Lecture 8-1 - Data Structures

Introduction to Trees

1

Data Structures

Linear Form
(arrays, records, lists, stacks, queues)

- Unique first component
- Unique last component
- Every other has a unique predecessor and a unique successor

Nonlinear (hierarchical) form

- Each component may have several successors

Trees

- a tree represents a hierarchy

examples:

organization structure of a corporation

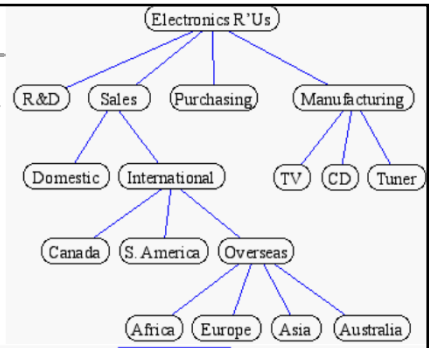
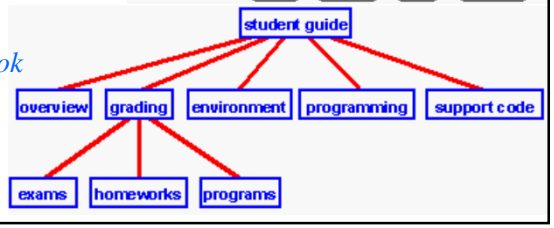
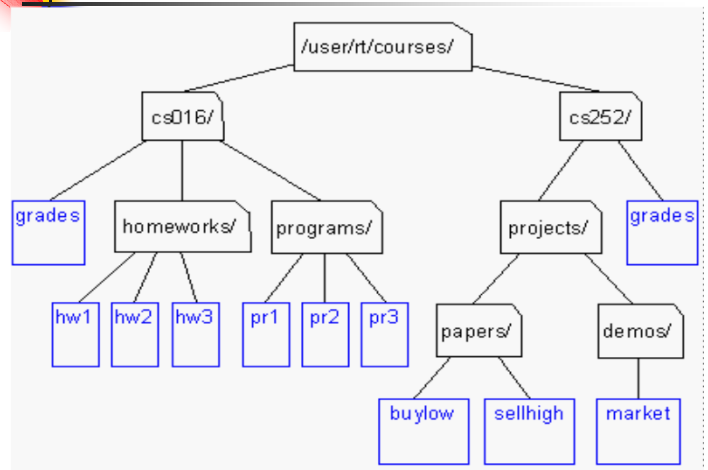


table of contents of a book



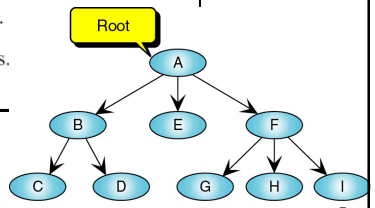
Unix or DOS/Windows file system



4

Tree Characteristics

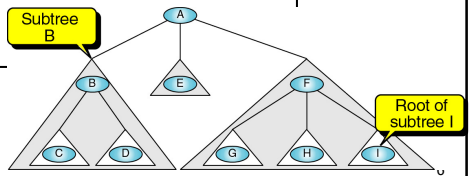
- A tree consists of a collection of nodes connected by directed arcs.
- A tree has a single *root* node.
  - By convention, the root node is usually drawn at the top.
- A node that points to (one or more) other nodes is the *parent* of those nodes while the nodes pointed to are the *children*.
- Every node (except the root) has exactly one parent.
- Nodes with no children are *leaf* nodes.
- Nodes with children are *interior* nodes.



5

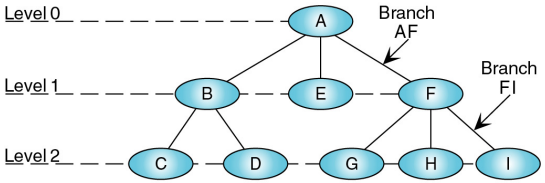
Tree Characteristics (cont.)

- Nodes that have the same parent are *siblings*.
- The *descendents* of a node consist of its children, and their children, and so on.
  - All nodes in a tree are descendents of the root node (except, of course, the root node itself).
- Any node can be considered the root of a *subtree*.
  - Like a subset, a subtree need not be "proper" (i.e., be strictly smaller than the original).
- A subtree rooted at a node consists of that node and all of its descendents.



6

- A node's *height* is equal to the maximum path length from that node to a leaf node:
  - A leaf node has a height of 0.
  - The height of a tree is equal to the height of the root.
- A node's *depth* is equal to the path length from the root to that node:
  - The root node has a depth of 0.
  - A tree's depth is the maximum depth of all its leaf nodes (which, of course, is equal to the tree's height).

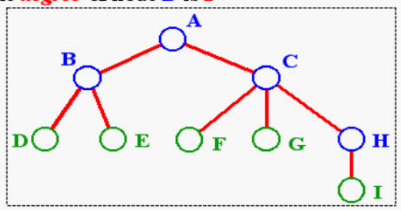


Parents: A, B, F
Children: B, E, F, C, D, G, H, I
Siblings: {B,E,F},{C,D}, {G,H,I}

Leaves C,D,E,G,H,I
Internal nodes B,F

7

- ***A*** is the ***root*** node.
- ***B*** is the ***parent*** of D and E.
- ***C*** is the ***sibling*** of B
- ***D*** and ***E*** are the ***children*** of B
- ***D, E, F, G, I*** are ***external nodes***, or ***leaves***
- ***A, B, C, H*** are ***internal nodes***
- The ***depth*** (***level***) of ***E*** is ***2***
- The ***height*** of the tree is ***3***
- The ***degree*** of node ***B*** is ***2***



Property: (# edges) = (#nodes) – 1

Degree of a node is the number of its children OR the number of arcs (branches) associated with it.

Indegree –

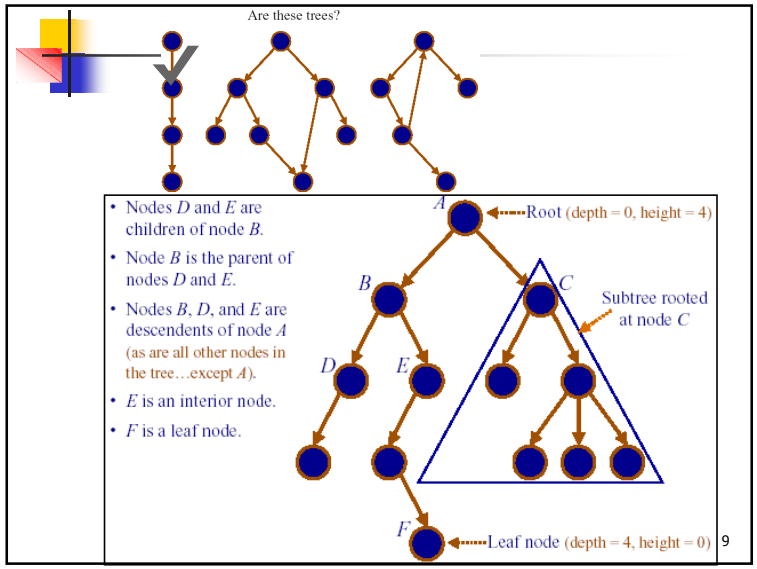
number of branches directed towards the node.

