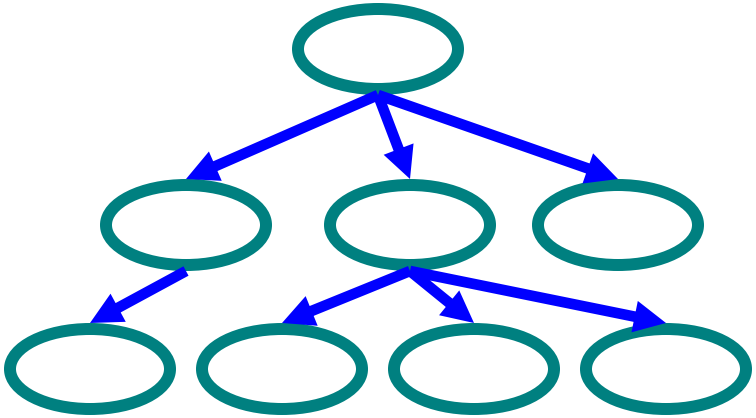
Tree Data Structures

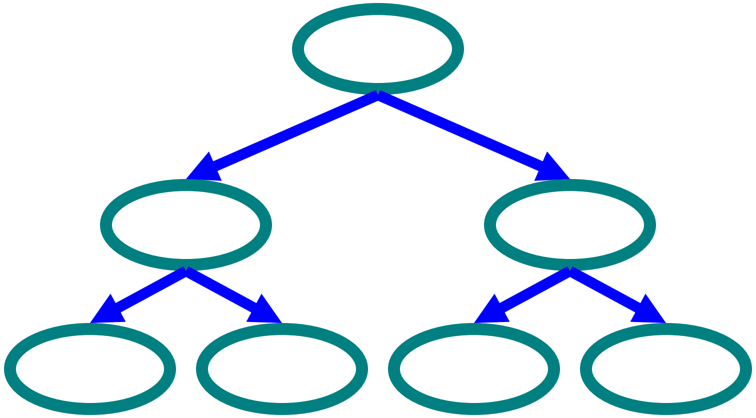
Borisaniya Bhavesh

Trees Data Structures

- Tree
  - Nodes
  - Each node can have 0 or more children
  - A node can have at most one parent
- Binary tree
  - Tree with 0–2 children per node



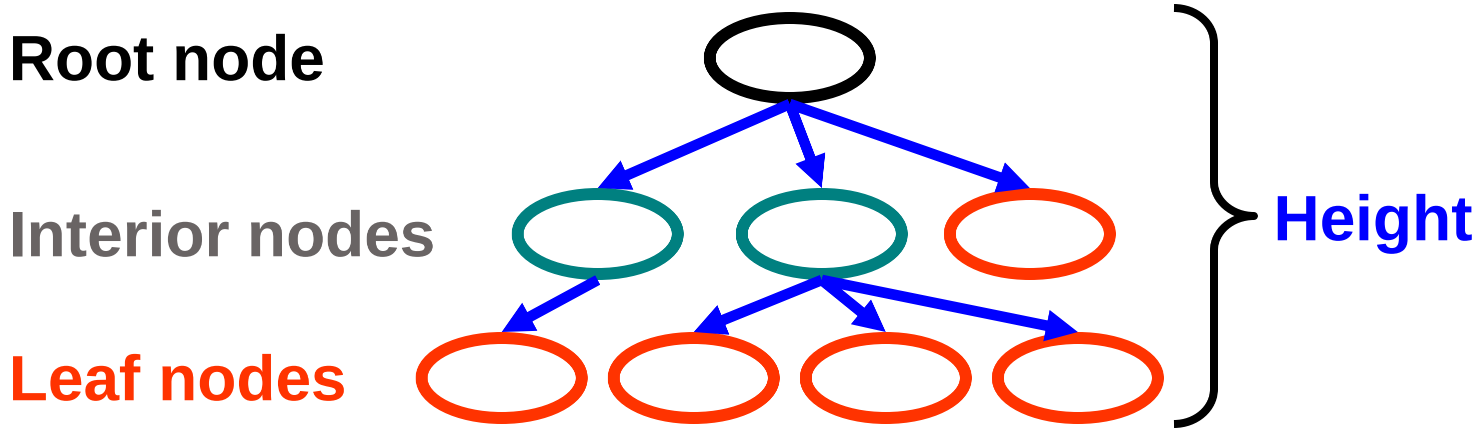
Tree



Binary Tree

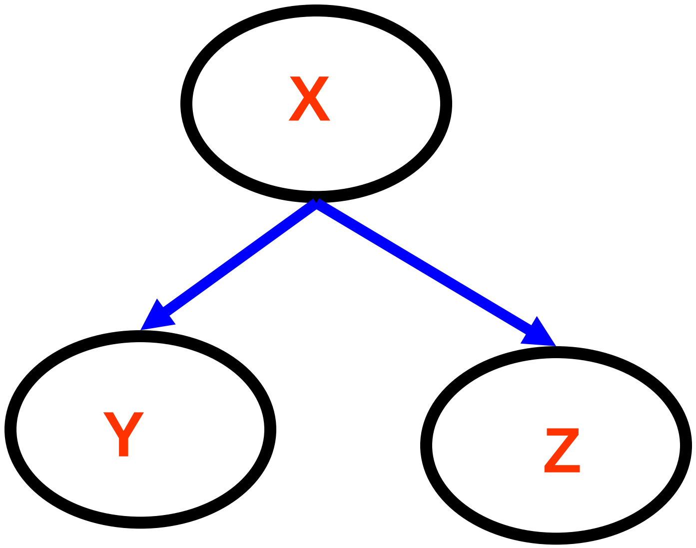
Trees

- Terminology
  - Root ⇒ no parent
  - Leaf ⇒ no child
  - Interior ⇒ non-leaf
  - Height ⇒ distance from root to leaf



