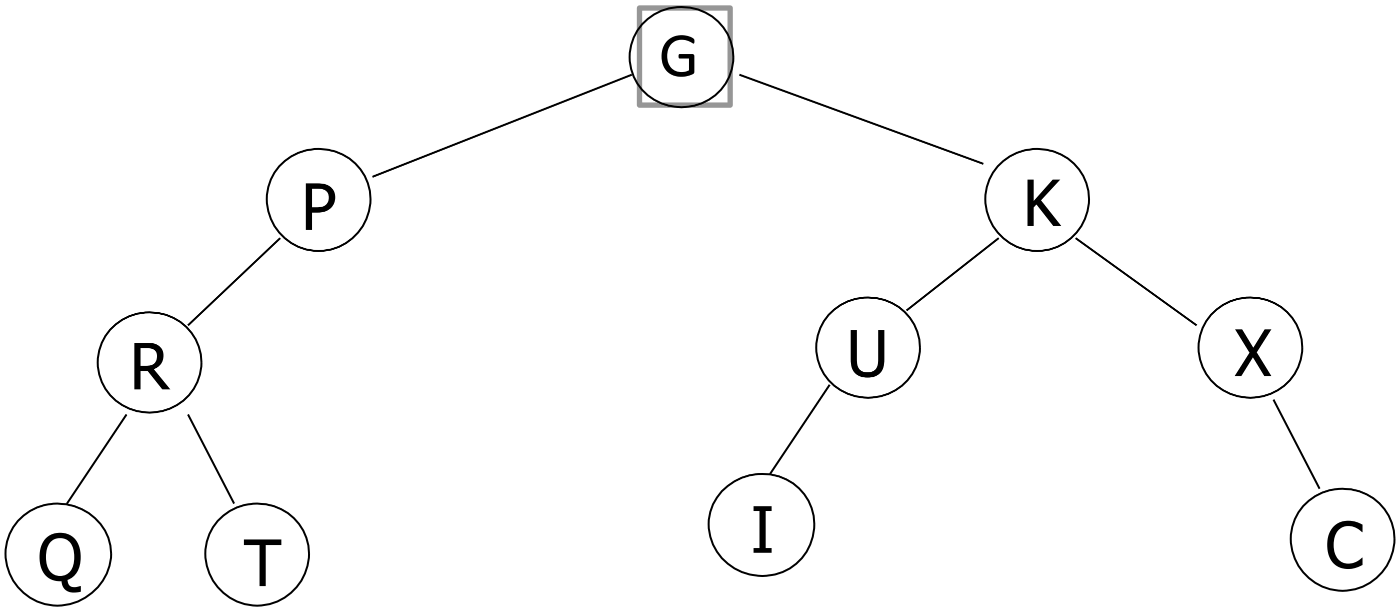
- Traverse the left subtree.
- Visit the root node.
- Traverse the right subtree.



Q, R, T, P, G, I, U, K, X, C

Postorder Traversal

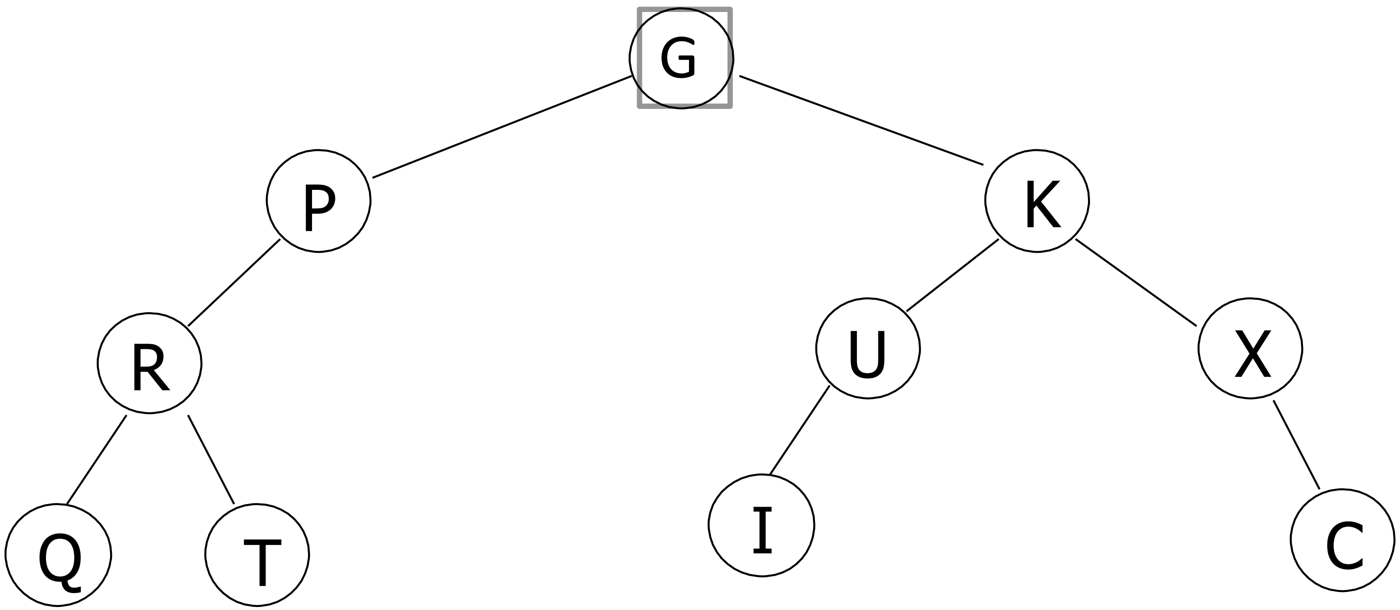
- **Traverse the left subtree.**
- **Traverse the right subtree.**
- **Visit the root node.**