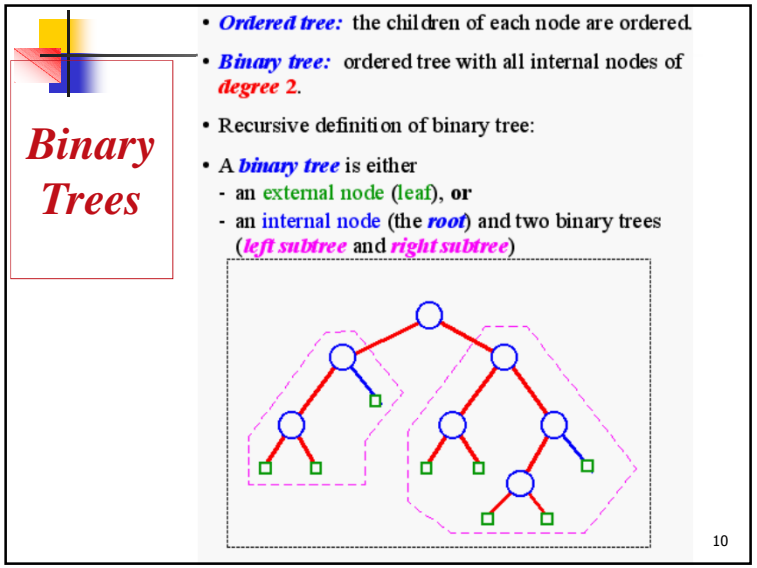
Outdegree –

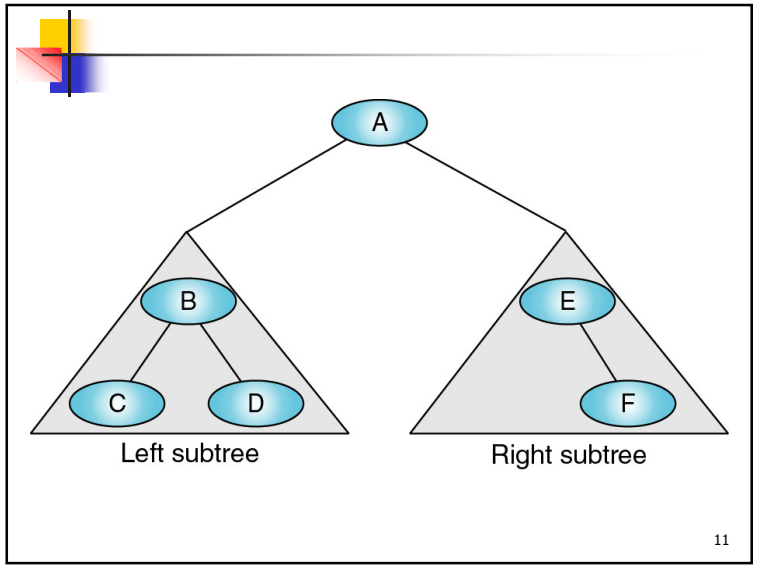
number of branches directed away from the node.

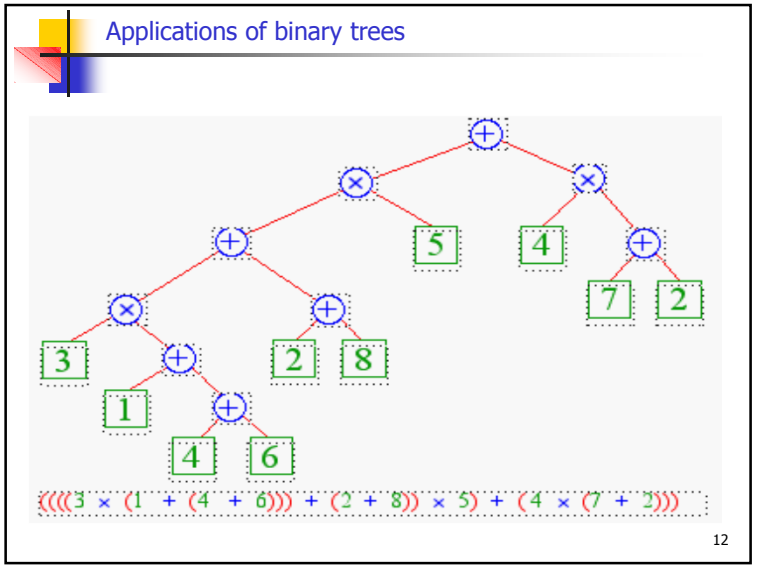
All nodes but the root must have indegree of exactly one.

8

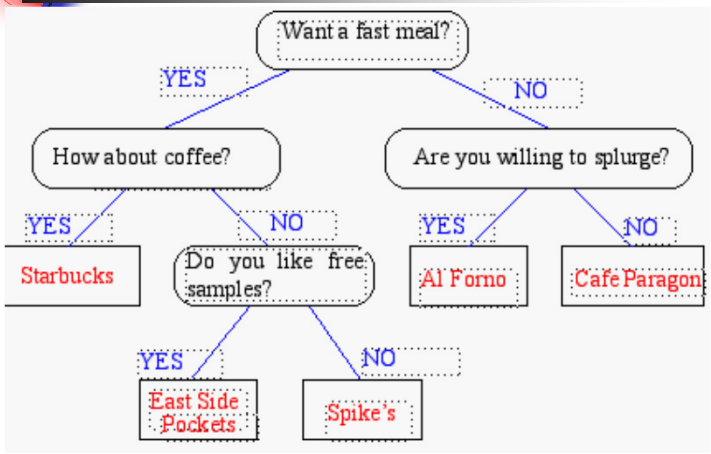

Are these trees?
Nodes D and E are children of node B.
Node B is the parent of nodes D and E.
Nodes B, D, and E are descendents of node A (as are all other nodes in the tree…except A).
E is an interior node.
F is a leaf node.
A
B
C
D
E
F
Root (depth = 0, height = 4)
Subtree rooted at node C
Leaf node (depth = 4, height = 0)
9

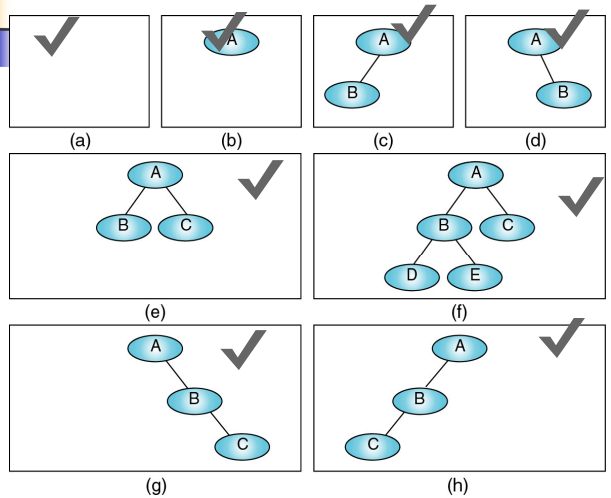

Binary Trees
Ordered tree: the children of each node are ordered.
Binary tree: ordered tree with all internal nodes of degree 2.
Recursive definition of binary tree:
A binary tree is either
- an external node (leaf), or
- an internal node (the root) and two binary trees (left subtree and right subtree)
10


A
B
C
D
E
F
Left subtree
Right subtree
11

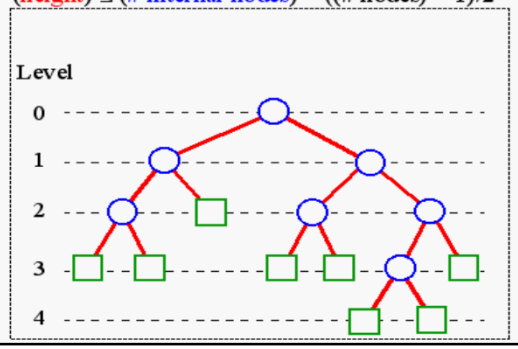

Applications of binary trees
(((((3 × (1 + (4 + 6))) + (2 + 8)) × 5) + (4 × (7 + 2)))
12

Applications of binary trees – contd.