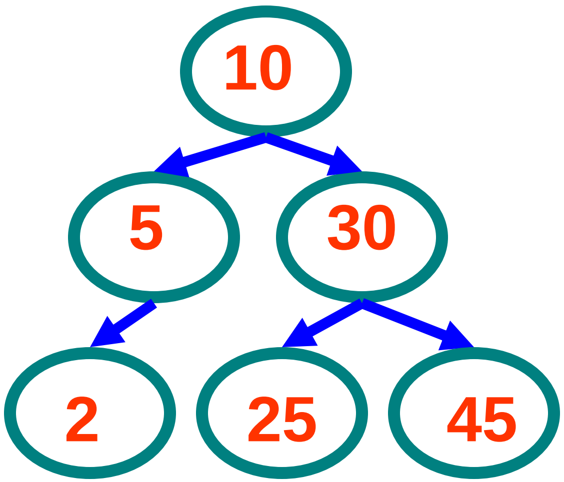
Binary Search Trees

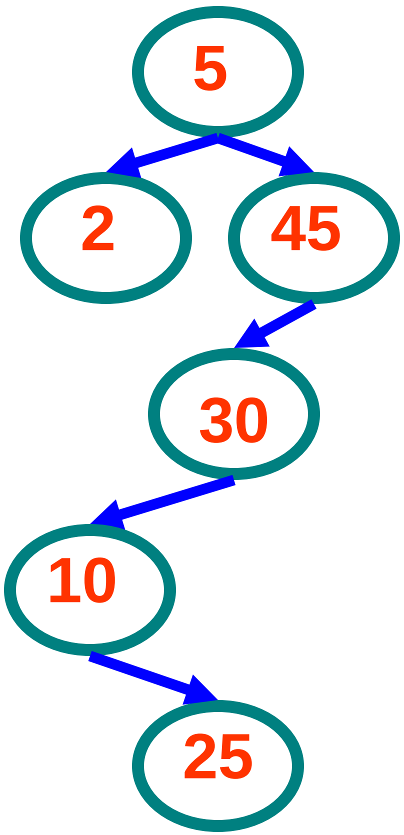
- Key property
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
  - Example
    - X > Y
    - X < Z



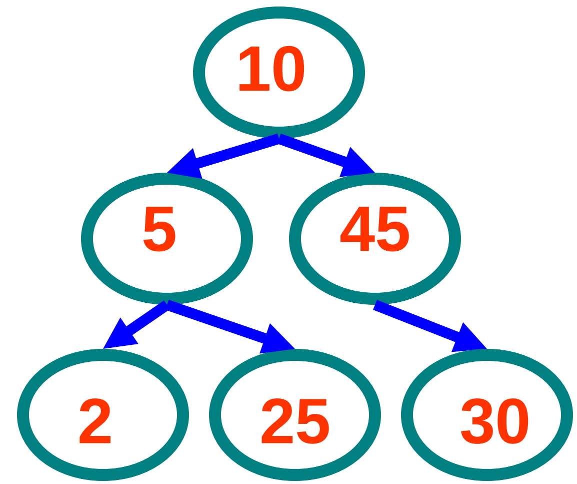
Binary Search Trees

- Examples





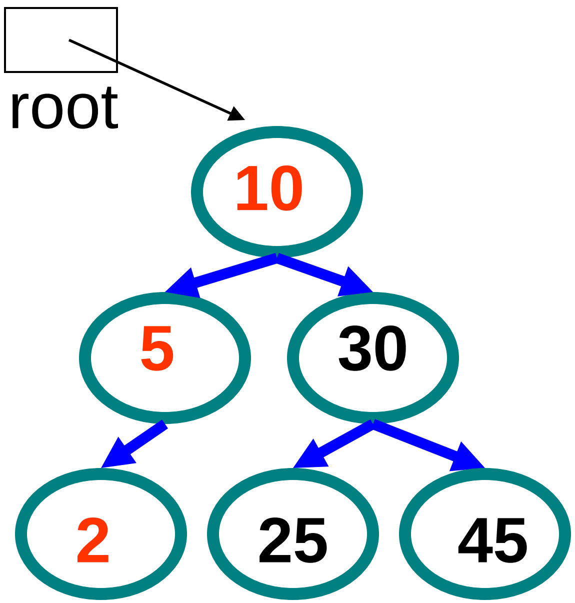
Binary search trees



Not a binary search tree

Example Binary Searches

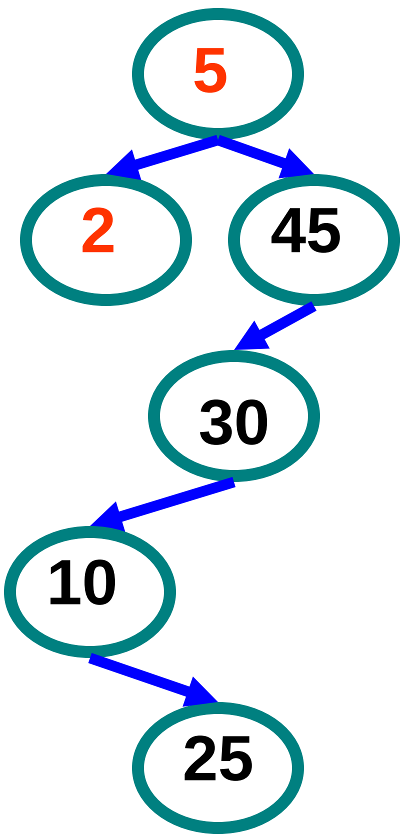
- Find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

