TREE
- Nonlinear Data Structure

- Deepali Londhe

1

UNIT – IV TREES

- **Tree :** Trees and binary trees-concept and terminology, Expression tree, Binary tree as an ADT, , Binary search tree, Recursive and Non recursive algorithms for binary tree traversals ,Binary search tree as ADT(Insert Search Delete, level wise Display)
- **Threaded binary tree:** Concept of threaded binary tree (inorder, preorder and postorder). Preorder and In-order traversals of in-order threaded binary tree, Applications of trees.

2

Linear Vs Non Linear Data Structures

1. A data structure is linear if every item is related (or attached) to its previous and next item e.g. array, linked list)

   It is non-linear if every item is attached to many other items in specific ways to reflect relationships (e.g, n-ary tree).

2. In linear data structure data items are arranged in a linear sequence. i.e can have only single successor predecessor

   In non-linear data structure data items are not in a sequence.

   i.e can have more than one successor and predecessor.

3

Linear Vs Non Linear Data Structures

3. Examples :

Linear data struc

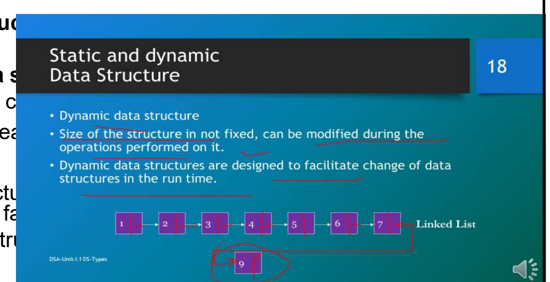
Non linear data s

4. Data being linear c

Data being nonlinea

5. Linear data structu possible in linear fa

Non linear data stru example:- trees

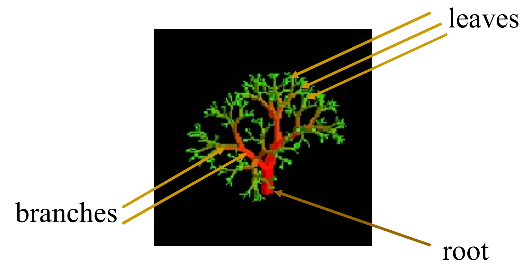


One thing remember-- array is always a static data structure and link list is always a dynamic data structure but others are dependent on array and link list, which is used. .

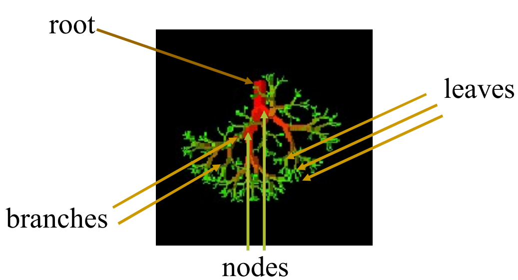
4

Trees and Binary Trees

Nature View of a Tree

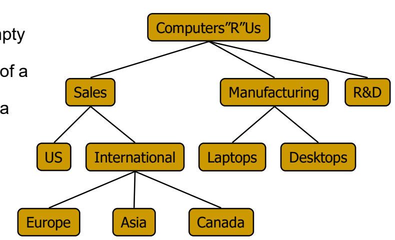


Computer Scientist’s View



What is a Tree

- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments



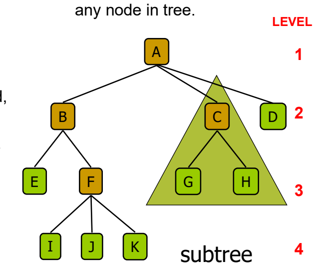
Definition of Tree

- A tree is a finite set of one or more nodes such that:

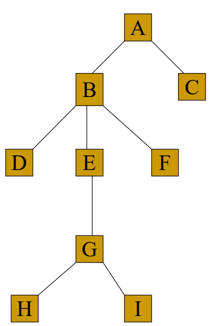
1. There is a specially designated node called the root.
2. The remaining nodes are partitioned into n>=0 disjoint sets $T_1, ..., T_n$, where each of these sets is a tree. $T_1, ..., T_n$ the subtrees of the root.