- (# external nodes) = (# internal nodes) + 1
- (# nodes at level i) $\leq 2^{i}$
- (# external nodes) $\leq 2^{(\text{height})}$
- (height) $\geq \log_2$ (# external nodes)
- (height) $\geq \log_2$ (# nodes) – 1
- (height) $\leq$ (# internal nodes) = ((# nodes) – 1)/2



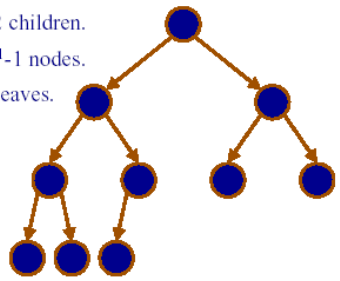
Binary Tree

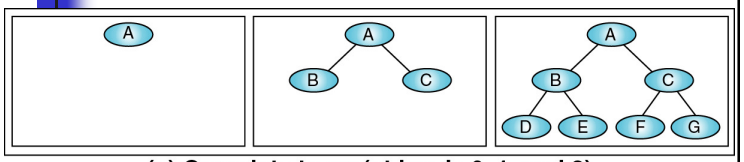
- Nodes have no more than two children:
  - Children are generally ordered from left to right.

Complete binary tree : every leaf is at the same depth.

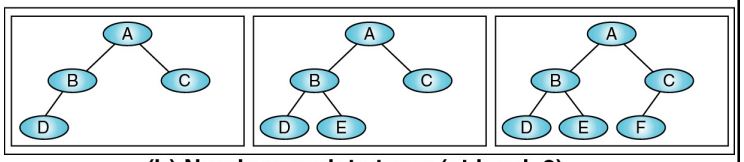
- Every internal node has 2 children.
- Height of n will have 2^{n+1}-1 nodes.
- Height of n will have 2^{n} leaves.

Nearly complete binary tree: full except for the bottom level which is filled from left to right.





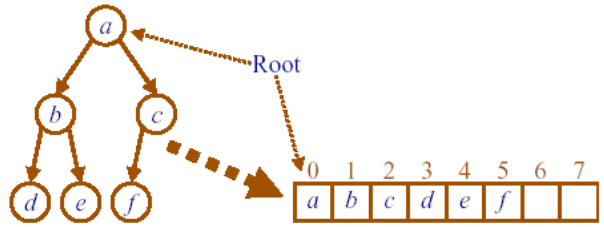
(a) Complete trees (at levels 0, 1, and 2)



(b) Nearly complete trees (at level 2)

Binary tree implementation

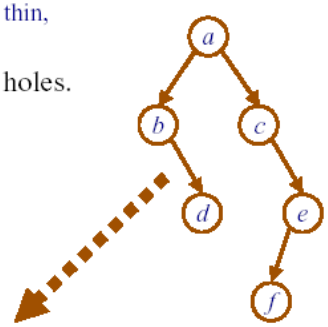
- Complete binary tree have structure that is efficiently implemented with a `Vector`:



  – Children of node i are stored at $2i + 1$ and $2i + 2$.
  – Parent of node i is at floor($(i - 1) / 2$).

Binary tree implementation

- If the tree is not complete (it is thin, unbalanced, etc.), the `Vector` implementation will be full of holes.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	c		d		e							f		

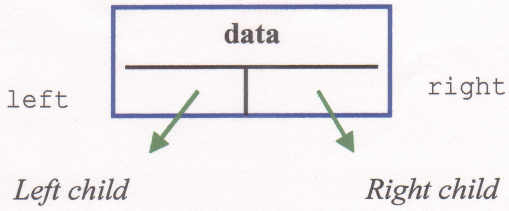
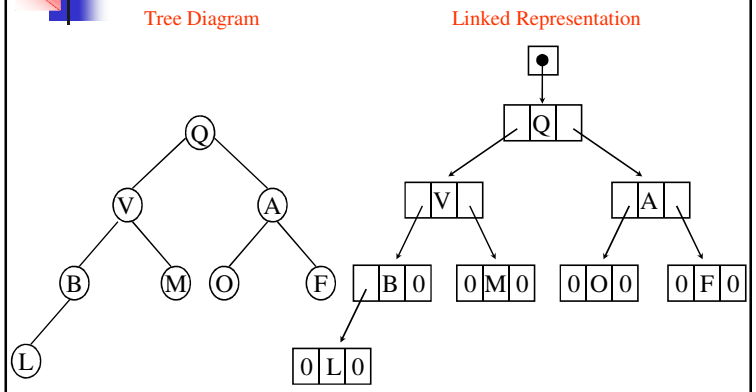


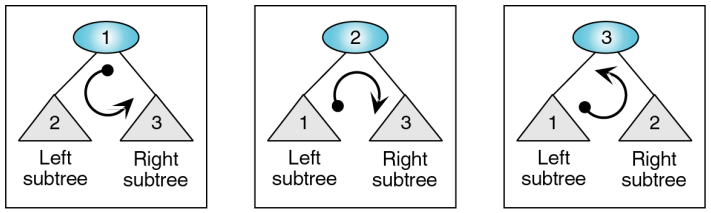
Fig. 9. Linked representation of binary tree

Linked representation of Binary Trees



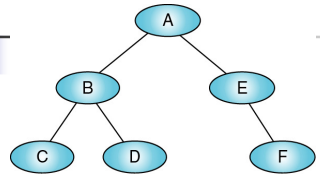
Common Binary Tree Operations

▪*Insert* Node	▪*Display* the binary tree
▪*Preorder* Traversal	▪Make a *copy*
▪*Inorder* Traversal	▪Determine whether two binary trees are *identical*
▪*Postorder* Traversal	▪*Delete* the tree
▪Determine *height*	
▪Determine the *number of elements* in it	

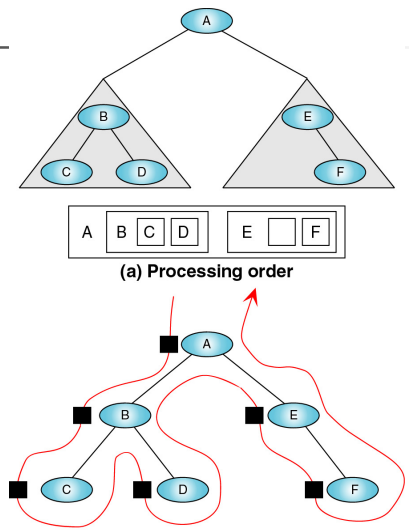


(a) Preorder traversal **(b) Inorder traversal** **(c) Postorder traversal**

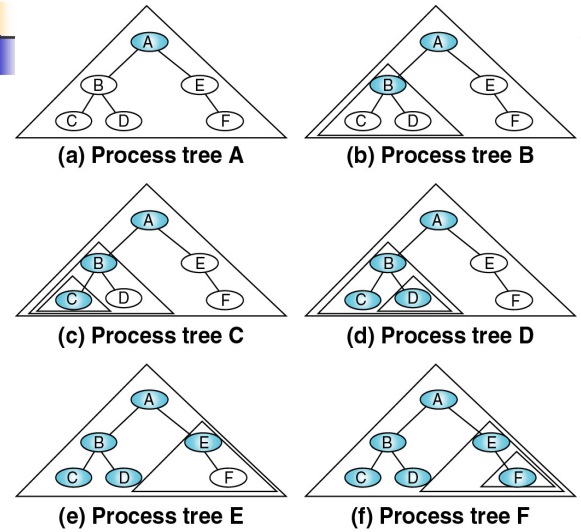
1. Node, left, right → Pre-order
2. Left, node, right → In-order
3. Left, right, node → Post-order



- **Preorder**
  - *Process the Value* in the node (e.g., Print the node)
  - Traverse the *left subtree* with a *preorder* traversal
  - Traverse the *right subtree* with a *preorder* traversal

