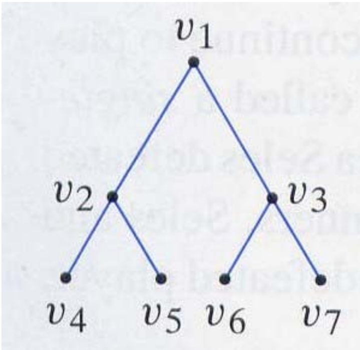
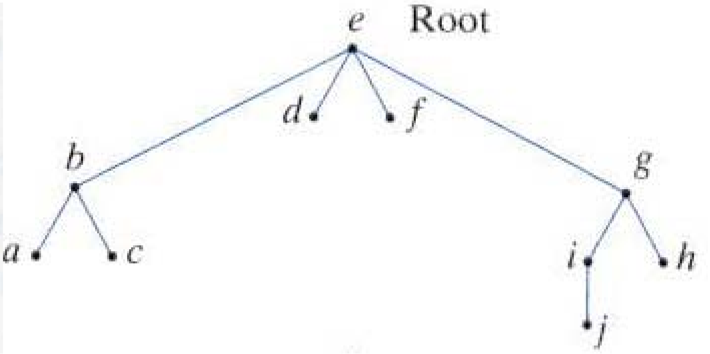
Chapter 4 Trees

4-1 Trees and Spanning Trees

Trees, *T*: A simple, cycle-free, loop-free graph satisfies: If *v* and *w* are vertices in *T*, there is a unique simple path from *v* to *w*.

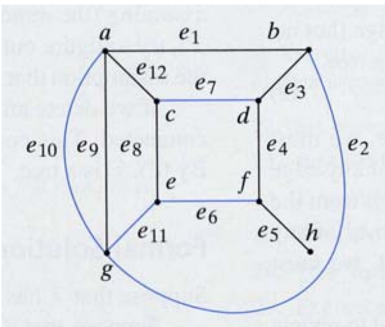
Eg. Trees.



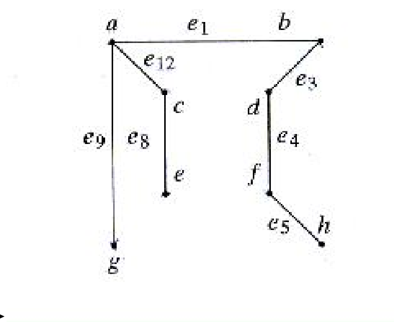


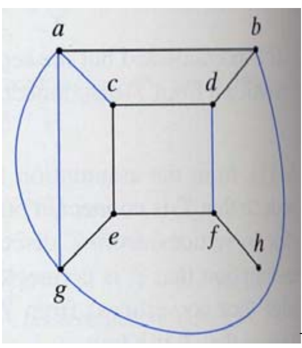
Spanning trees: A subgraph of *G* is a tree and it contains all of the vertices of *G*.

Eg. Two examples of graphs and their respective corresponding spanning trees.