Tree Terminology

- **Node:** item of information plus branched to other items
- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree:** the number of subtrees of a node; degree of A = 3.
- The degree of a tree is the maximum of the degree of the nodes in the tree
- Nodes that have degree zero are called *leaf* or terminal node
- Others: nonterminals
- **Forest :** set of n>=0 disjoint trees
- **Subtree**: tree consisting of a node and its descendants
- **Level:**Root at level . If node is at level *l*, then its childeren are at at level *l*+1
- The **height** or **depth** of a tree is defined to be maximum level of any node in tree.



Tree Properties



Property

Number of nodes

Height

Root Node

Leaves

Interior nodes

Ancestors of H

Descendants of B

Siblings of E

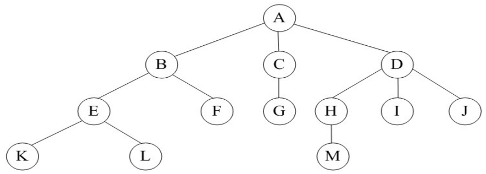
Right subtree of A

Degree of this tree

Tree ADT

- We use positions to abstract nodes
  - integer **size**()
  - boolean **isEmpty**()
  - integer **elements**()
  - boolean **isInternal**(p)
  - boolean **isExternal**(p)
  - boolean **isRoot**(p)
  - **swapElements**(p, q)
  - **replaceElement**(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Representation of Tree Node



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

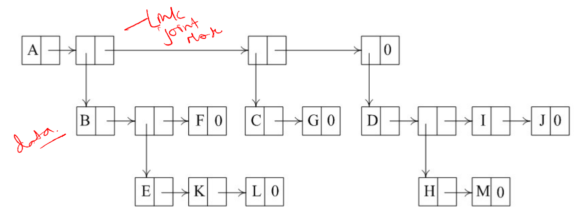
How many link fields are needed in such a representation?

Data	Link 1	Link 2	…	Link n

13

Representation of Tree Node

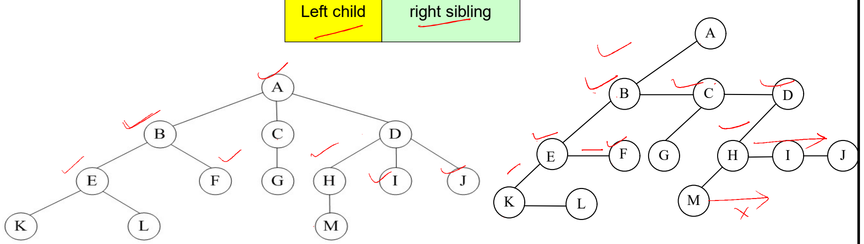
- (A (B (E (K, L), F), C (G), D (H (M), I, J)))
- GLL structure



14

Left Child-Right Sibling Representation

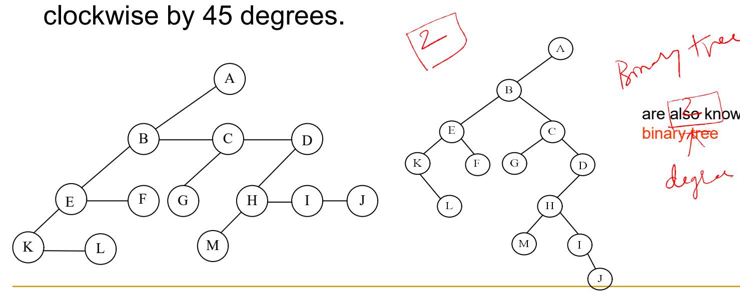
data	
Left child	right sibling



15

Representation as a Degree-Two Tree

- Rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees.

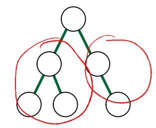


are ~~also~~ known as binary ~~tree~~

16

Binary tree

A binary tree is either empty, or it consists of a node called the root together with two binary trees called the left subtree and the right subtree of the root.



17

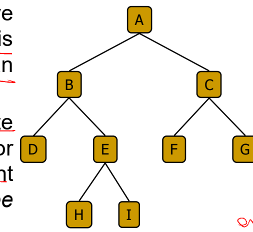
Binary Trees

0,1,2

- Tree in which any node can have atmost two branches, i.e. there is no node with degree greater than two.
- **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*.

recursion

- Any tree can be transformed into binary tree by left child-right sibling representation
- The left subtree and the right subtree are distinguished.