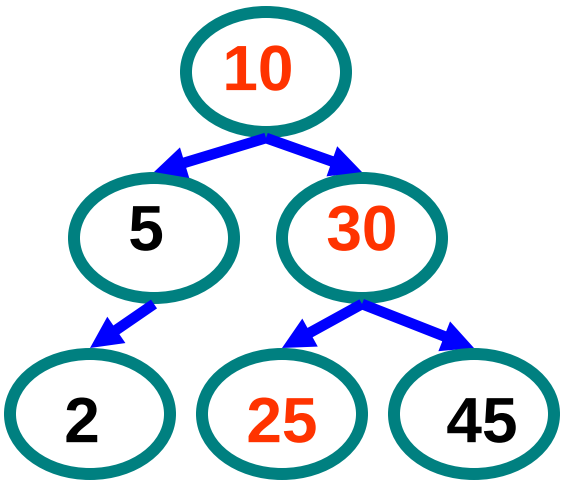


5 > 2, left

2 = 2, found

Example Binary Searches

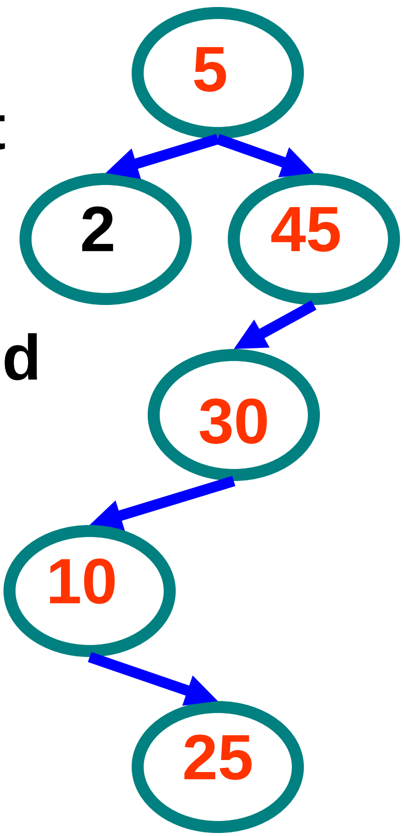
- Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

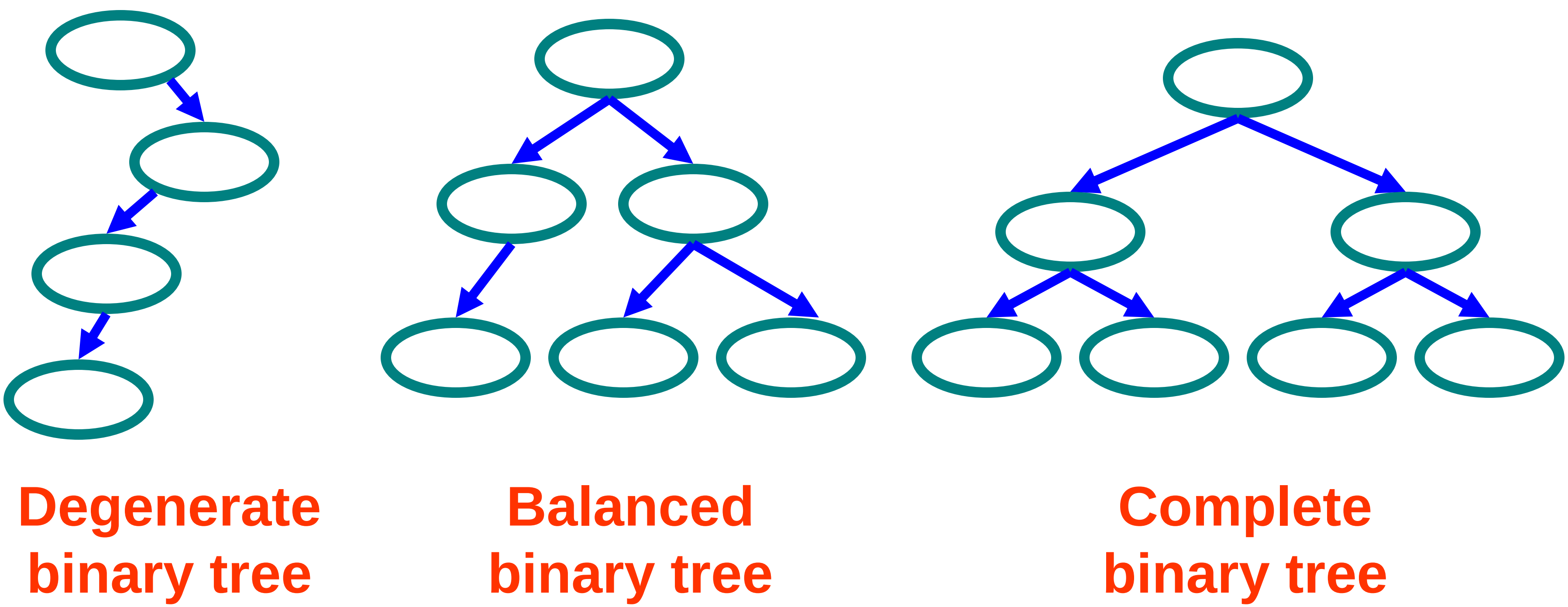
30 > 25, left

10 < 25, right

25 = 25, found

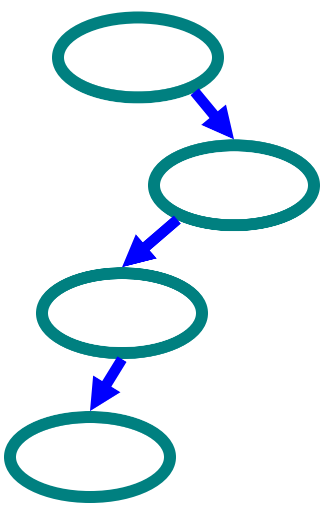
Types of Binary Trees

- Degenerate – only one child
- Complete – always two children
- Balanced – "mostly" two children
  - more formal definitions exist, above are intuitive ideas