Q, T, R, P, I, U, C, X, K, G

Level Order Traversal

- **Visit the nodes from level to level, beginning with the root node.**

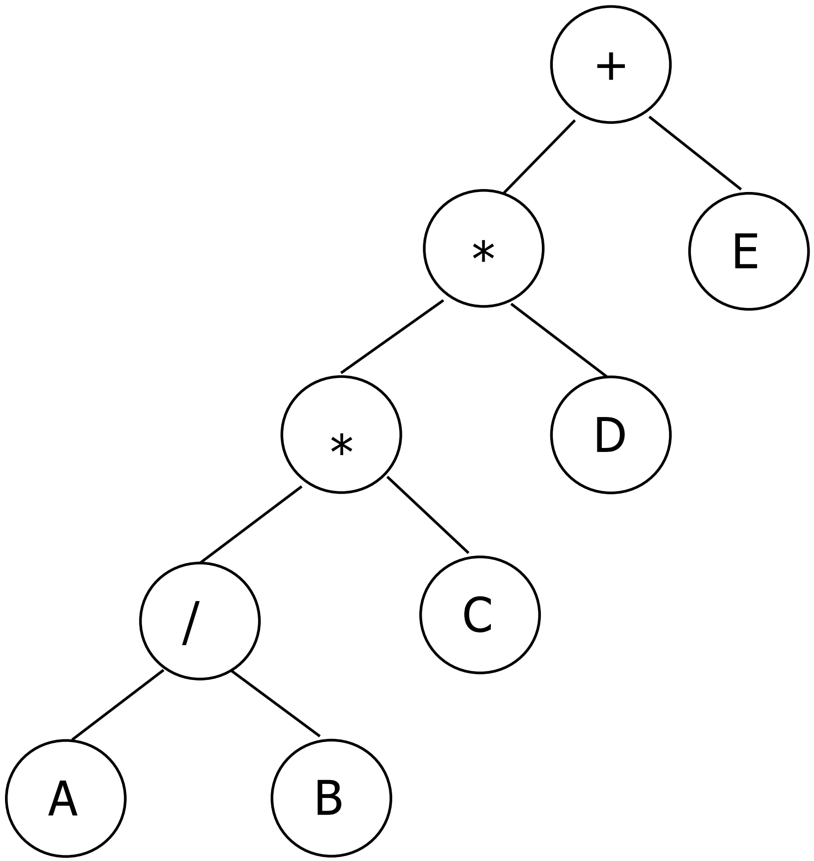


G, P, K, R, U, X, Q, T, I, C

Example: Expression Tree

- A Binary Tree built with operands and operators.
- Also known as a parse tree.
- Used in compilers.
- Notation
  - Preorder
    - Prefix Notation
  - Inorder
    - Infix Notation
  - Postorder
    - Postfix Notation