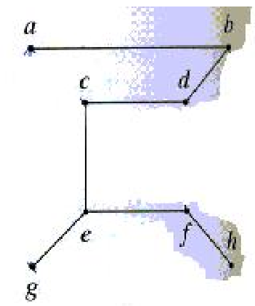


→





→



Theorem **A graph *G* has a spanning tree if and only if *G* is connected.**

Breadth-first search for spanning trees algorithm

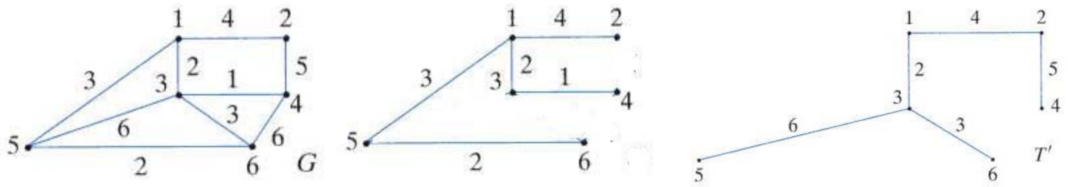
```
   Input:  A connected graph G with vertices ordered
           v1, v2, ... vn
  Output:  A spanning tree T

bfs(V,E) {
  // V = vertices ordered v1,...,vn; E = edges
  // V' = vertices of spanning tree T;
  // E' = edges of spanning tree T
  // v1 is the root of the spanning tree
  // S is an ordered list
  S = (v1)
  V' = {v1}
  E' = ∅
  while (true) {
    for each x ∈ S, in order,
      for each y ∈ V − V', in order,
        if ((x,y) is an edge)
          add edge (x,y) to E' and y to V'
    if (no edges were added)
      return T
    S = children of S ordered consistently with the
        original vertex ordering
  }
}
```

Depth-first search for spanning trees algorithm

```
   Input:  A connected graph G with vertices ordered
           v1, v2, ... vn
  Output:  A spanning tree T

dfs(V,E) {
  // V' = vertices of spanning tree T;
  // E' = edges of spanning tree T
  // v1 is the root of the spanning tree
  V' = {v1}
  E' = ∅
  w = v1
  while (true) {
    while (there is an edge (w,v) that when added to T
           does not create a cycle in T) {
      choose the edge (w,vk) with minimum k that when
         added to T does not create a cycle in T
      add (w,vk) to E'
      add vk to V'
      w = vk
    }
    if (w == v1)
      return T
    w = parent of w in T // backtrack
  }
}
```

Minimal Spanning trees: A spanning tree with minimum weight.

Eg. For the following leftmost graph *G*, *T* and *T'* are both the spanning trees of *G*, the weight of *T* is 12 and that of *T'* is 20, we can see that *T* is a minimal spanning tree.



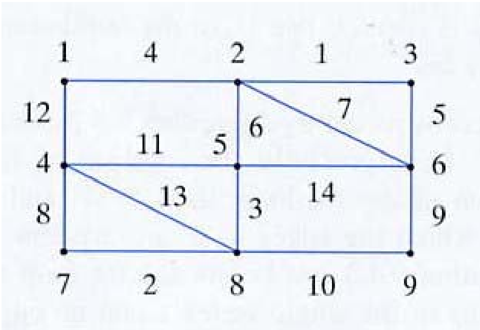
Prim's algorithm

Prim's Algorithm

Input: A connected, weighted graph G with vertices $1, \ldots, n$ and start vertex s. If (i, j) is an edge, $w(i, j)$ is equal to the weight of (i, j); if (i, j) is not an edge, $w(i, j)$ is equal to ∞ (a value greater than any actual weight).

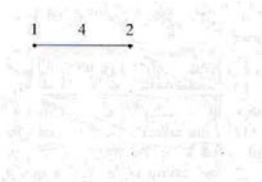
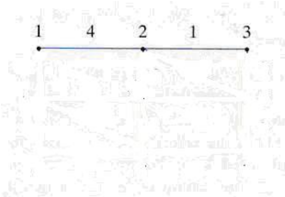
Output: The set of edges E is a minimal spanning tree (mst)

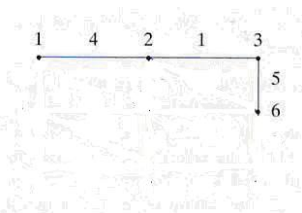
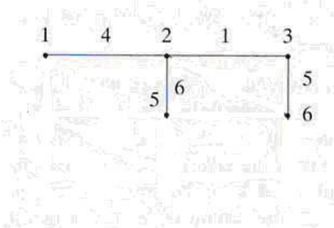
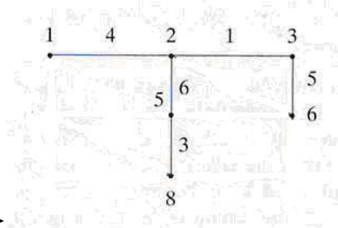
```
prim(w, n, s) {
   // v(i) = 1 if vertex i has been added to mst
   // v(i) = 0 if vertex i has not been added to mst
   for i = 1 to n
      v(i) = 0
   // add start vertex to mst
   v(s) = 1
   // begin with an empty edge set
   E = ∅

   // put n − 1 edges in the minimal spanning tree
   for i = 1 to n − 1 {
      // add edge of minimum weight with one
      // vertex in mst and one vertex not in mst
      min = ∞
      for j = 1 to n
         if (v(j) == 1) // if j is a vertex in mst
            for k = 1 to n
               if (v(k) == 0 ∧ w(j, k) < min) {
                  add_vertex = k
                  e = (j, k)
                  min = w(j, k)
               }
      // put vertex and edge in mst
      v(add_vertex) = 1
      E = E ∪ {e}
   }
   return E
}
```