- Applications:
  - arithmetic expressions
  - decision processes
  - searching

order

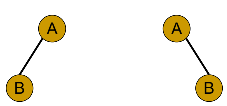
DT

BST (logn)

18

Differences Between A Tree and A Binary Tree

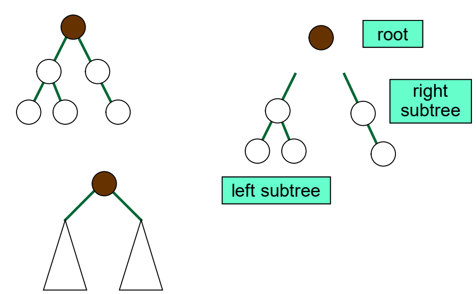
- A binary tree may be empty; a tree cannot be empty.
- No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
- The subtrees of a binary tree are ordered; those of a tree are not ordered.



- Are different when viewed as binary trees.
- Are the same when viewed as trees.

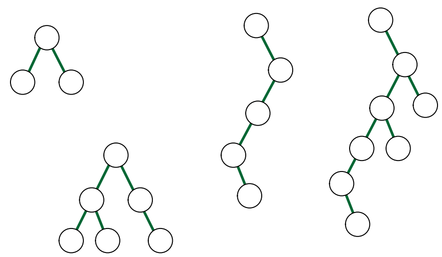
19

Root and subtrees



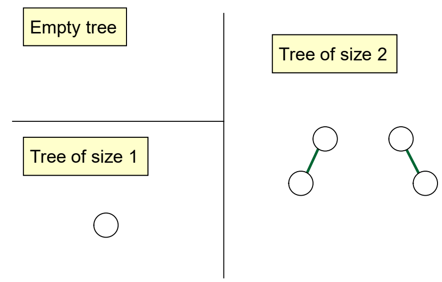
20

Some binary trees



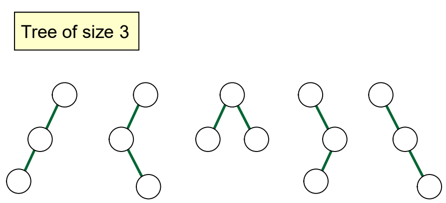
21

Small binary trees



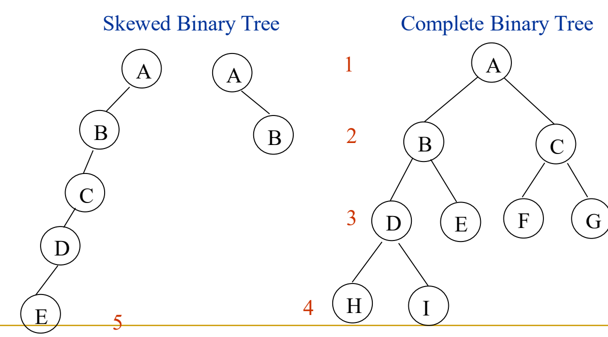
22

Small binary trees



23

Examples of the Binary Tree



24

Properties of Binary Trees

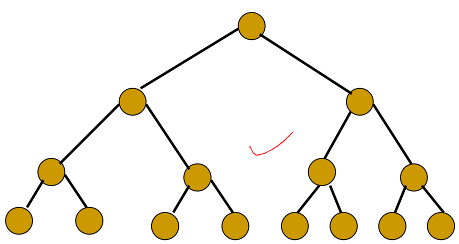
- Lemma 5.2 [Maximum number of nodes]
  - The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
  - The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

Lemma 5.3

- Relation between number of leaf nodes and degree-2 nodes
  - For any non-empty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.
- **Definition**: A **full binary tree** of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

Full Binary Tree

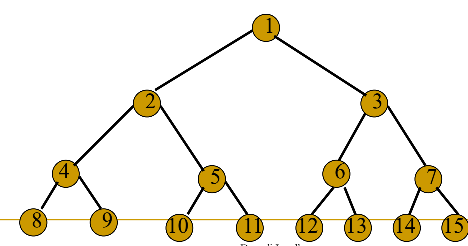
- A full binary tree of a given height k has $2^{k+1}-1$ nodes.