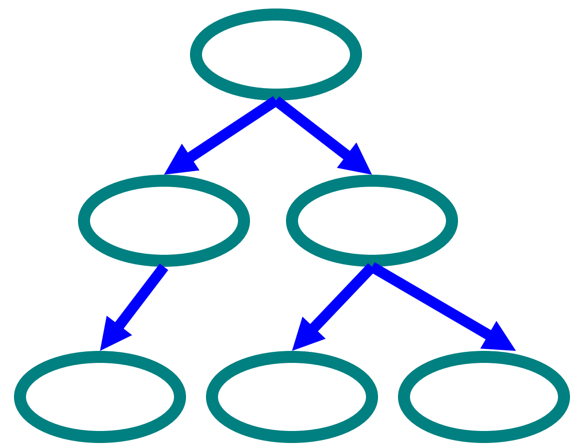


Binary Trees Properties

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linked list
- Balanced
  - Height = O(log(n)) for n nodes
  - Useful for searches



Degenerate binary tree



Balanced binary tree

Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - Degenerate tree
    - O(n) time
    - Like searching linked list / unsorted array

Operations on Binary Trees

- We will see operations on binary trees such as,
  - Traversal of trees
  - Insertion
  - Deletion
  - Searching
  - Copying

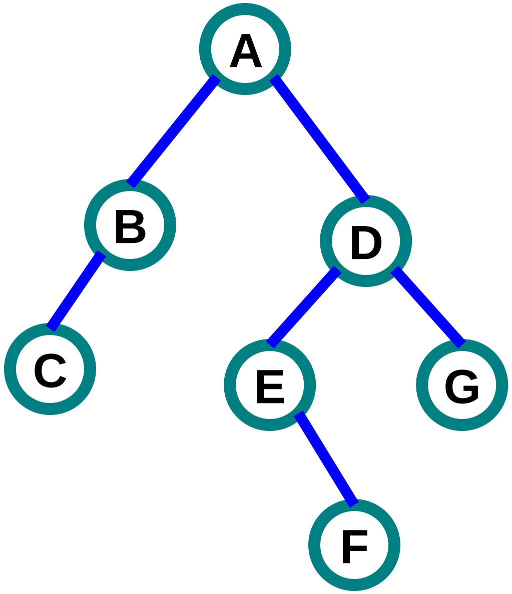
Traversal of binary trees

- It is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- The meaning of "processed"?
- For example, a tree could represent an arithmetic expression.
- There are three ways of traversing a binary tree:
  - Preorder
  - Inorder
  - Postorder

Traversal of binary trees

- **Preorder traversal of binary tree**

1. Process the root node
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

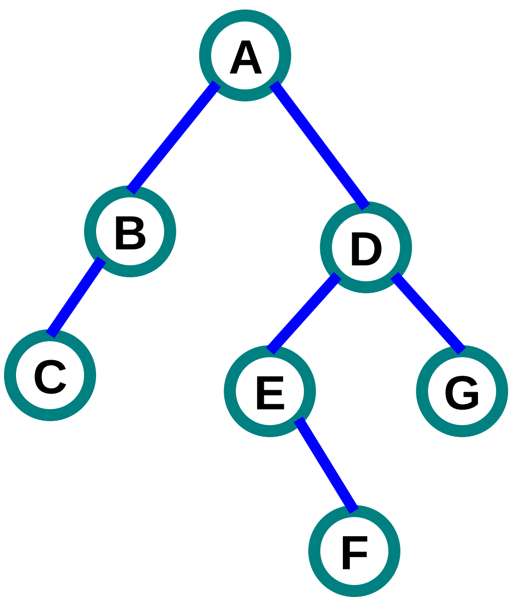


A B C D E F G

Traversal of binary trees

- **Inorder traversal of binary tree**

1. Traverse the left subtree in inorder
2. Process the root node
3. Traverse the right subtree in inorder

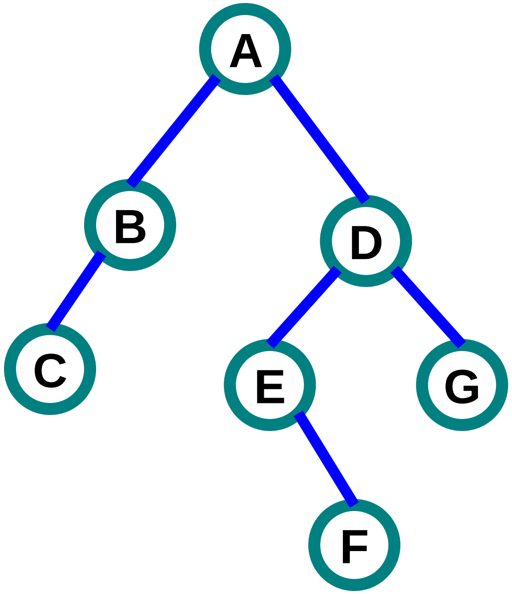


C B A E F D G

Traversal of binary trees

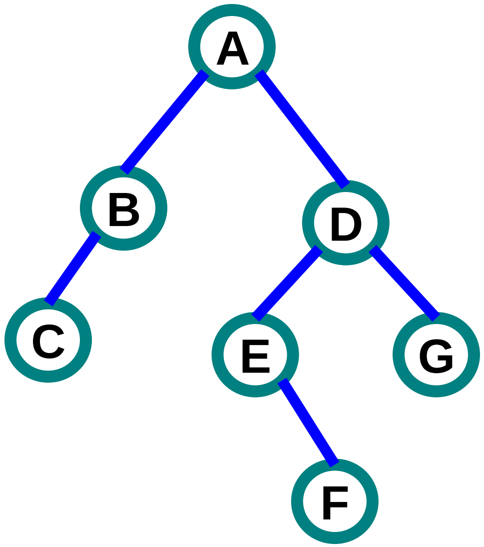
- **Postorder traversal of binary tree**