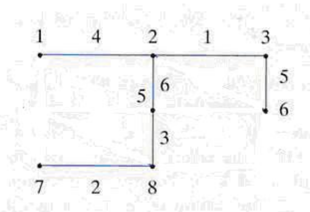
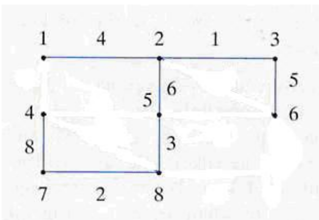
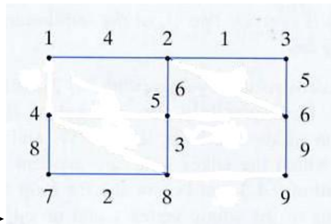


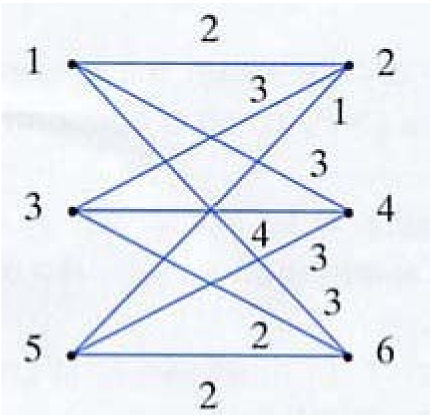
Eg. Find the minimal spanning tree of the left graph.

(Sol.)

 → 

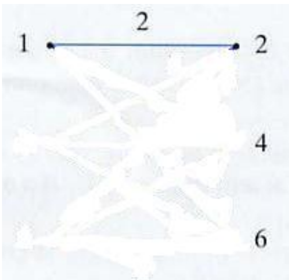
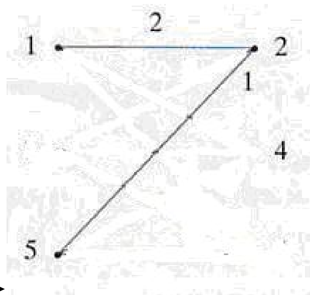
→  →  → 

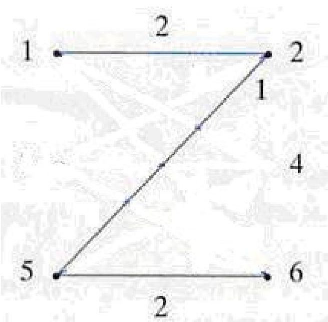
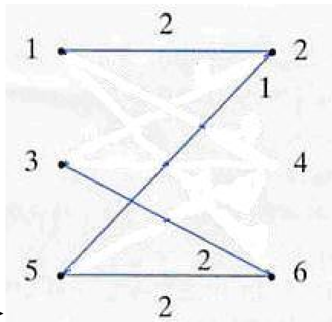
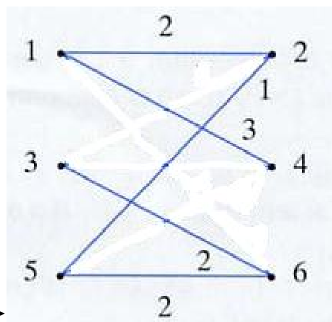
→  →  → 

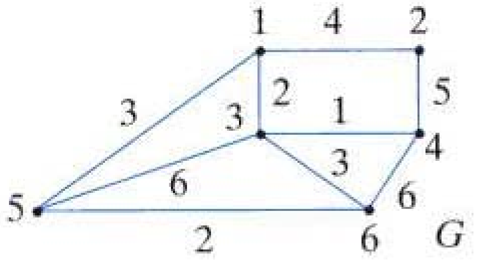


Eg. Find the minimal spanning tree of the left graph.

