A general algorithm for performing such an insertion is as follows:

1. ***If*** *the existing tree contains no nodes then append the new node as the root node of the tree and exit*
2. *Set the current pointer to the root node*
3. *Repeat step 4 while remaining tree is not empty*
4. ***if*** *the new name is lexically less than the current node’s name,*

***then***

***if*** *the left sub-tree is not empty,*

***then***

*Set the current pointer to the root of the left sub-tree*

***else***

*Append the new name as a left leaf to the present tree*

***exit***

***else-if*** *the right sub-tree is not empty*

***then***

*Set the current pointer to the root of the right sub-tree*

***else***

*Append the new name as a right leaf to the present tree*

***exit***

**Pseudo-Code**

struct treenode {

  int info;

treenode \*leftpointer;

treenode \*rightpointer;

};

//////////////////

treenode Createnode(int s)

{

Treenode newNode=new treenode;

newNode->info=s;

newNode->left=NULL;

newNode->right=NULL;

return newNode;

}

////////////////////////

Insert\_node (treenode Root, int x);

{

If (x < root-> info)

{

if (root-> leftpointer= = null)

{

root->leftpointer = createnode(x)

else

Insert\_node(Root->leftpointer, x)

}

}

else if (x > root -> info)

{

If( root-> rightpointer = null)

{

root-> rightpointe = createnode(x)

else

Iinsert\_node(root->rightpointer, x)

  }

}

else

cout<<”Duplicate node”<< X;

}

1. *Find the node marked for deletion and its parent. If the node to be deleted is the root, then head will be its parent.*
2. *If the node is not found, then display an appropriate message and return*
3. *If the node being deleted has a left empty subtree, then append its right subtree to the parent node and return. If the node being deleted has a right empty subtree, then append its left subtree to the parent node and return*
4. *Obtain the inorder successor of the node to be deleted and the parent of the inorder successor*
5. *If the node to be deleted is the parent of its inorder successor, then append the left subtree of the node for deletion as the left subtree of its inorder successor (which has no left subtree)*
6. *else append the right subtree of the successor as the left subtree of the successor’s parent*
7. *Append the right subtree of the node to be deleted as the right subtree of its successor*
8. *Append the left subtree of the node to be deleted as the left subtree of its successor*
9. *Connect the parent of the node to be deleted to the inorder successor and return*

**TREE DELETE**

1. **void treedelete(treenode\* x)**
2. **{**
3. /\* Find the node to be deleted\*/
4. If (not found)
5. /\* Display message and return\*/
6. Else
7. { if ((p->info==root->info)&&(root->left==root->right==NULL))
8. { delete root;}
9. Else if (p->left==NULL)
10. { if (isLeft(p))
11. { p->father->left=p->right;
12. Delete p;}
13. Else
14. { p->father->right=p->right;
15. Delete p;}
16. }
17. if (p->right==NULL)
18. { if (isLeft(p))
19. { p->father->left=p->left;
20. Delete p;}
21. Else
22. { p->father->right=p->left;
23. Delete p;}
24. }
25. If(isTerminal(p))
26. {if(isLeft(p))
27. {father(p)->left=NULL; delete p;}
28. Else
29. { father(p)->right=NULL; delete p;}
30. }
31. If((p->left==p->right!=NULL)&&(brother(p)==NULL))
32. {father(p)->left=p->left;
33. Father(p)->right=p->right; delete p;}
34. Else
35. Cout<<”Cant be deleted”;
36. }

**PRIMITIVE OPERATIONS ON TREES**

There are certain primitive operations needed to check the association of any node with some other node in tree. Some are listed such that:

* + Left (p): returns a pointer pointing to left node of *node p* if any

Treenode Left (Treenode p) {return p->left;}

* + Right (p): returns a pointer pointing to right node of *node p* if any

Treenode Right (Treenode p) {return p->right;}

* + Father (p): returns a pointer pointing to parent node of *node p* if any

Treenode Father (Treenode p) {return p->father;}

* + Brother (p): returns a pointer pointing to sibling of *node p* if any

Treenode Brother (Treenode p)

{if (p->father==NULL) /\* Root \*/

Return (NULL);

Else if (father(p)->left==p)

Return (right(father(p)));

Else if (father(p)->right==p)

Return (left(father(p)));

}

Similarly, there are some other operations are also discussed used for checking the where about of any node in tree like whether that node is left of, right of, parent of some other node.

* isLeft (p): returns true if *node p* is left of some other node

bool isLeft(nodeptr p)

{ if (p->father==null)

return false;

else if (p->father->right= = p)

return false;

else

return true;

}

* isRight (p): returns true if *node p* is right of some other node

bool isRight(nodeptr p)

{ if (p->father==null)

return false;

else if (p->father->left= = p)

return false;

else

return true;

}

* isFather (p): returns true if *node p* is father of some other node

bool isFather(nodeptr p)

{ if ((p->left!=NULL)&&(p->right!=NULL))

return false;

else

return true;

}

* isBrother (p): returns true if *node p* is sibling of some other node

bool isBrother(nodeptr p)

{if(((isLeft(p))&&(p->Father->right!=NULL))|| ((isRight(p))&&(p->Father->left!=NULL)))

return true;

else

return false;}

Void pretrav (NODEPTR tree)

{

if (tree!=NULL) // till leaves or terminals

{ cout<<tree->info; // prints the node data

pretrav(tree->left); // calls the next recursively

pretrav(tree->right);

}

}

Void intrav (NODEPTR tree)

{

if (tree!=NULL) // till leaves or terminals

{ pretrav(tree->left);

cout<<tree->info; // prints the node data

pretrav(tree->right);

}

}

Void posttrav (NODEPTR tree)

{

if (tree!=NULL) // till leaves or terminals

{ pretrav(tree->left);

pretrav(tree->right);

cout<<tree->info; // prints the node data

}

}