Example: Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

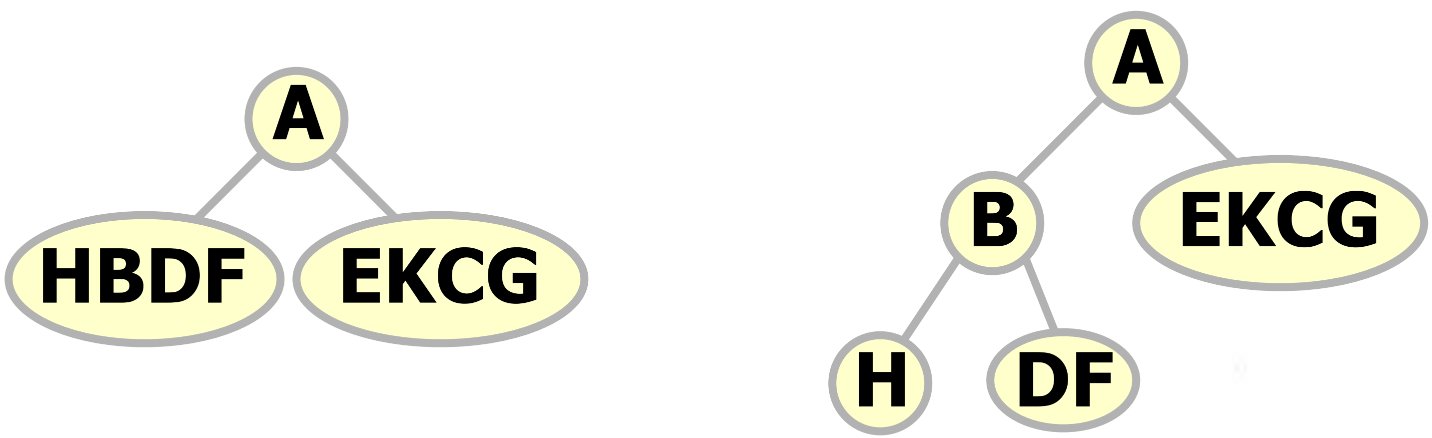
postfix expression

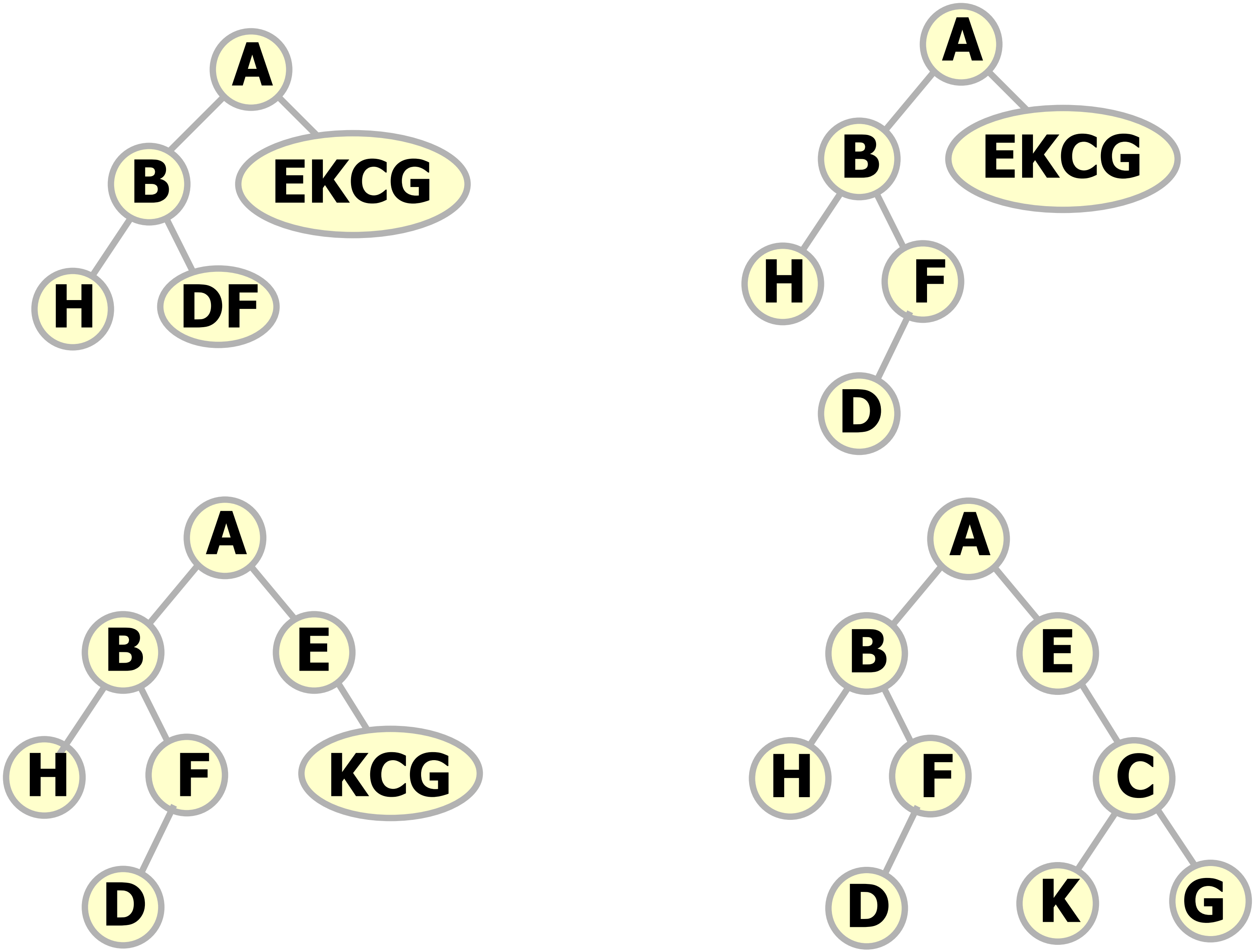
level order traversal

+ * E * D / C A B

Binary Tree Traversal Property

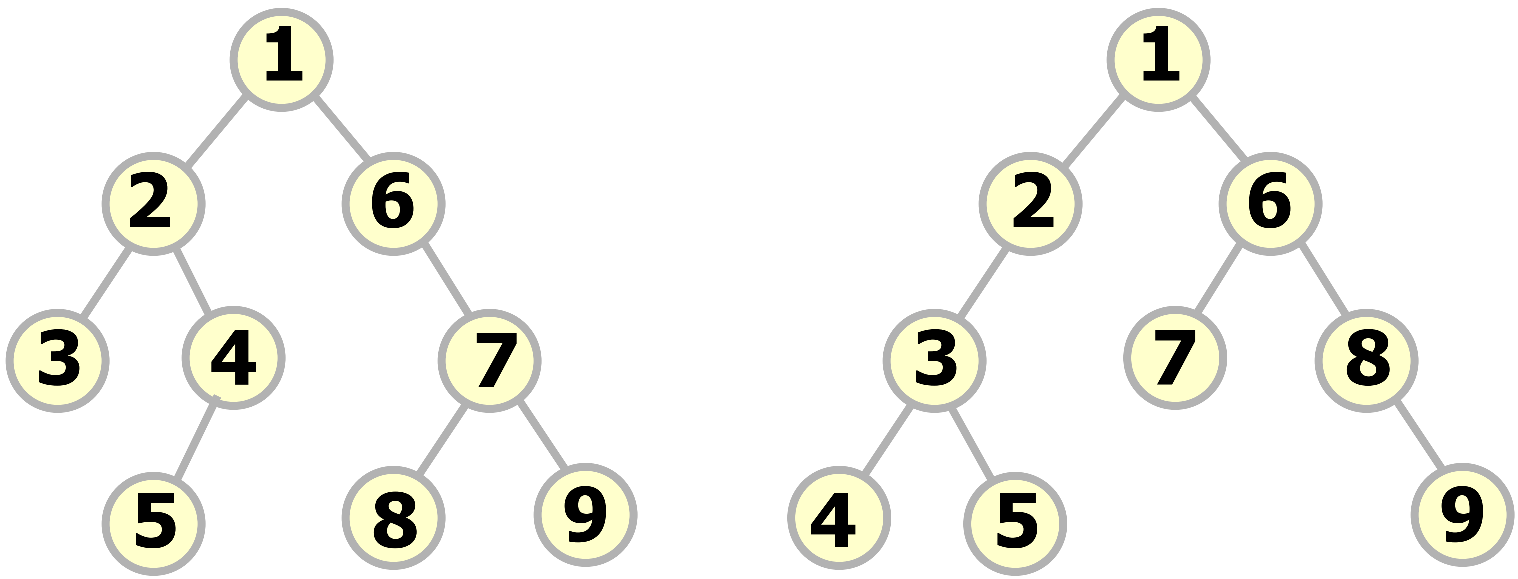
- A binary tree can be Uniquely constructed by preorder enumeration and inorder enumeration.
- For example, the preorder enumeration is { ABHFDECKG } and the inorder enumeration is { HBDFAEKCG }, the constructing process of binary tree is as follows:




A
B
EKCG
H
DF
A
B
EKCG
H
F
D
A
B
E
H
F
KCG
D
A
B
E
H
F
C
D
K
G

Binary Tree Traversal Property

- **If there is only a preorder enumeration {1, 2, 3, 4, 5, 6, 7, 8, 9} , we can get different binary tree.**



- A binary tree also can be uniquely constructed by postorder enumeration and inorder enumeration.

Preorder Traversal *(recursive)*

- A traversal routine is naturally written as a recursive function.

```
template <typename E>
void preorder(BinNode<E>* root) {
   if (root == NULL) return;   // Empty subtree, do
                               // nothing
   visit(root);                // Perform desired action
   preorder(root->left());
   preorder(root->right());
}
```

Preorder Traversal--preorder2

- **An important decision in the implementation of any recursive function on trees is when to check for an empty subtree.**
- **An alternate design as follows:**

```
template <typename E>
void preorder2(BinNode<E>* root) {
    visit(root); // Perform whatever action is desired
    if (root->left() != NULL) preorder2(root->left());
    if (root->right() != NULL) preorder2(root->right());
}
```

Preorder & preorder2

- The design of preorder2 is inferior to that of preorder for following two reasons:

(1) It can become awkward to place the check for the NULL pointer in the calling code.

(2) The more important concern with preorder2 is that it tends to be error prone.

  – the original tree is empty
  – Solution:

    ① an additional test for a NULL pointer at the beginning

    ② the caller of preorder2 has a hidden obligation to pass in a non-empty tree

Preorder Traversal

- Another issue to consider when designing a traversal is how to define the visitor function that is to be executed on every node.
  - Approach1: to write a new version of the traversal for each such visitor function.
  - Approach2: for the tree class to supply a generic traversal function which takes the visitor either as a template parameter or as a function parameter.

Application: Count the Number of Nodes

```
template <typename E>
int count(BinNode<E>* root) {
   if (root == NULL) return 0;     // Nothing to count
   return 1 + count(root->left())
              + count(root->right());
}
```

Non recursive algorithm

- Basic idea of preorder traversal using a stack:
  - When meet a node, visit the node and push it into the stack, and then traversal its left subtree.
  - After pop this node, traversal its right subtree.