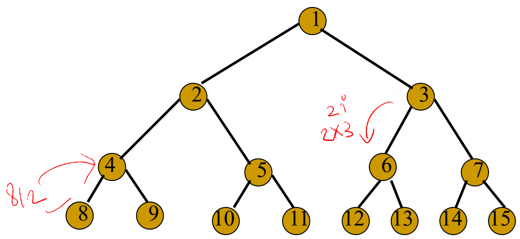
Height 3 full binary tree.

Labeling Nodes In A Full Binary Tree

- Label the nodes 1 through $2^{k+1} - 1$.
- Label by levels from top to bottom.
- Within a level, label from left to right.

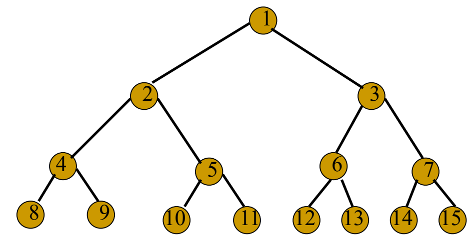


Node Number Properties



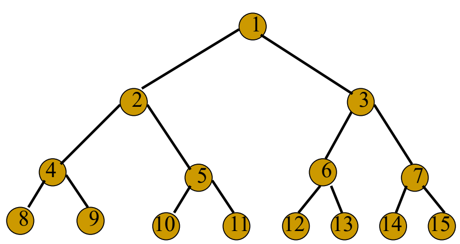
- Parent of node i is node i / 2, unless i = 1.
- Node 1 is the root and has no parent.

Node Number Properties



- Left child of node i is node 2i, unless 2i > n, where n is the number of nodes.
- If 2i > n, node i has no left child.

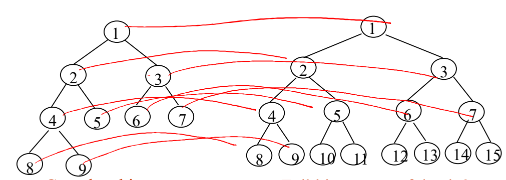
Node Number Properties



- Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.
- If 2i+1 > n, node i has no right child.

Complete Binary Trees

- A labeled binary tree containing the labels 1 to n with root 1, branches leading to nodes labeled 2 and 3, branches from these leading to 4, 5 and 6, 7, respectively, and so on.
- A binary tree with *n* nodes and level *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of level *k*.



Complete binary tree　　Full binary tree of depth 3

Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

Functions:

for all *bt, bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE)*

BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*

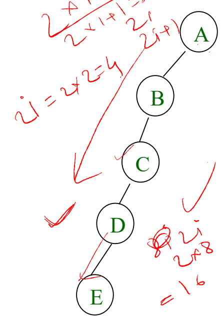
Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error else return the data in the root node of *bt*

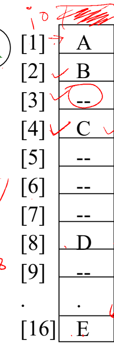
Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the right subtree of *bt*

Array Representation (1)

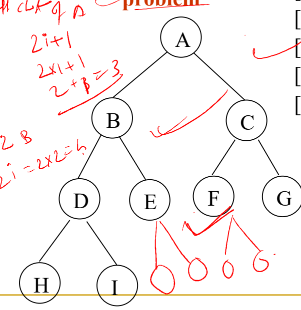
- Lemma 5.4: If a complete binary tree with *n* nodes is represented sequentially, then for any node with index *i*, $1 \le i \le n$, we have:
  - *parent*(*i*) is at if $i \neq 1$. If $i = 1$, *i* is at the root and has no parent.
  - *leftChild*(*i*) is at $2i$ if $2i \le n$. If $2i > n$, then *i* has no left child.
  - *rightChild*(*i*) is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then *i* has no right child.