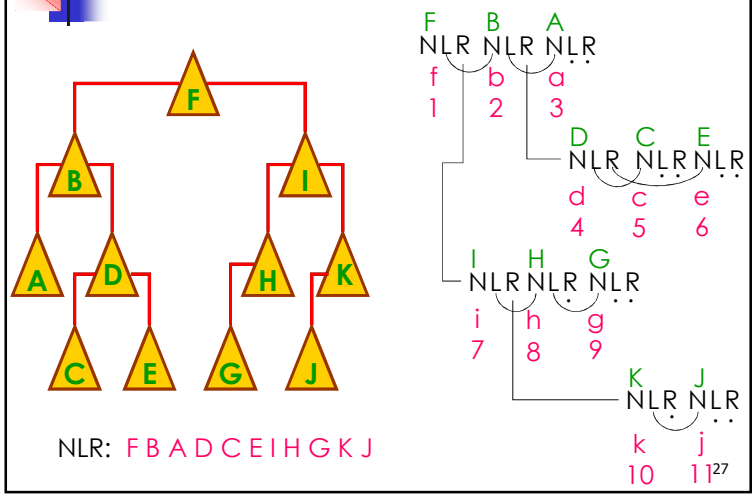


25



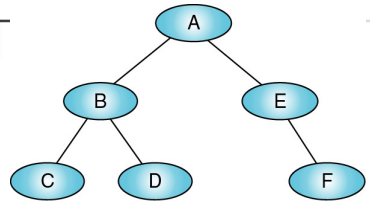
26

Preorder traversal: NLR



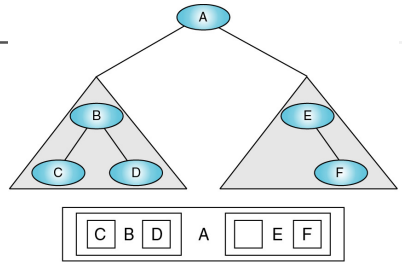
NLR: F B A D C E I H G K J

27

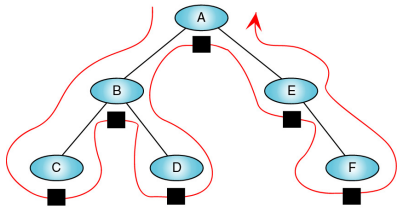


- **Inorder**
  - Traverse the *left subtree* with an *inorder* traversal
  - Process the *Value* in the node
  - Traverse the *right subtree* with an *inorder* traversal

28

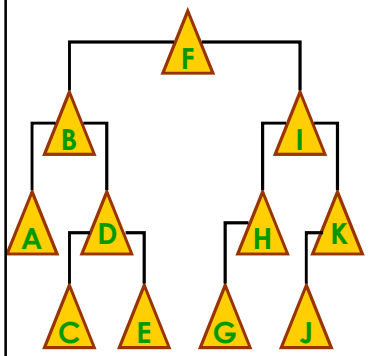


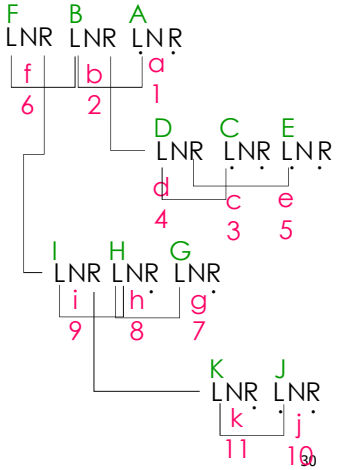
(a) Processing order



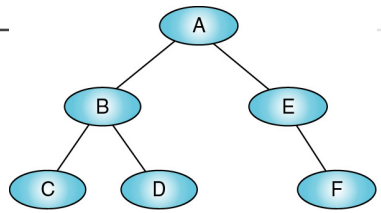
(b) "Walking" order

Inorder traversal: LNR

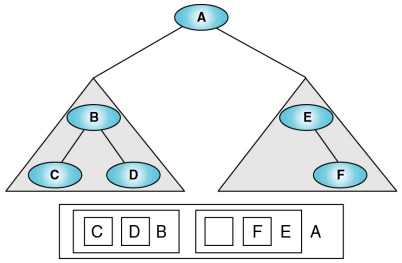




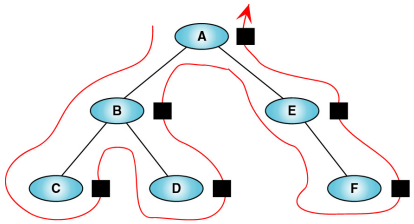
NLR: A B C D E F G H I J K



- **Postorder**
  - Traverse the *left subtree* with a *postorder* traversal
  - Traverse the *right subtree* with a *postorder* traversal
  - Process the *Value* in the node

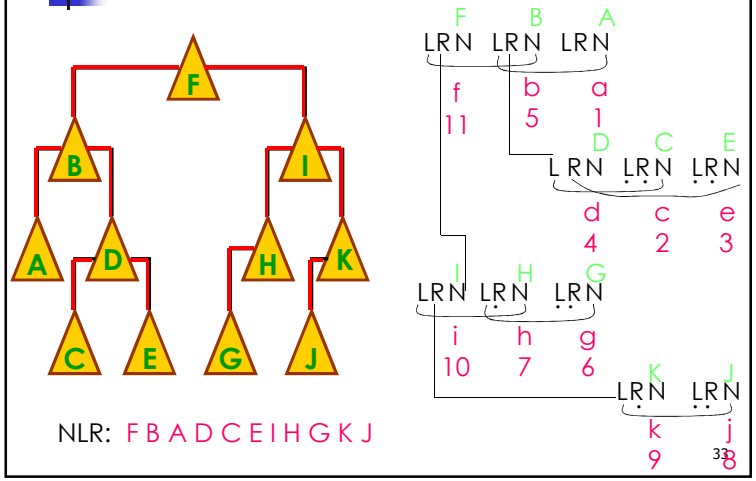


(a) Processing order



(b) "Walking" order

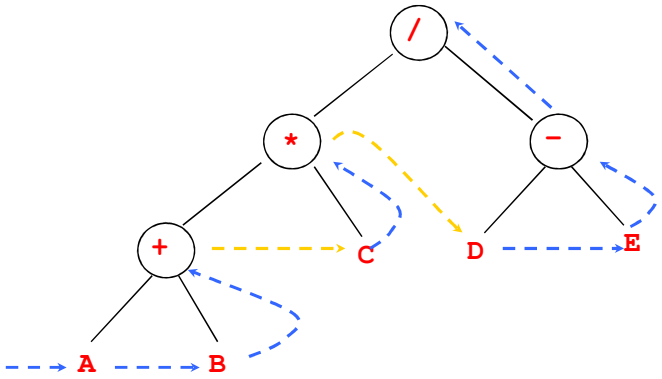
Postorder Traversal: LRN



NLR: F B A D C E I H G K J

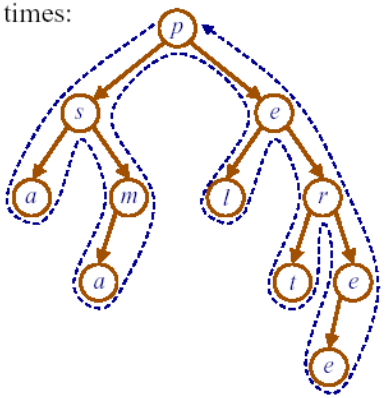
Traverse the tree in **Postorder:**

A B + C * D E − /

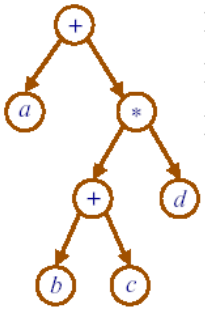


Binary Tree Traversals: Euler Tour

- An Euler Tour "walks" around the tree's perimeter.
- Each node is visited three times:
  - 1st visit: left side of node.
  - 2nd visit: bottom side of node.
  - 3rd visit: right side of node.
  - Leaf nodes are visited three times in succession.
- Traversal order depends on when node processed:
  - Pre-order: 1st visit.
  - In-order: 2nd visit.
  - Post-order: 3rd visit.