(Sol.)

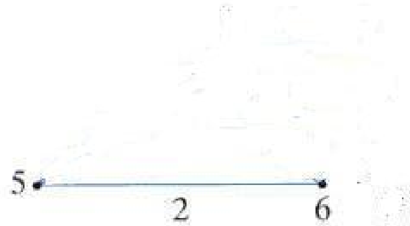
 → 

→  →  → 

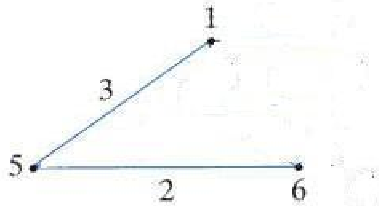


Eg. Find the minimal spanning tree of the left graph by Prim's algorithm. Select vertex 5 as the root.

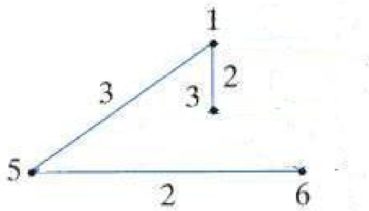
(Sol.)



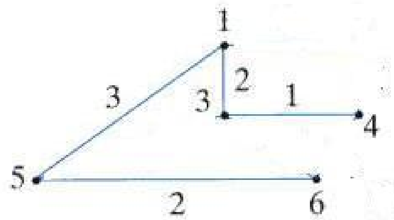
→



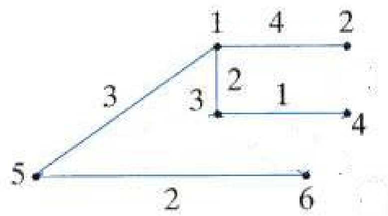
→

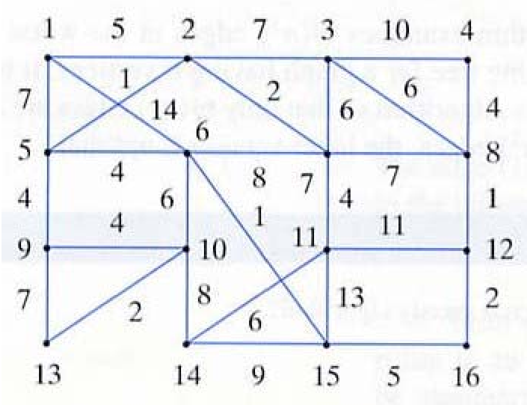


→



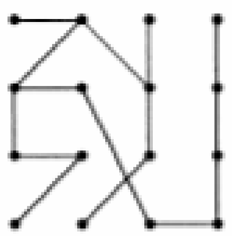
→





Eg. Find the minimal spanning tree of the left graph.

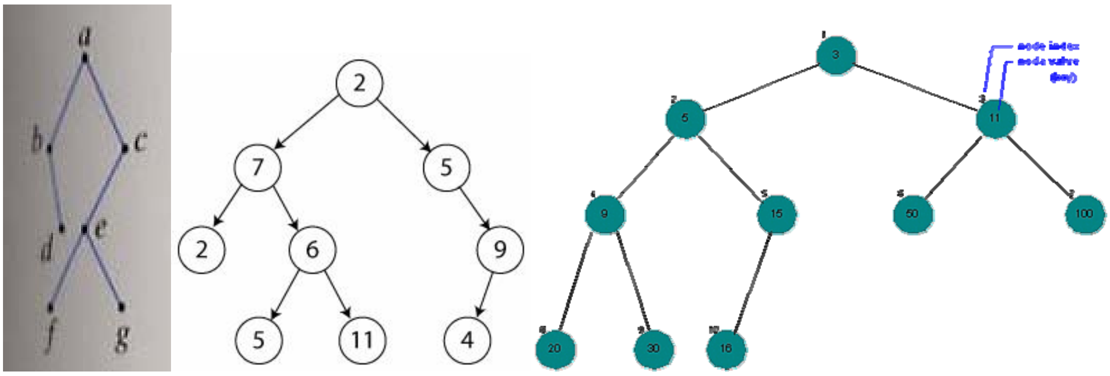
(Sol.)