Preorder Traversal *(Non recursive)*

```
void InOrderUnrec(BinNode *root){
    Stack<BinNode*> s;
    BinNode *p = root;              //指向当前访问的节点
    while (p!=NULL || !s.isEmpty()) {
       while (p != NULL) {       //遍历左子树
          visit(p);    //访问当前结点（前序遍历）
          s.push(p);          //节点入栈
          p = p->left();     //访问左子树
       }
       if (!s.isEmpty()) {
         p = s.pop();
          p = p->right();    //通过下一次循环实现右子树遍历
       }//endif
    }//endwhile
}
```

Inorder Traversal *(recursive)*

```
template <typename E>
void inorder(BinNode<E>* root) {
        if (root == NULL) return;
                // Empty subtree, do nothing
        inorder(root->left());
        visit(root);                // Perform desired action
        inorder(root->right());
}
```

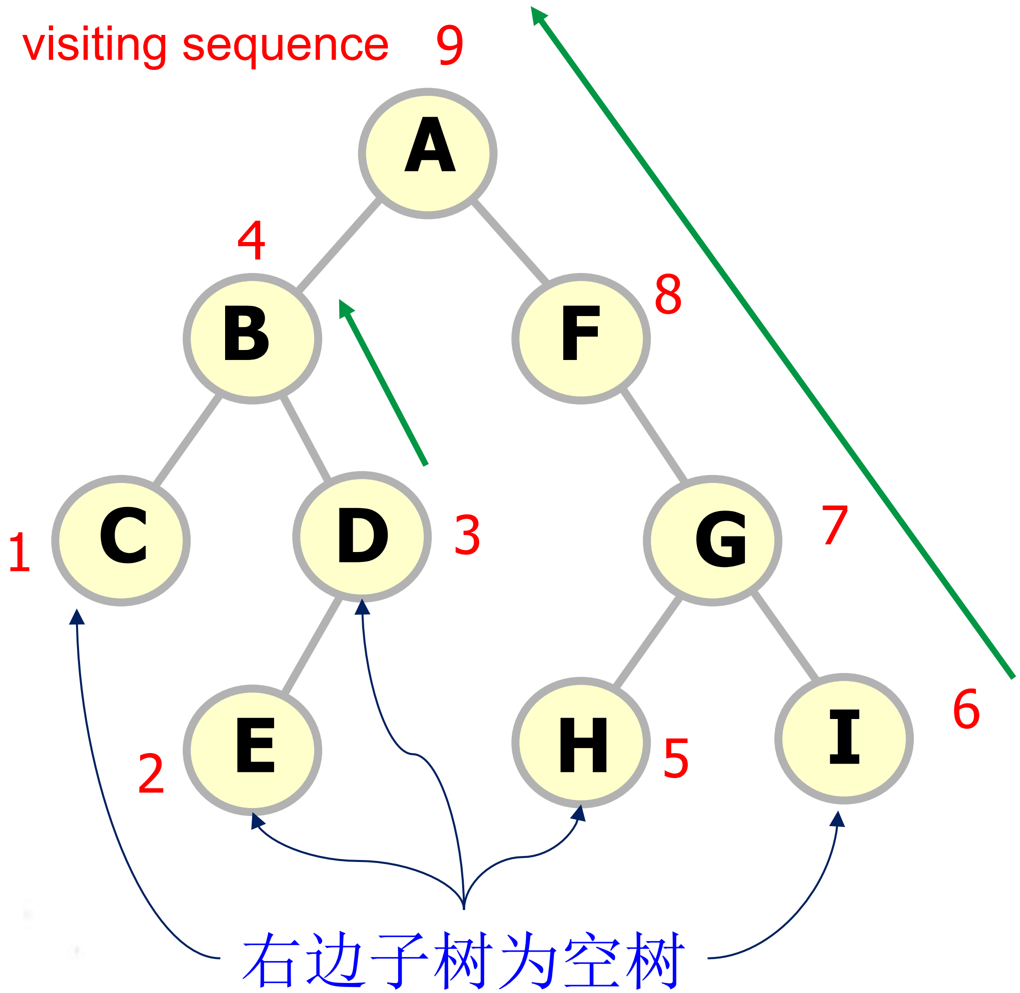
Inorder Traversal *(Non recursive)*

```
void InOrderUnrec(BinNode *root){
    Stack<BinNode*> s;
    BinNode *p = root;
    while (p!=NULL || !s.isEmpty()) {
      while (p != NULL) {        //遍历左子树
         s.push(p);
         p = p->left();
      }
      if (!s.isEmpty()) {
        p = s.pop();
        visit(p);             //访问根结点
        p = p->right();  //通过下一次循环实现右子树遍历
      }//endif
    }//endwhile
}
```

Postorder Traversal *(recursive)*

```
template <typename E>
void inorder(BinNode<E>* root) {
    if (root == NULL) return;
        // Empty subtree, do nothing
    postorder(root->left());
    postorder(root->right());
    visit(root);                // Perform desired action
}
```

Postorder Traversal *(Non recursive-I)*



Observation

1. 如果栈顶节点的右边子树为空(NULL)，直接出栈并访问(visit)；否则将其右子树的根节点压入栈

2. 如果被访问节点出栈后，新的栈顶节点是其父节点并且前者是后者的右子树根，则继续弹出栈顶节点，直到栈为空或者弹出的节点不是栈顶节点的右子树根为止。

Postorder Traversal *(Non recursive-I)*

```
void PostOrderUnrec(BinNode *root){
    Stack<BinNode*> s;
    BinNode *p = root;
    while (p!=NULL || !s.isEmpty()) {
        while (p != NULL) {       //遍历左子树
            s.push(p);
            p = p->left();
        }
        if (!s.isEmpty()) {
            p = s.top()->right();   //指向栈顶节点的右子树
            if(p == NULL) {       //栈顶节点无右子树，依次弹出栈顶节点
                BinNode * tmp;
                do{
                    tmp = s.pop();
                    visit(tmp);
                }while(!s.isEmpty() && tmp==s.top()->right());
            } //endif
        }//endif
    }//endwhile
}
```

Postorder Traversal *(Non recursive-II)*

根据前序遍历和后序遍历的对称性！

$$LRD = (LRD)^{T\circ T} = (DR^{T}L^{T})^{T}$$

前序遍历与反转

二叉树常见面试题

- 求二叉树的高度（深度） //递归和非递归
- 求二叉树中（叶）节点的个数
- 求二叉树第K层的节点个数（假设根节点为第1层）
- 判断一个节点是否在二叉树中
- 求两个节点的最近公共祖先 //时间复杂度O(n)
- 判断一棵二叉树是否是完全二叉树
- 求二叉树中最远的两个节点的距离（二叉树的直径）

7.5 General Tree ADT

ADT for General Tree Nodes

```
// General tree node ADT
template <typename  E> class GTNode {
public:
    E value(); // Return node's value
    bool isLeaf(); // True if node is a leaf
    GTNode* parent();                   // Return parent
    GTNode* leftmostChild();            // Return first child
    GTNode* rightSibling();             // Return right sibling
    void setValue(E&);                  // Set node's value
    void insertFirst(GTNode<E>*);       // Insert first child
    void insertNext(GTNode<E>*);        // Insert next sibling
    void removeFirst();                 // Remove first child
    void removeNext();                  // Remove right sibling
};
```

General Tree ADT

```
// General tree ADT
template  <typename  E>  class  GenTree {
public:
    void clear();                    // Send all nodes to free store
    GTNode<E>* root();               // Return the root of the tree
            // Combine two subtrees
    void newroot(E&, GTNode<E>*, GTNode<E>*);
    void print();                    // Print a tree
};
```

General Tree Traversals

- For general trees, preorder and postorder traversals are defined with meanings similar to their binary tree counterparts.
- Preorder traversal of a general tree first visits the root of the tree, then performs a preorder traversal of each subtree from left to right.
- A postorder traversal of a general tree performs a postorder traversal of the root's subtrees from left to right, then visits the root.
- Inorder traversals are generally not useful with general trees.