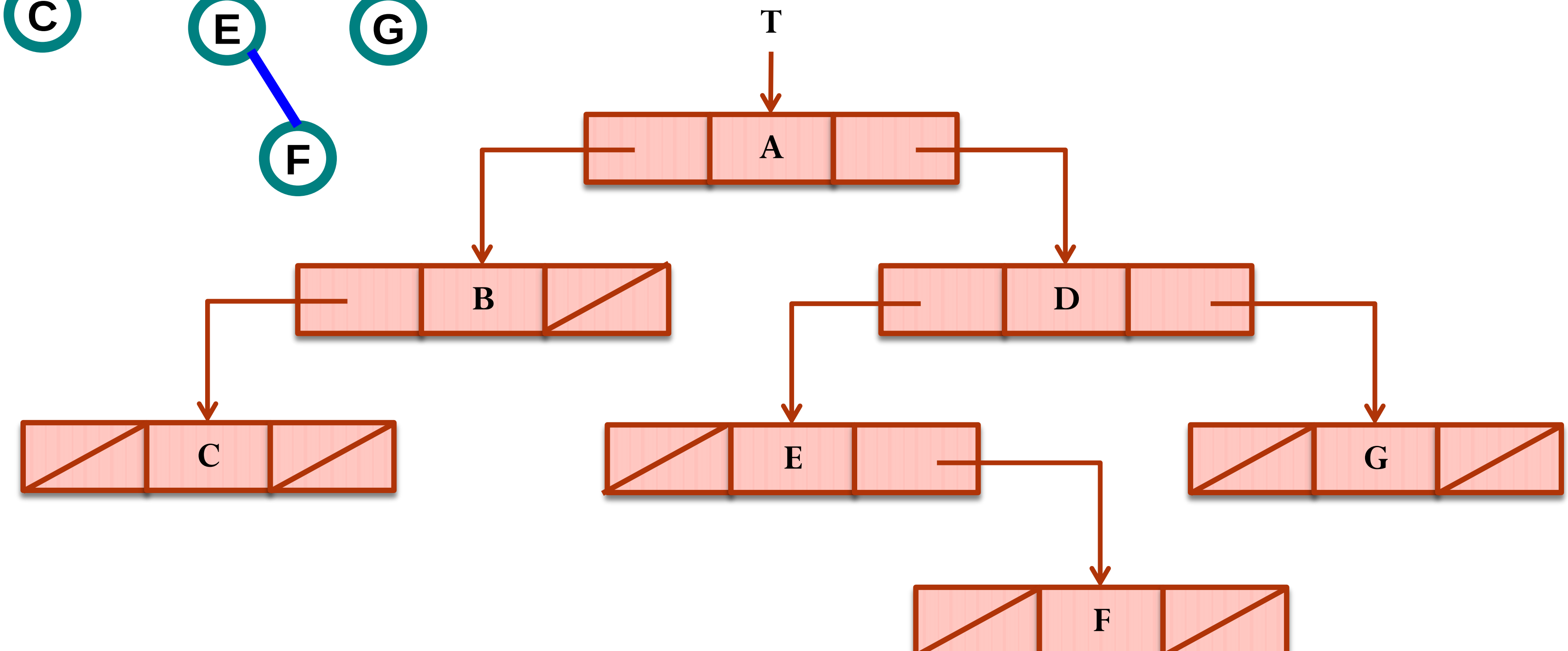
1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Process the root node



C B F E G D A

Traversal of binary trees





Preorder traversal of binary tree

Procedure: PREORDER(T)

- T – pointer gives address of root node in given binary tree
- S – auxiliary stack
- TOP – top index of stack
- LPTR, RPTR – address to left and right child of the current node

Preorder traversal of binary tree

1. [Initialize]

   If T= NULL

   then Write ('EMPTY TREE')

   Return

   else TOP ← 0

   Call PUSH (S, TOP, T)

2. [Process each stacked branch address]

   Repeat step 3 while TOP > 0

3. [Get stored address and branch left]

   P ← POP (S, TOP)

   Repeat while P!= NULL

   Write (DATA(P))

   If RPTR(P) != NULL

   then Call PUSH (S, TOP, RPTR (P)) // store address of nonempty right subtree

   P ← LPTR(P) // branch left

4. [Finished]

   Return

Recursive preorder traversal of binary tree

Procedure: RPREORDER(T)

1. [Process the root node]

   If T!= NULL

   then Write (DATA(T))

   else Write ('EMPTY DATA')

   Return

2. [Process the left subtree]

   If LPTR(T) != NULL

   then Call RPREORDER(LPTR(T))

3. [Process the right subtree]

   if RPTR(T) != NULL

   then Call RPREORDER(RPTR(T))

4. [Finished]

   Return

Postorder traversal of binary tree

Procedure: POSTORDER(T)

- T – pointer gives address of root node in given binary tree
- S – auxiliary stack
- TOP – top index of stack

Postorder traversal of binary tree

1. [Initialize]

   If T= NULL

   then Write ('EMPTY TREE')

   Return

   else TOP ← 0

   P← T

2. [Traverse in postorder]

   Repeat thru step 5 while true

3. [Descend left]

   Repeat while P!= NULL

   Call PUSH (S, TOP, P)
   P ← LPTR(P)

4. [Process a node whose left and right subtrees have been traversed]

   Repeat while S[TOP] < 0

   P ← POP(S, TOP)

   Write (DATA(P))

   If TOP=0 //has all nodes be processed

   then Return

5. [Branch right and then mark node from which we branched]

   P← RPTR(S[TOP])

   S[TOP] ← -S[TOP]

Recursive postorder traversal of binary tree

Procedure: RPOSTORDER(T)

1. [Process the root node]
   If T= NULL
   then Write ('EMPTY DATA')
   Return
2. [Process the left subtree]
   If LPTR(T) != NULL
   then Call RPOSTORDER(LPTR(T))
3. [Process the right subtree]
   if RPTR(T) != NULL
   then Call RPOSTORDER(RPTR(T))
4. [Process the root node]
   Write (DATA(T))
5. [Finished]
   Return

Recursive Inorder traversal of binary tree

Procedure: RPOSTORDER(T)