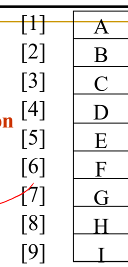


Array Representation (2)

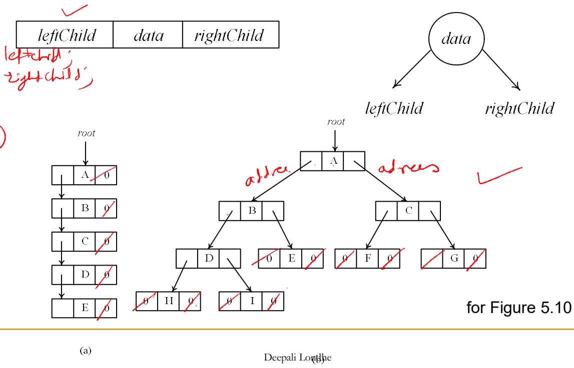


(1) waste space
(2) insertion/deletion problem





Linked Representation (2)



for Figure 5.10

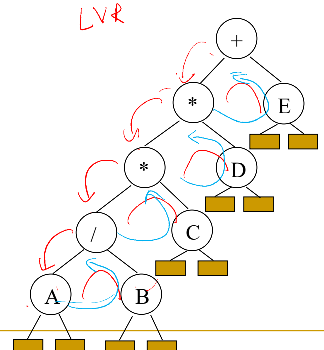
37

Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

38

Arithmetic Expression Using BT

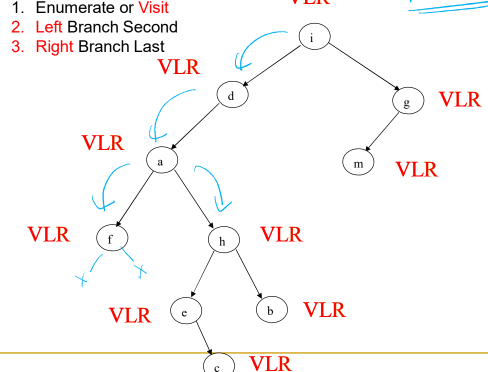


inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

39

Pre-Order Traversal of a Binary Tree

Nodes are recursively processed
1. Enumerate or Visit
2. Left Branch Second
3. Right Branch Last