General Tree Traversals

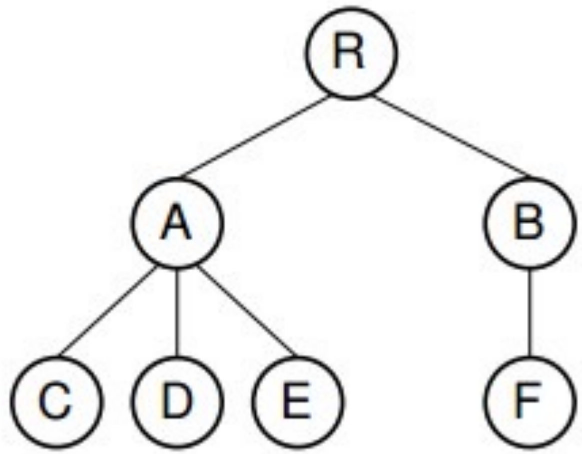


Figure 6.3 An example of a general tree.

- **preorder traversal : RACDEBF.**
- **postorder : CDEAFBR.**

General Tree Traversals

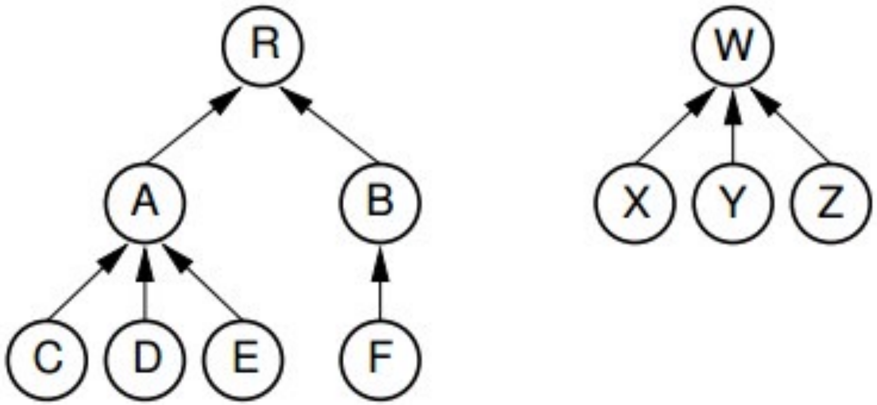
```
// Print using a preorder traversal
void printhelp(GTNode<E>* root) {
  if (root->isLeaf()) cout << "Leaf: ";
  else cout << "Internal: ";
  cout << root->value() << "\n";
  // Now process the children of "root"
  for (GTNode<E>* temp = root->leftmostChild();
       temp != NULL; temp = temp->rightSibling())
    printhelp(temp);
}
```

7.6 General Tree Implementations

Parent Point Implementation

- The simplest general tree implementation is to store for each node only a pointer to that node's parent. We will call this the parent pointer implementation.
- The parent pointer implementation stores precisely the information required to answer the following, useful question: "Given two nodes, are they in the same tree?"
- To answer the question, we need only follow the series of parent pointers from each node to its respective root. If both nodes reach the same root, then they must be in the same tree. If the roots are different, then the two nodes are not in the same tree.

Parent Pointer Implementation



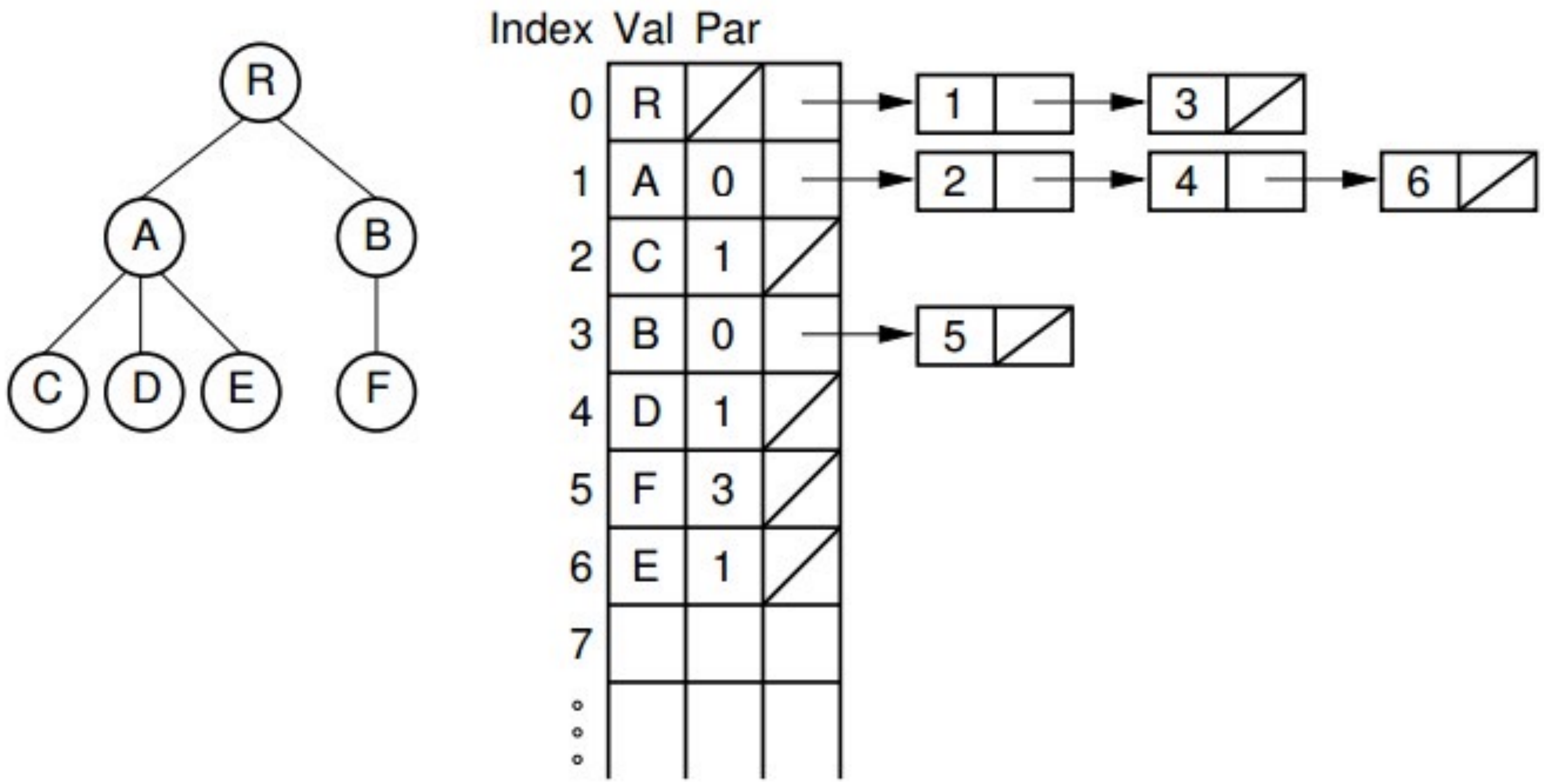
Parent's Index	/	0	0	1	1	1	2	/	7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