4-2 Binary Trees

Binary tree: A binary tree is a rooted tree in which each vertex has either 2 children (a left child and aright child), one child (a left child or a right child, but not both), or no child.

Eg. Three examples of binary trees.

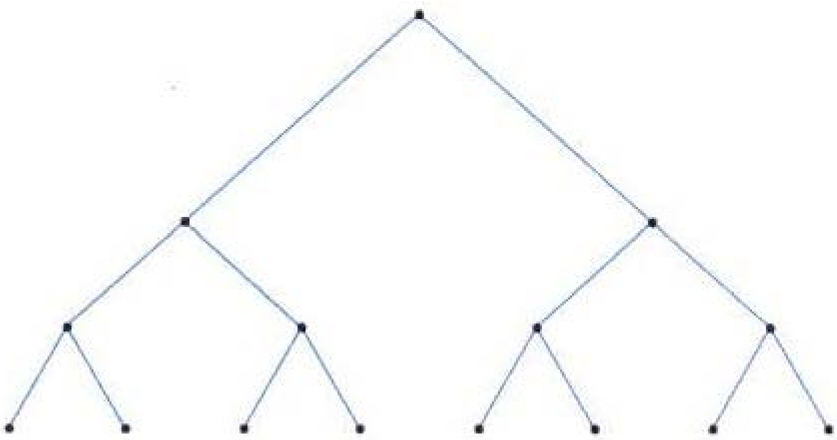


Full binary tree: A full binary tree in which each vertex has either 2 children or zero children.

Theorem If T is a full binary tree with i internal vertices, then T has i +1 terminal vertices and 2i +1 total vertices.

Theorem If a binary tree of height h has t terminal vertices, then $\log_2(t) \leq h$.

Eg. A binary tree has height h and the number of terminals t=8.

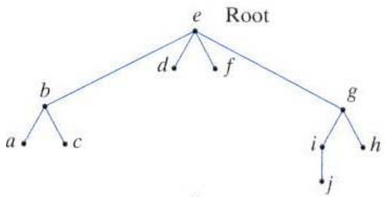


Child-Sibling Rules: To transform a general tree into a binary tree according to the following procedures.

1. Connect all brothers in the same level using the horizontal edges from the left one to the right one.

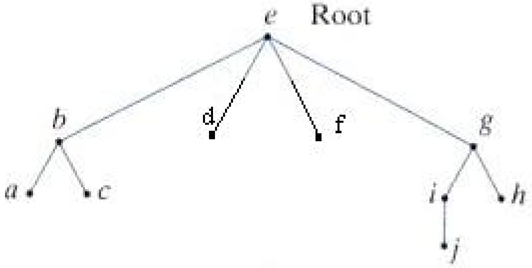
2. Delete all the links between the vertex (node) and its children except the leftmost child.

3. Turn the horizontal edges 45° clockwise.

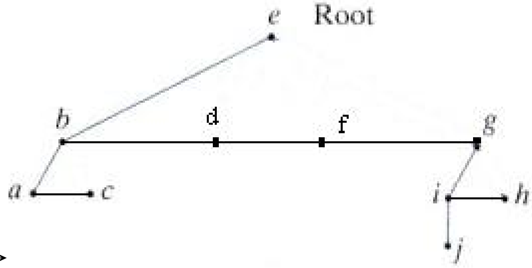


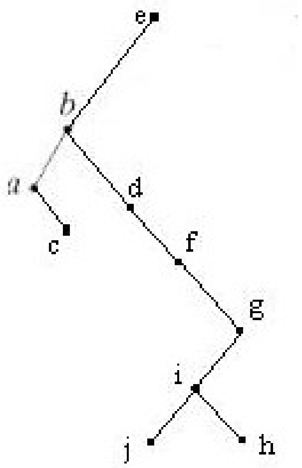
Eg. Transform the left tree into a binary tree.