Traversal Example



Pre-order: + a * + b c d (Polish notation)

In-order: a + (b + c) * d (parenthesis added)

Post-order: a b c + d * + (reverse Polish notation)

Traversals

- Computational complexity:
  - Each traversal requires constant work at each node (not including the recursive calls).
  - Order not important.
  - Iterating over all n elements in a tree requires $O(n)$ time.

37

Post-Order Enumeration

- Use the `Enumeration` interface:
  - Current traversal implementations are recursive → can't be used to create an enumeration that returns a single element at each step.
- Initialization: `slideLeft` – push the path from the root to the left-most leaf onto a `Stack`.
- For `nextElement`:
  - Pop node off the top of the stack.
  - Call `slideLeft` on node's right child (push the path to the *next* left-most leaf node).
  - Return node's object.
- No more elements when the stack is empty.

Post-Order Enumeration (cont.)

```
class PostorderTreeTraversal implements Enumeration {
  private Stack stk = new Vector();

  public PostorderTreeTraversal(BinaryNode root) {
    slideLeft(root);
  }

  private void slideLeft(BinaryNode node) {
    while (node != null){
      stk.addLast(node);  node = node.left;
    }
    if (stk.isEmpty()) return;

    node = (BinaryNode)stk.getLast();
    if (node.rght != null) slideLeft(node.rght);
  }

  public boolean hasMoreElements() {
    return !stk.isEmpty();
  }
```

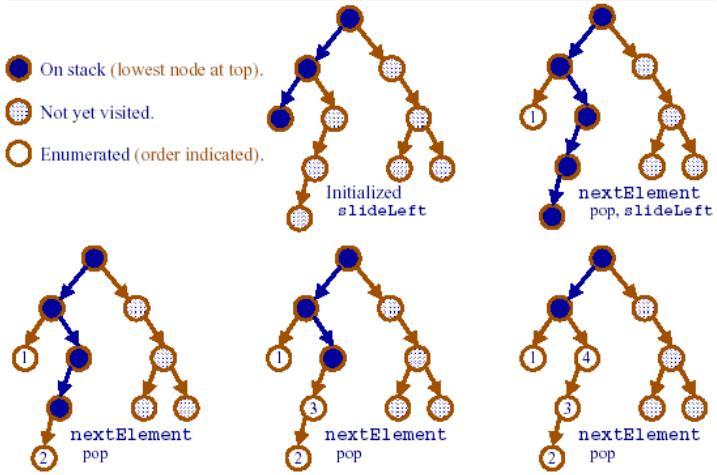
Post-Order Enumeration (cont.)

```
  public Object nextElement() {
    BinaryNode node = (BinaryNode)stk.getLast();
    stk.removeLast();

    if (!stk.isEmpty()) {
      BinaryNode parent = (BinaryNode)stk.getLast();
      if (parent.rght != node)        // Push path to left-most
        slideLeft(parent.rght);       //   leaf of right child.
    }
    return node.obj;
  }
}
```

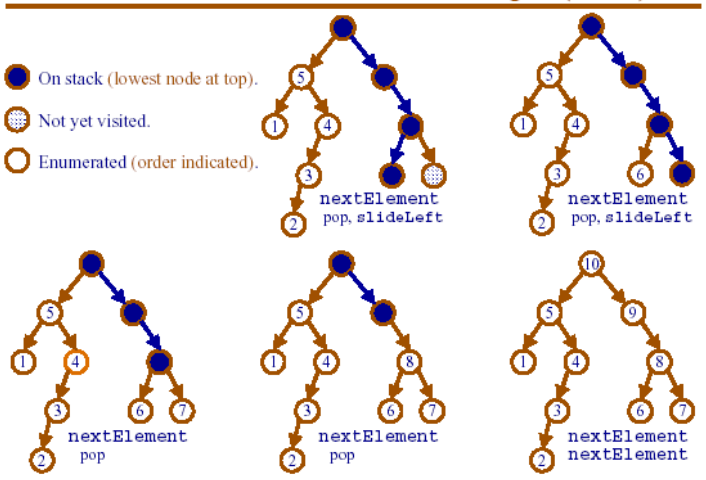
Execution time:

- Badly skewed tree (i.e., only one child per interior node) → $O(n)$ time for single call to `nextElement` to push the path of the leaf-most leaf.
- Would seem that worst case (for n calls of `nextElement`) would be $O(n^2)$.
- Actually, it is $O(n)$ in all cases.

Post-Order Enumeration: Example



Post-Order Enumeration: Example (cont.)



Pre-Order Enumeration

- Initialization: push the root node onto the **`Stack`**.
- For **`nextElement`**:
  - Get the node on the top of the stack (but don't remove it) and save it to return node's object at the end of method.
  - If it has a left child, push it onto the stack.
  - Otherwise:
    - Pop elements until you find a node that has a right child.
    - Push right child onto the stack

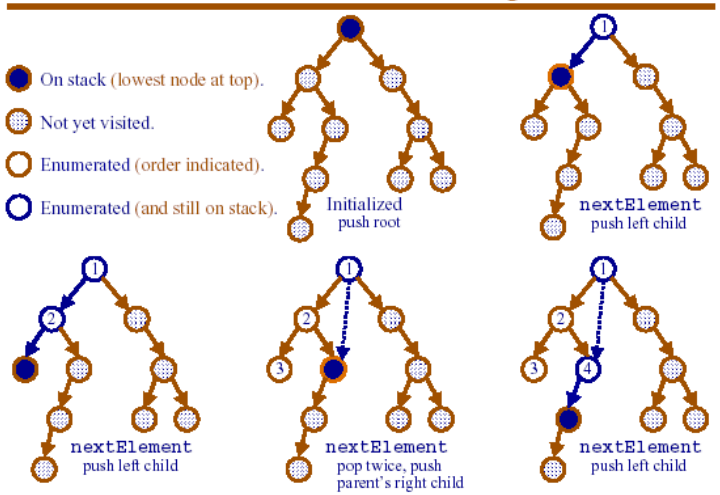
```
class PreorderTreeTraversal implements Enumeration {
  private Stack stk = new Vector();

  public PreorderTreeTraversal(BinaryNode root) {
    stk.addLast(root);              // Root node is first in pre-order traversal.
  }
```

Pre-Order Enumeration (cont.)

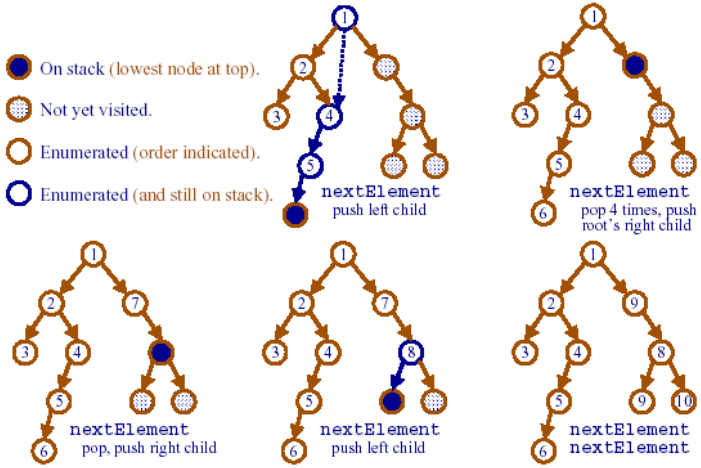
```
  public boolean hasMoreElements() { return !stk.isEmpty(); }

  public Object nextElement() {
    BinaryNode node = (BinaryNode)stk.getLast();

    if (node.left != null) stk.addLast(node.left);
    else {
      BinaryNode parent = node;
      stk.removeLast();
      while (parent.rght == null) {
        if (stk.isEmpty()) return node.value;

        parent = (BinaryNode)stk.getLast();
        stk.removeLast();
      }
      stk.addLast(parent.rght);
    }
    return node.obj;
  }
}
```

Execution time: as with Post-order enumeration → O(n).