- **This implementation is not general purpose, because it is inadequate for such important operations as finding the leftmost child or the right sibling for a node.**

List of Children Implementation

- It simply stores with each internal node a linked list of its children.
- Each node contains a value, a pointer (or index) to its parent, and a pointer to a linked list of the node's children, stored in order from left to right.
- Each linked list element contains a pointer to one child.
- Thus, the leftmost child of a node can be found directly because it is the first element in the linked list.
- However, to find the right sibling for a node is more difficult.

List of Children Implementations



Dynamic Node Implementation

- The standard implementation for binary trees stores each node as a separate dynamic object containing its value and pointers to its two children.
- Unfortunately, nodes of a general tree can have any number of children, and this number may change during the life of the node.
- A general tree node implementation must support these properties.
- One solution is simply to limit the number of children permitted for any node and allocate pointers for exactly that number of children.

Dynamic Node Implementation

- There are two major objections to this.
  - First, it places an undesirable limit on the number of children, which makes certain trees un-representable by this implementation.
  - Second, this might be extremely wasteful of space because most nodes have fewer children than this limit.
- The alternative is to allocate variable space for each node. There are two basic approaches. One is to allocate an array of child pointers as part of the node. In essence, each node stores an array-based list of child pointers.

Dynamic Node Implementation

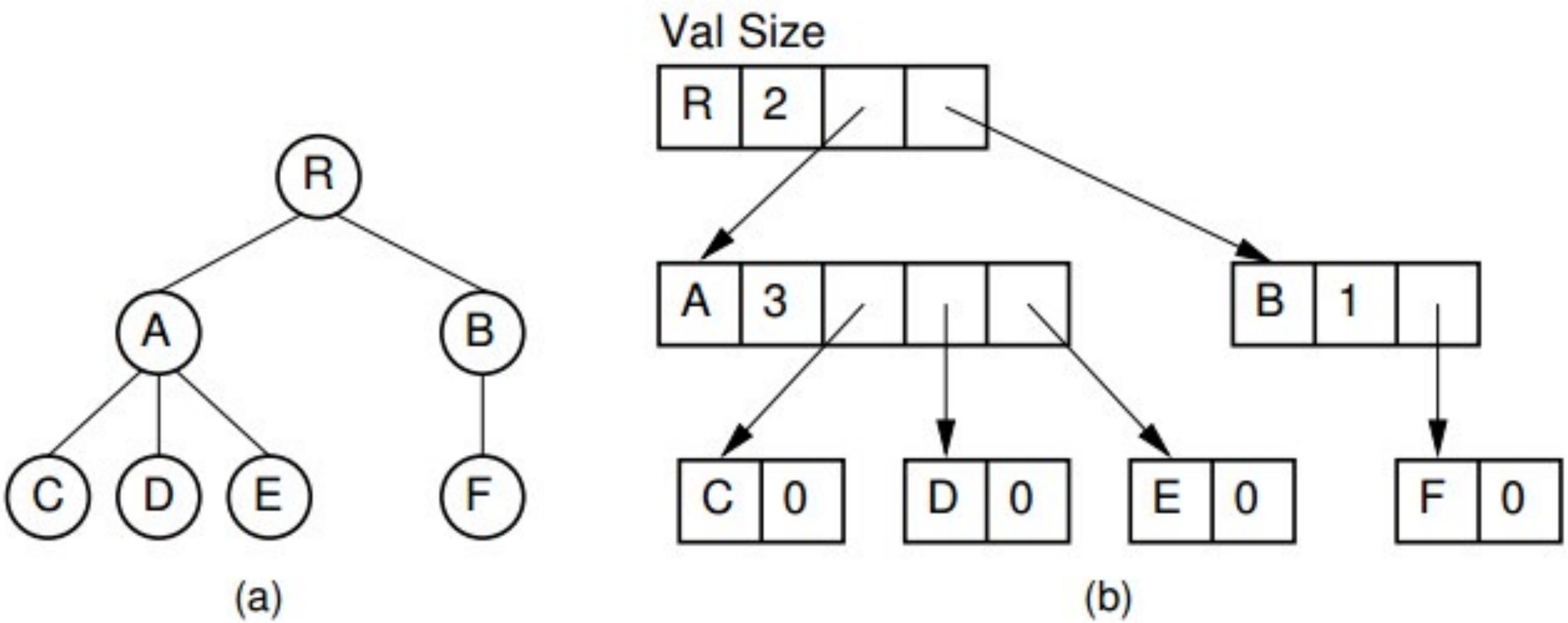


Figure 6.12 A dynamic general tree representation with fixed-size arrays for the child pointers. (a) The general tree. (b) The tree representation. For each node, the first field stores the node value while the second field stores the size of the child pointer array.

Dynamic Node Implementation

- Another way is to allocate a linked list of child pointers as part of the node.