1. [Process the root node]
   If T= NULL
   then Write ('EMPTY DATA')
   Return
2. [Process the left subtree]
   If LPTR(T) != NULL
   then Call RPOSTORDER(LPTR(T))
3. [Process the root node]
   Write (DATA(T))
4. [Process the right subtree]
   if RPTR(T) != NULL
   then Call RPOSTORDER(RPTR(T))
5. [Finished]
   Return

Copy Tree

Function: COPY(T)

- NEW – new node to be inserted in new tree
- T – pointer pointing to the root node in original tree
- DATA – data part of node
- LPTR, RPTR – pointer pointing to the left and right subtree

Copy Tree

1. [Null pointer ?]
   if T = NULL
   then Return(NULL)
2. [Create a new node]
   NEW ← NODE
3. [Copy information field]
   DATA(NEW) ← DATA(T)
4. [Set the structural links]
   LPTR(NEW) ← COPY(LPTR(T))
   RPTR(NEW) ← COPY(RPTR(T))
5. [Return address of new node]
   Return (NEW)

Binary Search Tree Construction

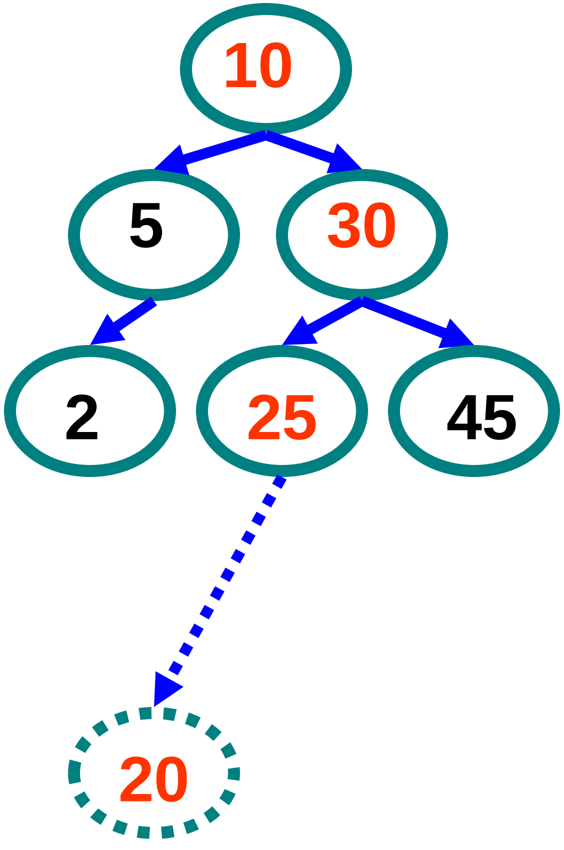
- How to build & maintain binary trees?
  - Insertion
  - Deletion
- Maintain key property (invariant)
  - Smaller values in left subtree
  - Larger values in right subtree

Binary Search Tree – Insertion

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y
- Observations
  - O(log(n)) operation for balanced tree
  - Insertions may unbalance tree

Example Insertion

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

Binary Search Tree – Insertion

Procedure: BSTINS(T, Z)

- T – pointer pointing to root node of the tree
- Z – value or data to be inserted in the tree
- CUR – pointer pointing to the current element in tree
- PRED – pointer pointing to the parent element of CUR
- LPTR, RPTR – pointer pointing to left and right subtree
- NEW – new element to be inserted
- DATA – data part of element in tree

Binary Search Tree – Insertion

1. [Create a node]

   NEW $\leftarrow$ node

   INFO(NEW) $\leftarrow$ Z

   LPTR(NEW) $\leftarrow$ RPTR(NEW) $\leftarrow$ NULL

2. [Check if tree is null]

   if T= NULL

   then T $\leftarrow$ NEW

3. [Initialize search]

   PRED $\leftarrow$ NULL

   CUR $\leftarrow$ T

Binary Search Tree – Insertion

4. [Search the position of new element in tree]

   while CUR != NULL

   PRED ← CUR

   if Z = DATA(CUR)

   then Write ('SIMILAR DATA')

   Return

   else if Z< DATA(CUR)

   then CUR ← LPTR(CUR)

   else CUR ← RPTR(CUR)

5. [Insert the new element]

   if Z < DATA(PRED)

   then LPTR(PRED) ← NEW

   else RPTR(PRED) ← NEW

6. [Finish]

   Return

Binary Search Tree – Deletion

- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else   // must delete internal node

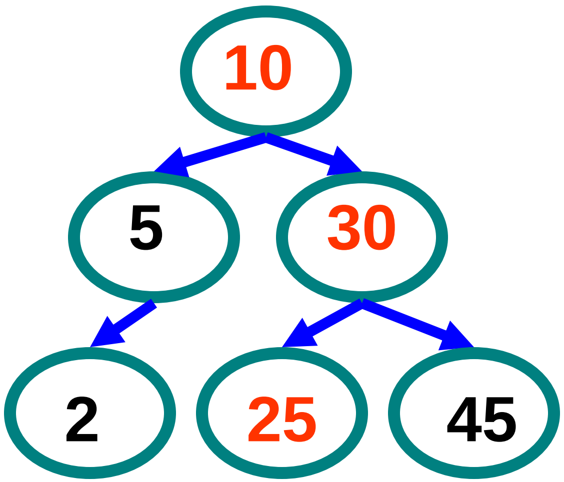
     a) Replace with largest value Y on left subtree

     OR smallest value Z on right subtree

     b) Delete replacement value (Y or Z) from subtree
- Observation
  - O(log(n)) operation for balanced tree
  - Deletions may unbalance tree

Example Deletion (Leaf)