Pre-Order Enumeration: Example



Pre-Order Enumeration: Example (cont.)



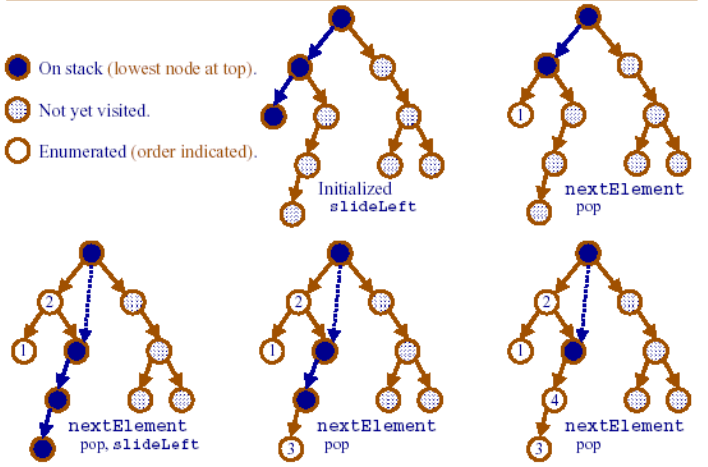
In-Order Enumeration

- Initialization: **`slideLeft`** – push the path from the root to the left-most *node* (not leaf) onto a **`Stack`**.
- For **`nextElement`**:
  - Pop node of top of stack.
  - Call **`slideLeft`** on node's right child (push the path to the *next* left-most *node*).
  - Return node's object.

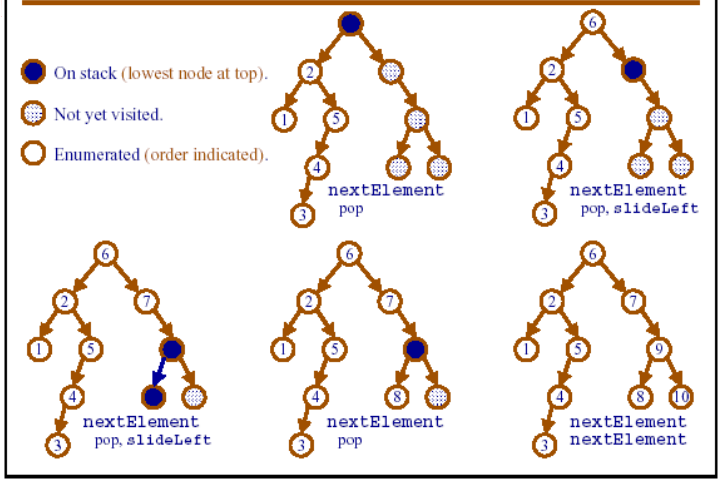
In-Order Enumeration (cont.)

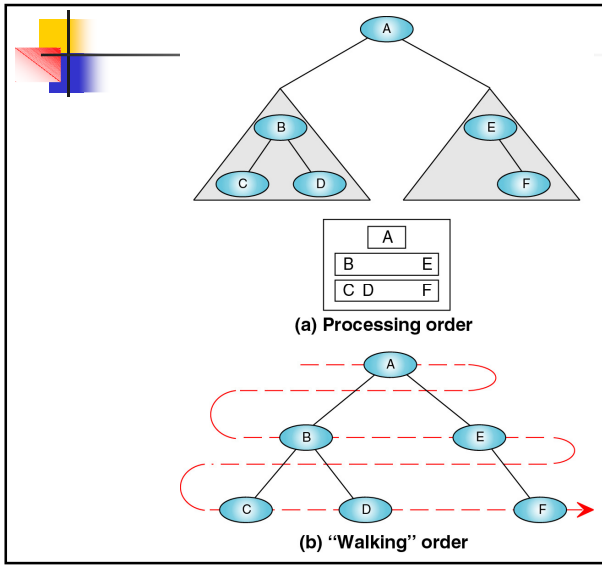
```
class InorderTreeTraversal implements Enumeration {
  private Stack stk = new Vector();

  public PreorderTreeTraversal(BinaryNode root) {
    slideLeft(root);            // Root node is first in pre-order traversal.
  }

  public boolean hasMoreElements() { return !stk.isEmpty(); }

  public Object nextElement() {
    BinaryNode node = (BinaryNode)stk.getLast();
    stk.removeLast();
    slideLeft(node.rght);
    return node.obj;
  }

  private void slideLeft(BinaryNode node) {
    while (node != null) {
      stk.addLast(node);  node = node.left;
    }
  }
}
```

In-Order Enumeration: Example



In-Order Enumeration: Example (cont.)





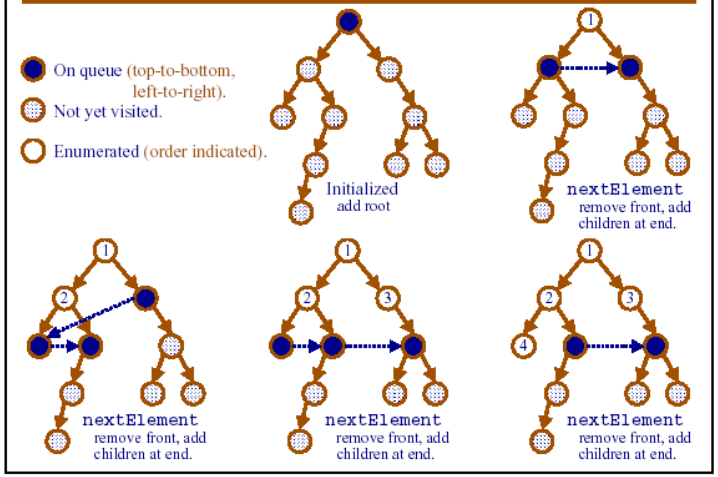
Level-Order Enumeration

- Haven't seen this traversal yet:
  - Traverses node a level at a time from left to right.
  - Start with root level and then traverse it children and then their children and so on.
  - Uses a `Queue` instead of a `Stack`.
- Initialization: place root node on `Queue`.
- For `nextElement`:
  - Take node from front of `Queue`.
  - Place node's children on `Queue` (left child first, then right).
  - Return node's object.

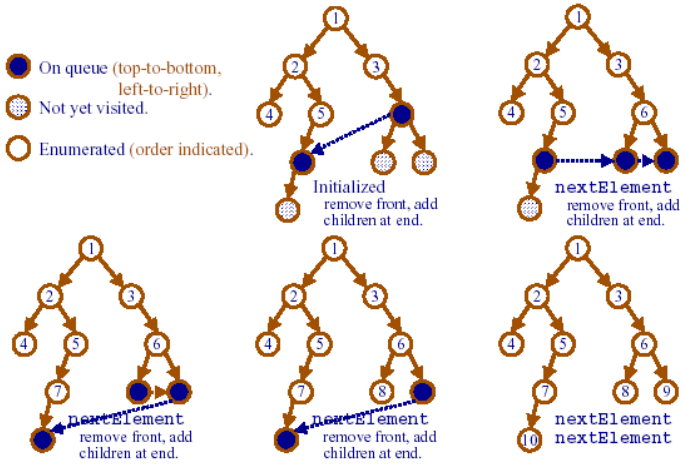
Level-Order Enumeration (cont.)