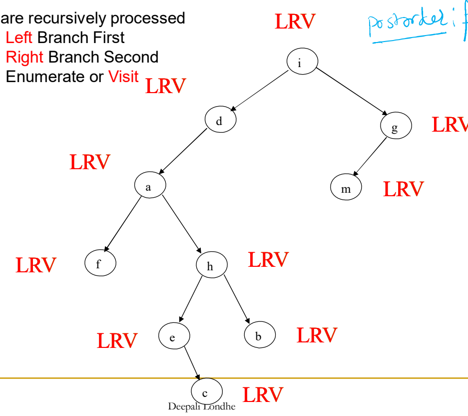


40

Post-Order Traversal of a Binary Tree

Nodes are recursively processed

1. Left Branch First
2. Right Branch Second
3. Enumerate or Visit



postorder: fcebhadmgi

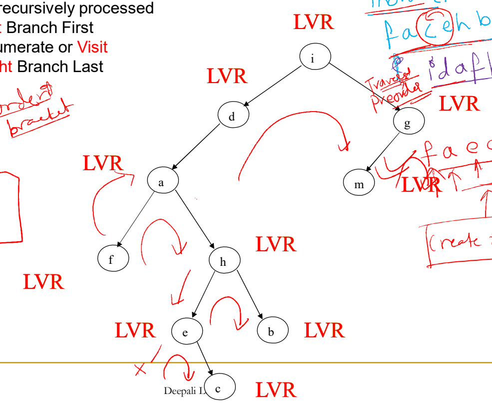
Deepali Londhe 41

41

In-Order Traversal of a Binary Tree

Nodes are recursively processed

1. Left Branch First
2. Enumerate or Visit
3. Right Branch Last



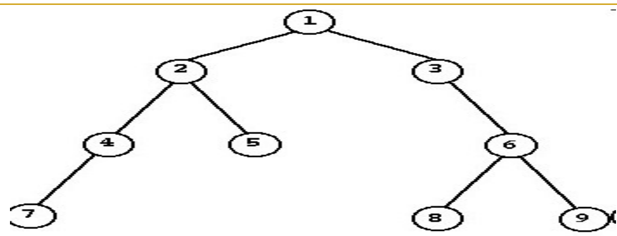
inorder:- facehbdimg

id

faechbdimg

Deepali Londhe 42

42



NLR:	1	2	4	7	5	3	6	8	9
LNR:	7	4	2	5	1	3	8	6	9
LRN:	7	4	5	2	8	9	6	3	1

preorder

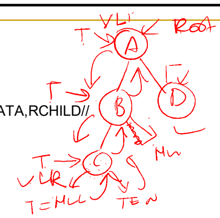
Deepali Londhe 43

43

Pseudo Codes

Preorder ABC, D

- Procedure PREORDER(T)
  //T is a binary search tree where each node has three fields LCHILD,DATA,RCHILD//
  if T≠ 0 then [Print(DATA(T))
  call PREORDER(LCHILD(T))
  call PREORDER(RCHILD(T))]
  end PREORDER
- Procedure INORDER(T)
  //T is a binary search tree where each node has three fields LCHILD,DATA,RCHILD//
  if T≠ 0 then [Call INORDER(LCHILD(T))
  Print(DATA(T))
  Call (INORDER(RCHILD(T))]
  end INORDER
- Procedure POSTORDER(T)
  //T is a binary search tree where each node has three fields LCHILD,DATA,RCHILD//
  if T≠ 0 then [call POSTORDER(LCHILD(T))
  call POSTORDER(RCHILD(T))
  Print(DATA(T))
  end POSTORDER



Deepali Londhe 44

44

Pseudo code for Recursive Preorder

- Procedure RPREORDER(T)
  //T is a binary tree where each node has three fields LCHILD,DATA,RCHILD//
1. [Process the root node]
   if T≠ NULL then
   Print(DATA(T))
   else
   return
2. [Process the left subtree]
   call RPREORDER(LCHILD(T))
3. [Process the right subtree]
   call RPREORDER(RCHILD(T))]
4. [finished]
   return

Pseudo code for Recursive Inorder

- Procedure RINORDER(T)
  //T is a binary search tree where each node has three fields LCHILD,DATA,RCHILD//
1. [Check for empty tree]
   if T= NULL
   then write "Empty tree"
   return
2. [Process the left subtree]
   If LCHILD(T) ≠ NULL
   call RINORDER(LCHILD(T))
3. [Process the node]
   Print(DATA(T))
4. [Process the right subtree]
   If RCHILD(T) ≠ NULL
   call RINORDER(RCHILD(T))]
5. [finished]
   return

Pseudo code for Recursive Postorder

Implicit

explicit

- Procedure RPOSTORDER(T)
  //T is a binary tree where each node has three fields LCHILD,DATA,RCHILD//
1. [Check for empty tree]
   if T= NULL
   then write "Empty tree"
   return
2. [Process the left subtree]
   If LCHILD(T) ≠ NULL
   call RPOSTORDER(LCHILD(T))
3. [Process the right subtree]
   If RCHILD(T) ≠ NULL
   call RPOSTORDER(RCHILD(T))
4. [Process the node]
   Print(DATA(T))
5. [finished]
   return

MR- Inorder

LVR

Pseudo code for Nonrecursive Inorder

Explicit

LVR

Preorder

VLR

LVR

- Procedure INORDER(T) ← root
  // S & TOP denote stack and associative top
  1. [Check & Initialize]
     if T= NULL
     then write "Empty tree"
     return
     TOP=0
  2. [Process each stacked branch address]
     Repeat step 3 while TOP ≠ -1 OR T ≠ NULL
  3. [Get address and branch left]
     Repeat while T ≠ NULL
     call PUSH(S,TOP,T)
     T ←LCHILD(T)
     If TOP ≠ -1 — stack empty
     T ←POP(S,TOP)
     Write DATA(T)
     T ←RCHILD(T)
  4. [Finished]
     return