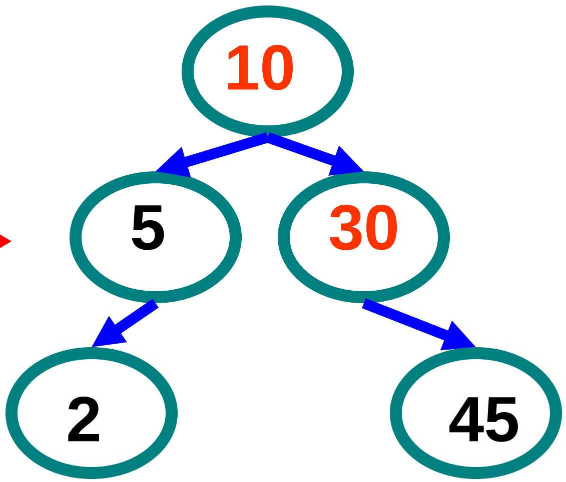
- Delete (25)



10 < 25, right

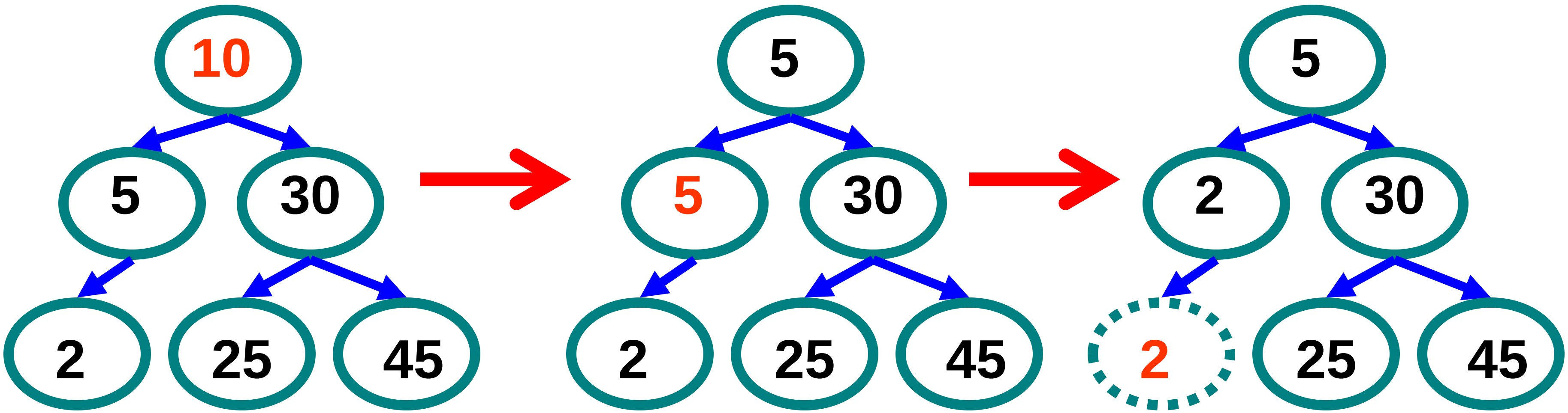
30 > 25, left

25 = 25, delete



Example Deletion (Internal Node)

- Delete (10)



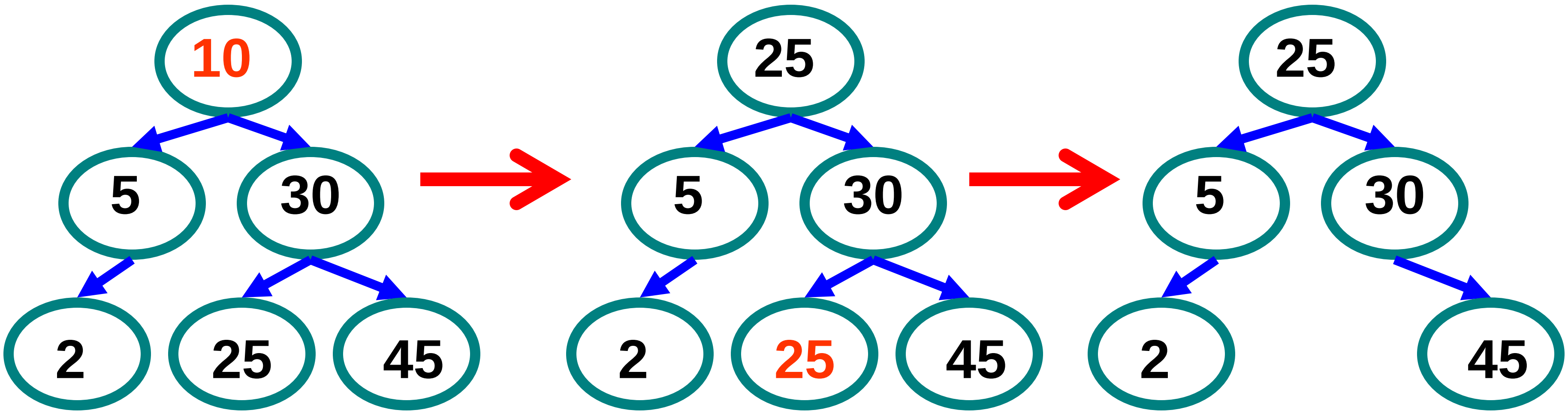
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

Binary Search Tree – Deletion

Procedure: TREE_DELETE(T, X)

- X – value of the node marked for deletion
- PARENT – is a pointer variable denotes the address of the parent node marked for deletion
- CUR – denotes the address of the node to be deleted
- PRED, SUC – are the pointer variables used to find the inorder successor of CUR
- Q – contains the address of the node to which either the left or right link of the parent of X must be assigned in order to complete the deletion
- D – contains the direction from the parent node to the node marked for deletion

Binary Search Tree – Deletion

1. [Check if tree is empty]

   if T = NULL

   then Write ('TREE IS EMPTY')

2. [Check if root node to delete]

   if DATA(T) = X

   then CUR = T

   go to step 5

3. [Initialize search]