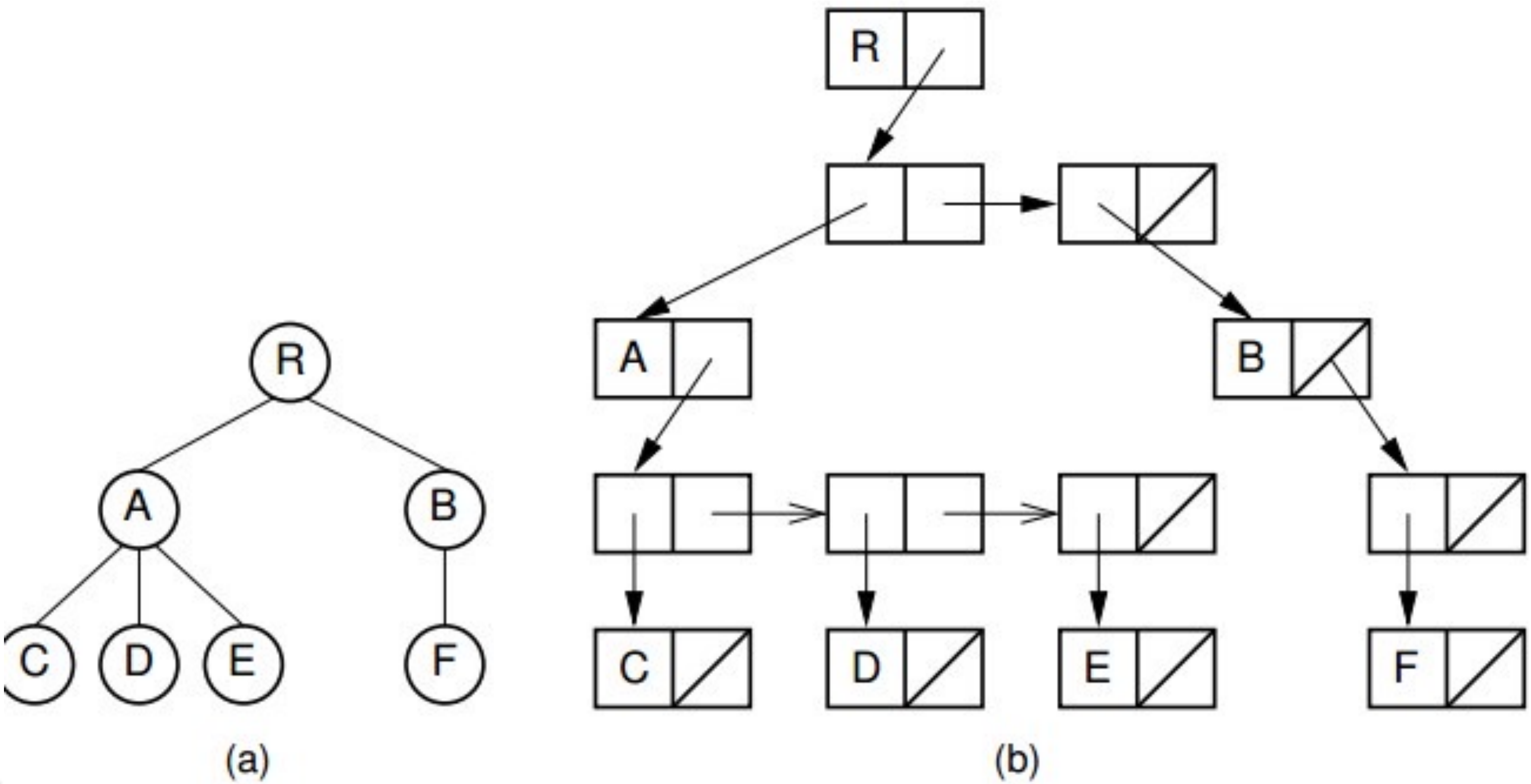
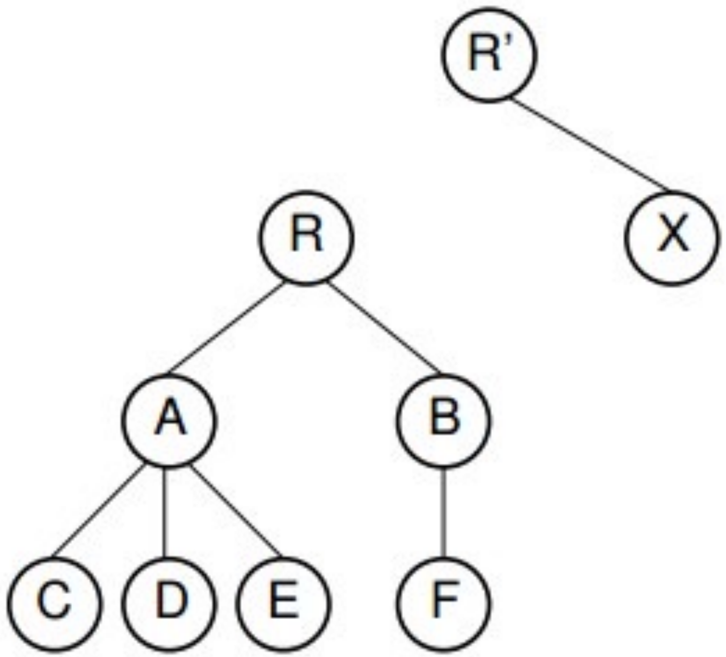


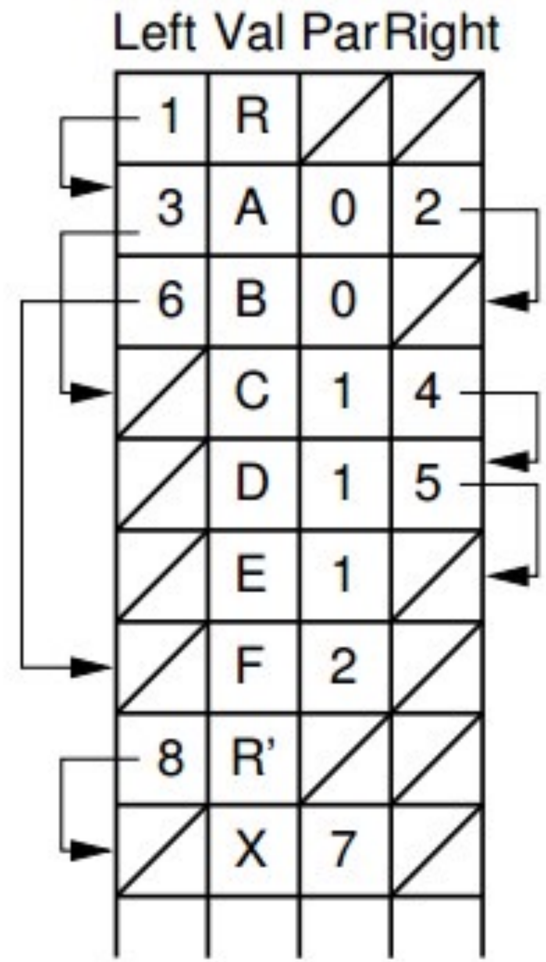
Figure 6.13 A dynamic general tree representation with linked lists of child pointers. (a) The general tree. (b) The tree representation.

Left-Child/Right-Sibling Implementation

- Each node stores its value and pointers to its parent, leftmost child, and right sibling.
- Thus, each of the basic ADT operations can be implemented by reading a value directly from the node.
- If two trees are stored within the same node array, then adding one as the subtree of the other simply requires setting three pointers
- This implementation is more space efficient than the "list of children" implementation, and each node requires a fixed amount of space in the node array

Left-Child/Right-Sibling Implementation





7.7 Converting Forest to Binary Tree

Converting forest to binary tree

- The “left-child/right-sibling” implementation stores a fixed number of pointers with each node. This can be readily adapted to a dynamic implementation. In essence, we substitute a binary tree for a general tree. Each node of the “left-child/right-sibling” implementation points to two “children” in a new binary tree structure.
- The left child of this new structure is the node’s first child in the general tree. The right child is the node’s right sibling.