DBEAC

Inorder:- DBEHGACF

Pseudo code for Nonrecursive Preorder

```
Procedure PREORDER(T)
// S & TOP denote stack and associative top
1. [Check & Initialize]
     if T= NULL
     then write "Empty tree"
        return
   TOP=0
2. [Process each stacked branch address]
        Repeat step 3  while TOP ≠ -1 OR T ≠ NULL
3. [Get address and branch left]
          Repeat while T ≠ NULL
             Write DATA(T)
             call PUSH(S,TOP,T)
             T←LCHILD(T)
        If  TOP ≠ -1
             T ←POP(S,TOP)
             T ←RCHILD(T)
4. [Finished]
     return
```

MR Postorder LRV

Inorder LVR

Preor VLR

VLR

ABDC

49

Pseudo code for Nonrecursive Postorder LRV

```
Procedure POSTORDER(T)
do
  while(t!=NULL && flg==0)
        top++;
        stack[top]=t;
        intstk[top]=1;
        t=t->left;
  if(top!=-1)
        t=stack[top];
        flg=intstk[top];
        top--;
        if(flg==1)
              top++;
              stack[top]=t;
              intstk[top]=2;
              t=t->right;
              flg=0;
        else
              printf("%c",t->data);
while(top!=-1);
```

LRV

CEBDA

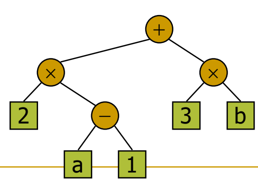
Print: ABC−*D+

Expression

50

Arithmetic Expression Tree

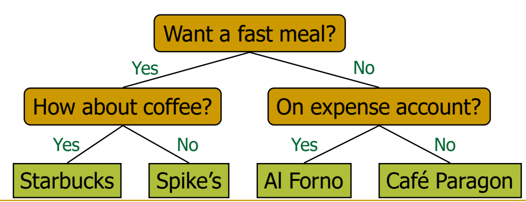
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression (2 × (a − 1) + (3 × b))



51

Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision

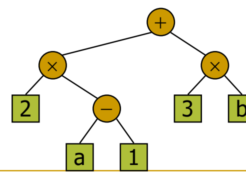


52

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

```
Algorithm inOrder (v)
    if isInternal (v){
        print("(")
        inOrder (leftChild (v))}
    print(v.element ())
    if isInternal (v){
        inOrder (rightChild (v))
        print (")")}
```



$((2 \times (a - 1)) + (3 \times b))$

Building a Binary Expression Tree

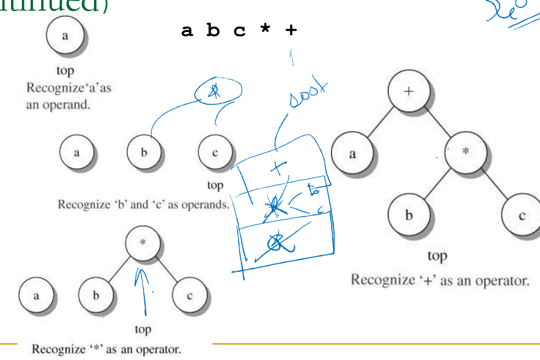
- Build an expression tree from a postfix expression using an iterative algorithm. An operand is a single character such as 'a' or 'b'.
- If the token is an operand, create a leaf node whose value is the operand and whose left and right subtrees are null. Push the node onto a stack of TNode references.

Building a Binary Expression Tree (continued)

- If the token is an operator, create a new node with the operator as its value. Pop the two child nodes from the stack and attach them to the new node. The first child popped from the stack becomes the right subtree of the new node and the second child popped from the stack becomes the left subtree.

Building a Binary Expression Tree (continued)

root = +



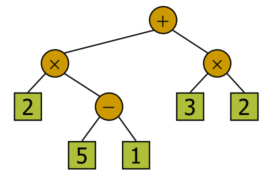
Student Question

- Show the stack and the partial expression trees, step by step, for the expression
  a b c - d e / + *

57

Evaluate Arithmetic Expressions

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees



Algorithm *evalExpr*(*v*)
  if *isExternal* (*v*)
    return *v.element* ()
  else
    x ← *evalExpr*(*leftChild* (*v*))
    y ← *evalExpr*(*rightChild* (*v*))
    ◊ ← operator stored at *v*
    return *x* ◊ *y*

58