(Sol.)



→





→

Constructing a Binary Search Tree

Input: A sequence $w_1, \ldots, w_n$ of distinct words and the length n of the sequence

Output: A binary search tree T

```
make_bin_search_tree(w, n) {
  let T be the tree with one vertex, root
  store w_1 in root
  for i = 2 to n {
    v = root
    search = true // find spot for w_i
    while (search) {
      s = word in v
      if (w_i < s)
        if (v has no left child) {
          add a left child l to v
          store w_i in l
          search = false // end search
        }
        else
          v = left child of v
      else // w_i > s
        if (v has no right child) {
          add a right child r to v
          store w_i in r
          search = false // end search
        }
        else
          v = right child of v
    } // end while
  } // end for
  return T
}
```

4-3 Tree Traversals

Preorder traversal algorithm

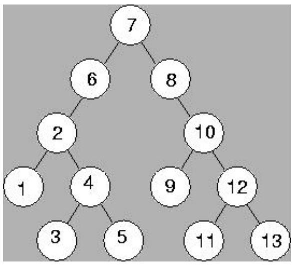
Preorder Traversal

Input: *PT*, the root of a binary tree

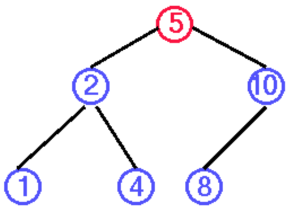
Output: Dependent on how "process" is interpreted in line 3

```
   preorder(PT) {
1.    if (PT is empty)
2.       return
3.    process PT
4.    l = left child of PT
5.    preorder(l)
6.    r = right child of PT
7.    preorder(r)
   }
```

1. **Starting from the root, firstly visited the left child, and then visited the right child.**
2. **After visiting all the left descendants of a node, we can visit its right descendants.**



Eg. For the left binary tree, preorder traversal yields: 7, 6, 2, 1, 4, 3, 5, 8, 10, 9, 12, 11, 13.



Eg. For the left binary tree, preorder traversal yields: 5, 2, 1, 4, 10, 8.

Postorder traversal algorithm

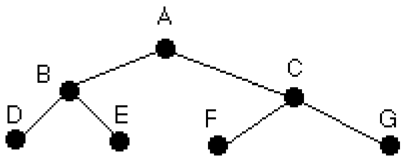
Postorder Traversal

Input: *PT*, the root of a binary tree

Output: Dependent on how "process" is interpreted in line 7

```
   postorder(PT) {
1.    if (PT is empty)
2.       return
3.    l = left child of PT
4.    postorder(l)
5.    r = right child of PT
6.    postorder(r)
7.    process PT
   }
```

1. **Starting from the leftmost child, firstly visited its right brother, and then visited their parent.**
2. **After visiting all the descendants, we can visit the ancestor on the higher level.**



Eg. For the left binary tree, postorder traversal yields: D, E, B, F, G, C, A.