Converting forest to binary tree

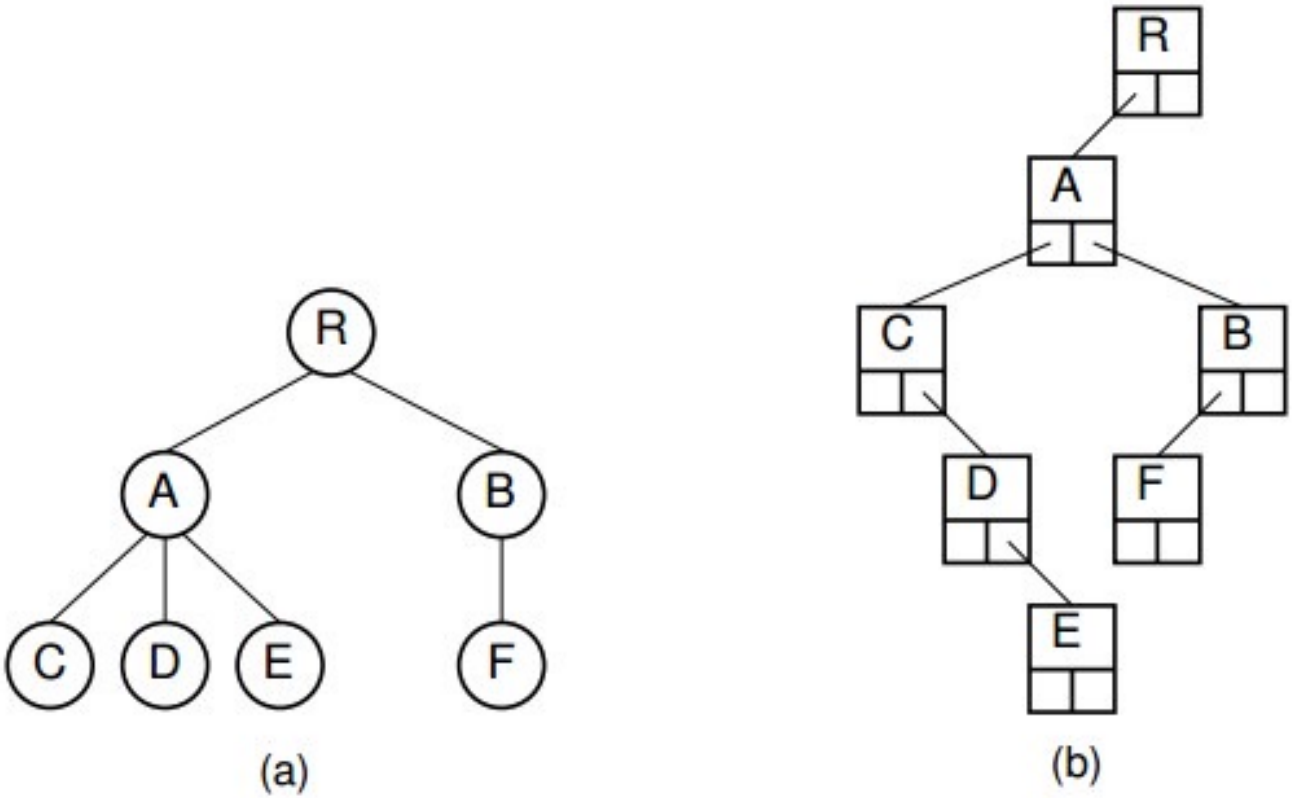


Figure 6.15 A general tree converted to the dynamic "left-child/right-sibling" representation. Compared to the representation of Figure 6.13, this representation requires less space.

Converting forest to binary tree

- **Here we simply include links from each node to its right sibling and remove links to all children except the leftmost child.**

- **Figure 6.15 shows how this might look in an implementation with two pointers at each node. The implementation of Figure 6.15 only requires two pointers per node.**

- **The representation of Figure 6.15 is likely to be easier to implement, space efficient, and more flexible than the other implementations presented in this section.**

- **We can easily extend this conversion to a forest of general trees, because the roots of the trees can be considered siblings.**

Converting forest to binary tree

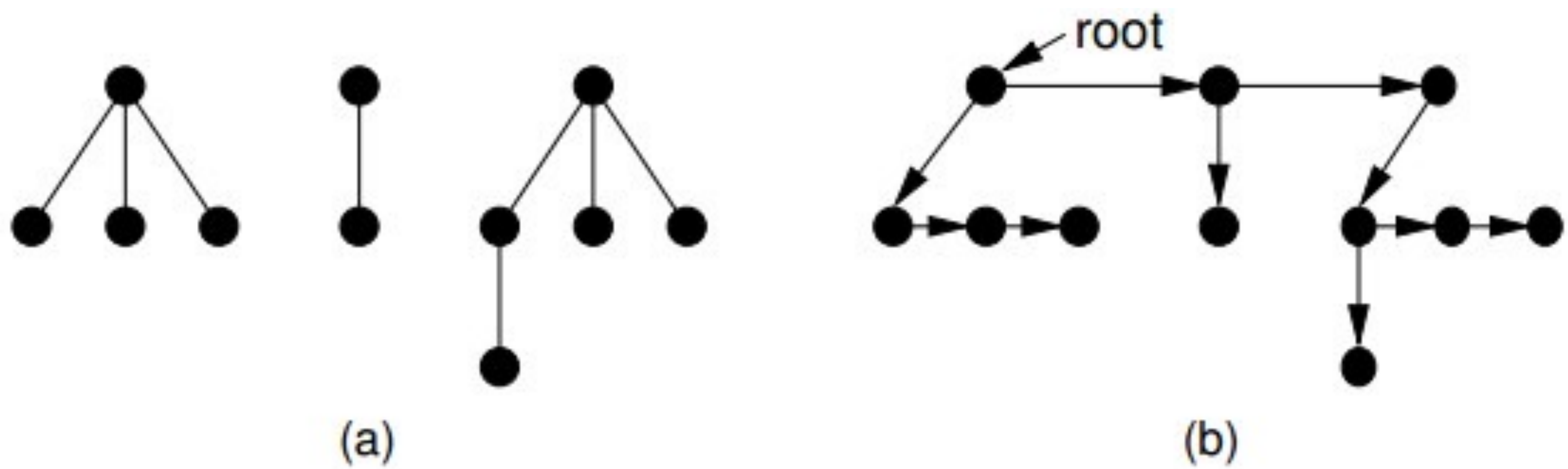


Figure 6.14 Converting from a forest of general trees to a single binary tree. Each node stores pointers to its left child and right sibling. The tree roots are assumed to be siblings for the purpose of converting.

《数据结构与算法》课程组

重庆大学计算机学院

End of Chapter