   CUR = T

Binary Search Tree – Deletion

4. [Search for the node marked for deletion]

```
FOUND ← false
Repeat while not FOUND and CUR != NULL
    if DATA(CUR) = X
    then FOUND ← true
    else if  X < DATA(CUR)
      then (branch left)
            PARENT ← CUR
            CUR ← LPTR (CUR)
            D ← 'L'
      else  (branch right)
            PARENT ← CUR
            CUR ← RPTR(CUR)
            D ← 'R'
If FOUND = false
then  Write('NODE NOT FOUND')
        Return
```

Binary Search Tree – Deletion

5. [Perform the indicated deletion and restructure the tree]

    if LPTR(CUR) = NULL

    then (empty left subtree)

        Q ← RPTR(CUR)

    else if RPTR(CUR) = NULL

        then (empty right subtree)

            Q ← LPTR(CUR)

        else (check right child for success)

            SUC ← RPTR (CUR)

            if LPTR(SUC) = NULL

            then LPTR(SUC) ← LPTR(CUR)

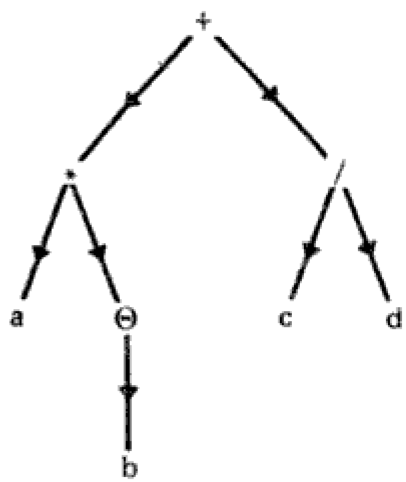
                Q ← SUC

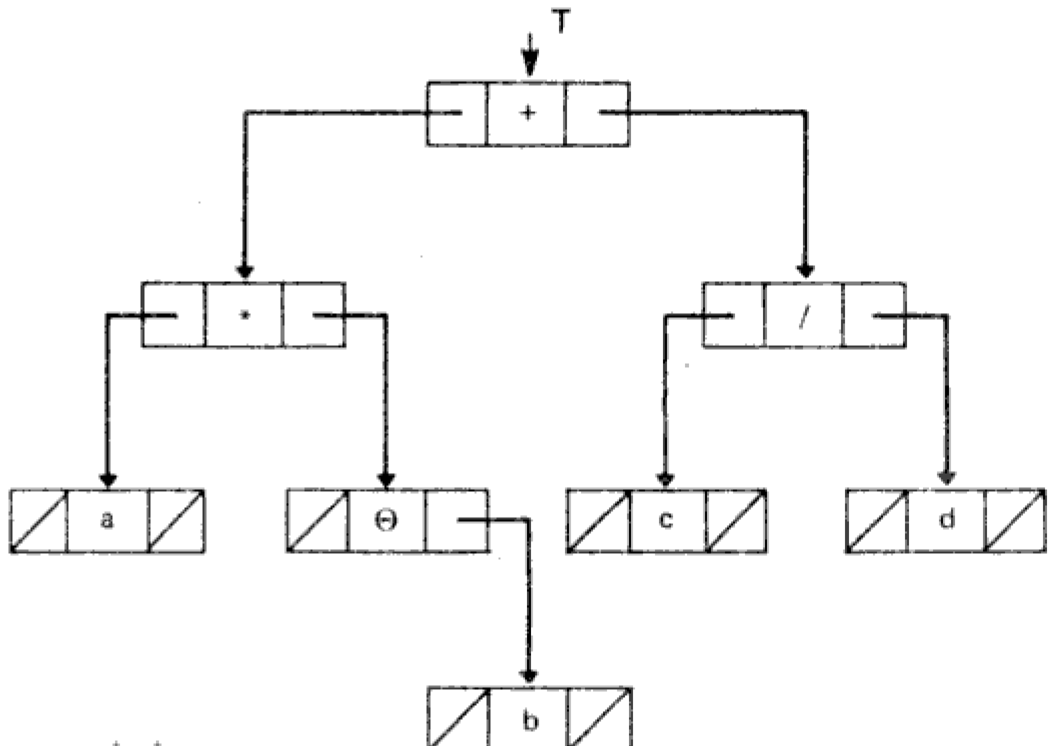
Binary Search Tree – Deletion

```
else (search for successor of CUR)
    PRED ← RPTR(CUR)
    SUC ← LPTR(PRED)
    Repeat while LPTR(SUC) != NULL
            PRED ← SUC
            SUC ← LPTR(PRED)
        (connect successor)
        LPTR(PRED) ← RPTR(SUC)
        LPTR(SUC) ← LPTR(CUR)
        RPTR(SUC) ← RPTR(CUR)
        Q ← SUC
(Connect parent of X to its replacement)
 if D = 'L'
then  LPTR(PARENT) ← Q
else   RPTR(PARENT) ← Q
Return
```

Applications of Tree

- The Manipulation of Arithmetic Expression





$a * \theta b + c / d$

Applications of Tree

- Symbol Table Construction
- Syntax Analysis

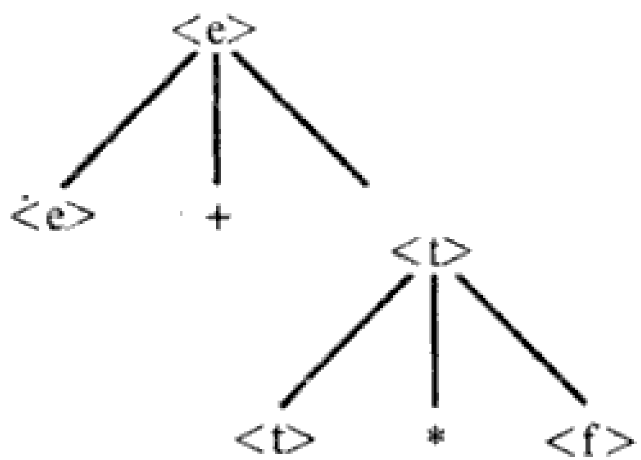


FIGURE 5-2.12 Syntax tree for <expr> + <term> * <factor> in G_1.