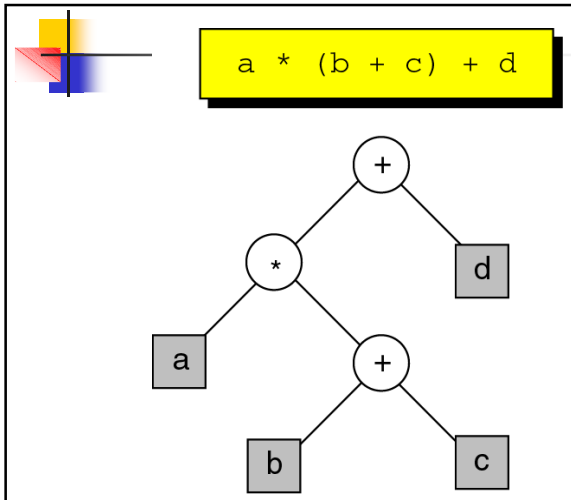
```
class LevelorderTreeTraversal implements Enumeration {
  private Queue que = new LinkedList();

  public PreorderTreeTraversal(BinaryNode root) {
    addLast(root);              // Root node is first in level-order traversal.
  }

  public boolean hasMoreElements() { return !que.isEmpty(); }

  public Object nextElement() {
    BinaryNode node = (BinaryNode)que.getFirst();
    que.removeFirst();
    if (node.left != null) que.addLast(node.left);
    if (node.rght != null) que.addLast(node.rght);
    return node.obj;
  }
}
```

Level-Order Enumeration: Example

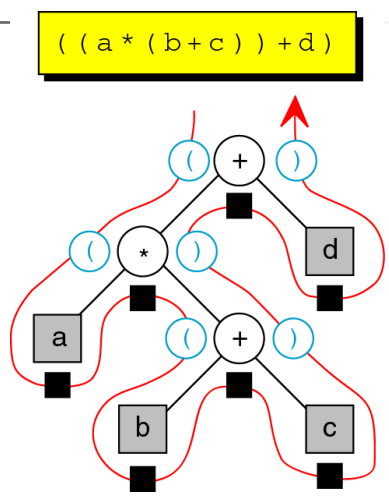


Level-Order Enumeration: Example





56

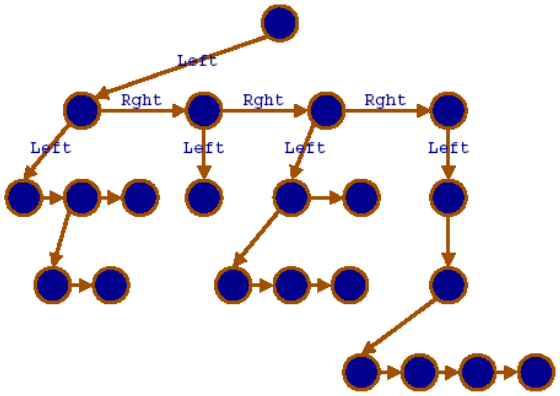


57

Representing General Trees

- Binary trees are the most common type found in CS:
  - However, it is sometimes useful to allow more than two children per node.
- General trees (any number of children per node):
  - In-order traversal is not well defined.
  - Pre-order, post-order, and level-order will still work
- Any general tree can be represented using a binary tree:
  - A node's **`left`** data member refers to its *first child*.
  - A node's **`rght`** data member refers to its *next sibling*.

General Tree: Binary Tree Representation

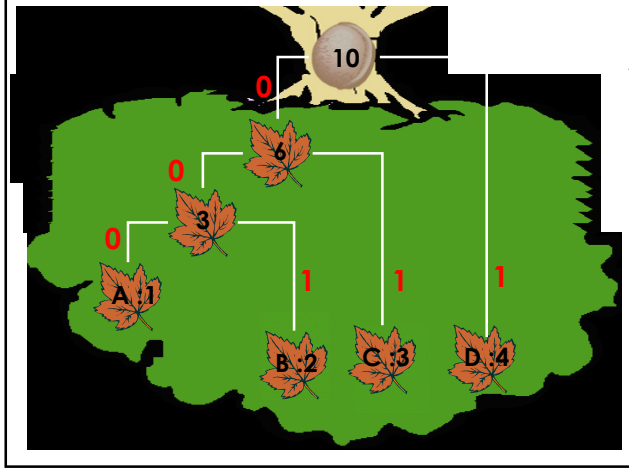


59

Huffman code:

❖ In 1952, A graduate student D.A. Huffman developed a compression code that shrank the amount of memory that was needed to store character values. He took advantage of the binary trees, by placing 0 and 1 to the left and right branches of the subtrees respectively.

60



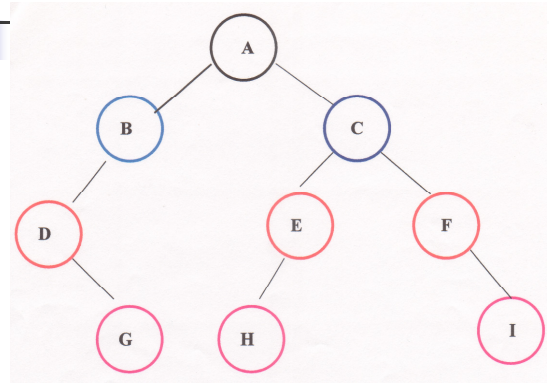
A = 000

B = 001

C = 01

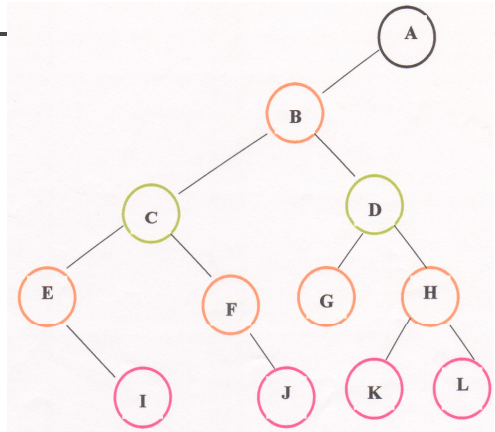
D =1

61



Preorder: A B D G C E H I F
Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

62



Preorder: A B C E I F J D G H K L
Inorder: E I C F J B G D K H L A
Postorder: I E J F C G K L H D B A

63

```
//Header File Binary Tree
#ifndef H_binaryTree
#define H_binaryTree

#include <iostream>

using namespace std;

    //Definition of the node
template<class elemType>
struct nodeType
{
  elemType                  info;
  nodeType<elemType>        *llink;
  nodeType<elemType>        *rlink;
};
```

64

```
//Definition of the class
template <class elemType>
class binaryTreeType
{
public:
   const binaryTreeType<elemType>& operator=
            (const binaryTreeType<elemType>&);
    //Overload the assignment operator.
   bool isEmpty();
    //Function to determine if the binary tree is empty.
    //Postcondition: Returns true if the binary tree is empty;
    //              otherwise, returns false.
   void inorderTraversal();
    //Function to do an inorder traversal of the binary tree.
    //Postcondition: The nodes of the binary tree are output
    //              in the inorder sequence.
   void preorderTraversal();
    //Function to do a preorder traversal of the binary tree.
    //Postcondition: The nodes of the binary tree are output
    //              in the preorder sequence.
   void postorderTraversal();
    //Function to do a postorder traversal of the binary tree.
    //Postcondition: The nodes of the binary tree are output
    //              in the postorder sequence.
```



```
   int treeHeight();
     //Function to deteremine the height of the binary tree.
     //Postcondition: The height of the binary tree is returned.
   int treeNodeCount();
     //Function to determine the number of nodes in the
     //binary tree.
     //Postcondition: The number of nodes in the binary tree
     //              is returned.
   int treeLeavesCount();
     //Function to determine the number of leaves in the
     //binary tree.
     //Postcondition: The number of leaves in the binary tree
     //              is returned.
   void destroyTree();
     //Deallocates the memory space occupied by the binary tree.
     //Postcondition: root = NULL;

   binaryTreeType(const binaryTreeType<elemType>& otherTree);
     //copy constructor

   binaryTreeType();
     //default constructor

   ~binaryTreeType();
     //destructor
```