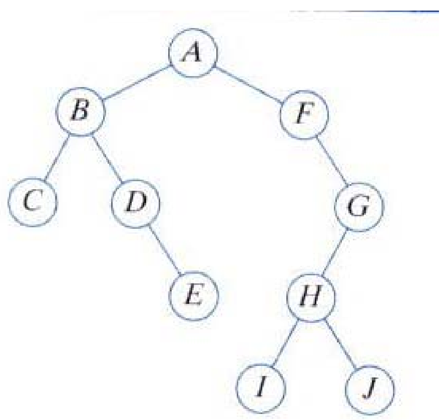
Inorder traversal algorithm

Inorder Traversal

Input: *PT*, the root of a binary tree

Output: Dependent on how "process" is interpreted in line 5

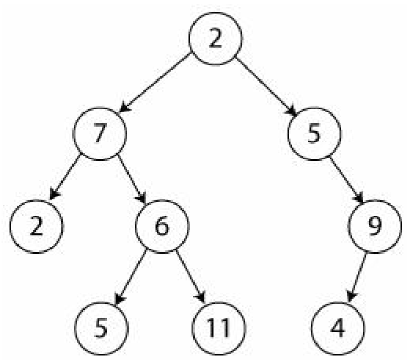
```
    inorder(PT) {
1.      if (PT is empty)
2.         return
3.      l = left child of PT
4.      inorder(l)
5.      process PT
6.      r = right child of PT
7.      inorder(r)
    }
```



Eg. For the binary tree, Preorder traversal is A, B, C, D, E, F, G, H, I, and J.

Postorder traversal is C, E, D, B, I, J, H, G, F, and A.

Inorder traversal is C, B, D, E, A, F, I, H, J, and G.



Eg. For the binary tree, the preorder traversal yields: 2, 7, 2, 6, 5, 11, 5, 9, 4. And the postorder traversal yields: 2, 5, 11, 6, 7, 4, 9, 5, 2. The inorder traversal yields: 2, 7, 5, 6, 11, 2, 5, 4, 9

Application of tree traversals: Facilitating the computer evaluation of arithmetic expression.

Rules of transforming arithmetic expression into a binary tree and then obtaining the prefix, the infix, and the postfix forms:

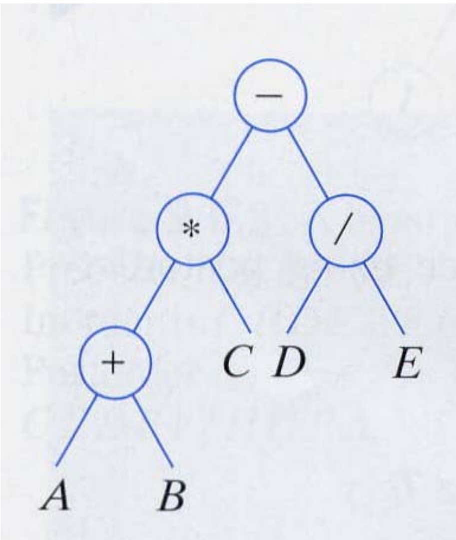
1. Add appropriate parentheses in the arithmetic expression.

2. Starting from the innermost parenthesis, the operator is as the root, the left operand is as the left child and the right operand is as the right child.

3. Obtain the prefix, the infix, and the postfix forms according to the preorder, the inorder, and the postorder traversal algorithms, respectively.

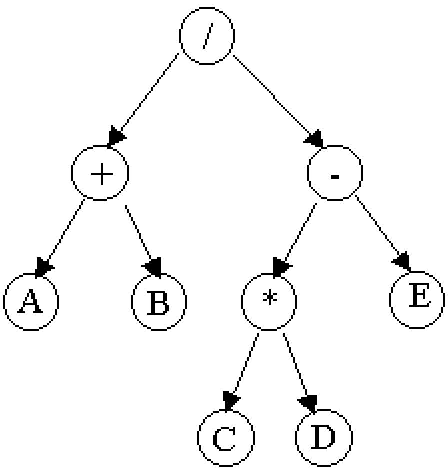
Eg. Transform an expression involving the operators (A+B)*C-D/E into the prefix, the infix, and the postfix forms.

(Sol.) Rewrite (A+B)*C-D/E into (((A+B)*C)-(D/E)) and obtain the tree as



Prefix form: -*+ABC/DE

Postfix form: AB+C*DE/-



Eg. Transform an expression involving the operators (A+B)/((C*D)-E) into the prefix, the infix, and the postfix forms.

(Sol.) Traverse the left binary tree, we have

Prefix form: /+AB-*CDE

Infix form: A+B/C*D-E

Postfix form: AB + CD*E-/