New functions

```
void nonRecursiveInTraversal();

void nonRecursivePreTraversal();

void nonRecursivePostTraversal();
```



```
protected:
   nodeType<elemType> *root;

private:
   void copyTree(nodeType<elemType>* &copiedTreeRoot,
            nodeType<elemType>* otherTreeRoot);
     //Function to make a copy of the binary tree to
     //which otherTreeRoot points.
     //Postcondition: The pointer copiedTreeRoot points to
     //              the root of the copied binary tree.

   void destroy(nodeType<elemType>* &p);
     //Function to destroy the binary tree to which p points.
     //Postcondition: The nodes of the binary tree to which
     //              p points are deallocated.
     //              p = NULL.

   void inorder(nodeType<elemType> *p);
     //Function to do an inorder traversal of the binary
     //tree to which p points.
     //Postcondition: The nodes of the binary tree to which p
     //              points are output in the inorder sequence.
```

```
void preorder(nodeType<elemType> *p);
    //Function to do a preorder traversal of the binary
    //tree to which p points.
    //Postcondition: The nodes of the binary tree to which p
    //               points are output in the preorder sequence.
  void postorder(nodeType<elemType> *p);
    //Function to do a postorder traversal of the binary
    //tree to which p points.
    //Postcondition: The nodes of the binary tree to which p
    //               points are output in the postorder sequence.

  int height(nodeType<elemType> *p);
    //Function to determine the height of the binary tree
    //to which p points.
    //Postcondition: The height of the binary tree to which p
    //               points is returned.

  int max(int x, int y);
    //Function to determine the larger of x and y.
    //Postcondition: The larger of x and y is returned.
```

69

```
int nodeCount(nodeType<elemType> *p);
    //Function to determine the number of nodes in the binary
    //tree to which p points.
    //Postcondition: The number of nodes in the binary tree
    //               to which p points is returned.

  int leavesCount(nodeType<elemType> *p);
    //Function to determine the number of leaves in the binary
    //tree to which p points.
    //Postcondition: The number of nodes in the binary tree
    //               to which p points is returned.
};
```

70

```
//Definition of member functions

template<class elemType>
binaryTreeType<elemType>::binaryTreeType()
{
    root = NULL;
}

template<class elemType>
bool binaryTreeType<elemType>::isEmpty()
{
    return (root == NULL);
}

template<class elemType>
void binaryTreeType<elemType>::inorderTraversal()
{
    inorder(root);
}

template<class elemType>
void binaryTreeType<elemType>::preorderTraversal()
{
    preorder(root);
}

template<class elemType>
void binaryTreeType<elemType>::postorderTraversal()
{
    postorder(root);
}
```

71

```
template<class elemType>
int binaryTreeType<elemType>::treeHeight()
{
    return height(root);
}

template<class elemType>
int binaryTreeType<elemType>::treeNodeCount()
{
    return nodeCount(root);
}

template<class elemType>
int binaryTreeType<elemType>::treeLeavesCount()
{
    return leavesCount(root);
}
```

72

```
template <class elemType>
void  binaryTreeType<elemType>::copyTree
                (nodeType<elemType>* &copiedTreeRoot,
                   nodeType<elemType>* otherTreeRoot)
{
   if(otherTreeRoot == NULL)
         copiedTreeRoot = NULL;
   else
   {
         copiedTreeRoot = new nodeType<elemType>;
         copiedTreeRoot->info = otherTreeRoot->info;
         copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
         copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
   }
} //end copyTree
```

73

```
template<class elemType>
void binaryTreeType<elemType>::inorder(nodeType<elemType> *p)
{   if(p != NULL)
    {
          inorder(p->llink);
          cout<<p->info<<" ";
          inorder(p->rlink);
    }
}
template<class elemType>
void binaryTreeType<elemType>::preorder(nodeType<elemType> *p)
{   if(p != NULL)
    {     cout<<p->info<<" ";
          preorder(p->llink);
          preorder(p->rlink);
    }
}
template<class elemType>
void binaryTreeType<elemType>::postorder(nodeType<elemType> *p)
{   if(p != NULL)
    {     postorder(p->llink);
          postorder(p->rlink);
          cout<<p->info<<" ";
    }
}
```

```
//Overload the assignment operator
template<class elemType>
const binaryTreeType<elemType>& binaryTreeType<elemType>::
         operator=(const binaryTreeType<elemType>& otherTree)
{

   if(this != &otherTree)                  //avoid self-copy
   {
         if(root != NULL)                  //if the binary tree is not empty,
                                            //destroy the binary tree
                  destroy(root);

         if(otherTree.root == NULL)        //otherTree is empty
                  root = NULL;
         else
                  copyTree(root, otherTree.root);
   }     //end else

  return *this;
}
```

75

```
template <class elemType>
void  binaryTreeType<elemType>::destroy(nodeType<elemType>* &p)
{   if(p != NULL)
    {     destroy(p->llink);
          destroy(p->rlink);
          delete p;
          p = NULL;
    }
}
template <class elemType>
void  binaryTreeType<elemType>::destroyTree()
{   destroy(root);
}
   //copy constructor
template <class elemType>
binaryTreeType<elemType>::binaryTreeType
          (const binaryTreeType<elemType>& otherTree)
{   if(otherTree.root == NULL)                    //otherTree is empty
          root = NULL;
    else
          copyTree(root, otherTree.root);
}
```

```
template <class elemType>
binaryTreeType<elemType>::~binaryTreeType()
{   destroy(root);
}
template<class elemType>
int binaryTreeType<elemType>::height(nodeType<elemType> *p)
{
    if(p == NULL)
        return 0;
    else
        return 1 + max(height(p->llink), height(p->rlink));
}
template<class elemType>
int binaryTreeType<elemType>::max(int x, int y)
{
    if(x >= y)
        return x;
    else
        return y;
}
```

77

```
template<class elemType>
int binaryTreeType<elemType>::nodeCount(nodeType<elemType> *p)
{
    cout<<"Write the definition of the function nodeCount"
        <<endl;

    return 0;
}

template<class elemType>
int binaryTreeType<elemType>::leavesCount(nodeType<elemType> *p)
{
    cout<<"Write the definition of the function leavesCount"
        <<endl;

    return 0;
}

#endif
```

NON recursive traversal implementation

79

```
//NonRecursive Traversal algorithms

template <class elemType>
void binaryTreeType<elemType>::nonRecursiveInTraversal()
{
    stackType<nodeType<elemType>* > stack;
    nodeType<elemType> *current;
    current = root;

    while((current != NULL) || (!stack.isEmptyStack()))
        if (current != NULL)
        {       stack.push(current);
                current = current->llink;
        }
        else
        {       current = stack.top();
                stack.pop();
                cout<<current->info<<" ";
                current = current->rlink;
        }
    cout<<endl;
}
```

80

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursivePreTraversal()
{
    stackType<nodeType<elemType>*> stack;
    nodeType<elemType> *current;

    current = root;

    while((current != NULL) || (!stack.isEmptyStack()))
        if(current != NULL)
        {       cout<<current->info<<" ";
                stack.push(current);
                current = current->llink;
        }
        else
        {       current = stack.top();
                stack.pop();
                current = current->rlink;
        }
    cout<<endl;
}
```

81

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursivePostTraversal()
{
    cout<<"Implement this method " << endl;
}
```

82

Client example

83

Test for Binary trees

```
#include <iostream>

using namespace std;

int main()
{
    cout<<"Cannot build the tree without the insert function."<<endl ;

    return 0;
}
```

84