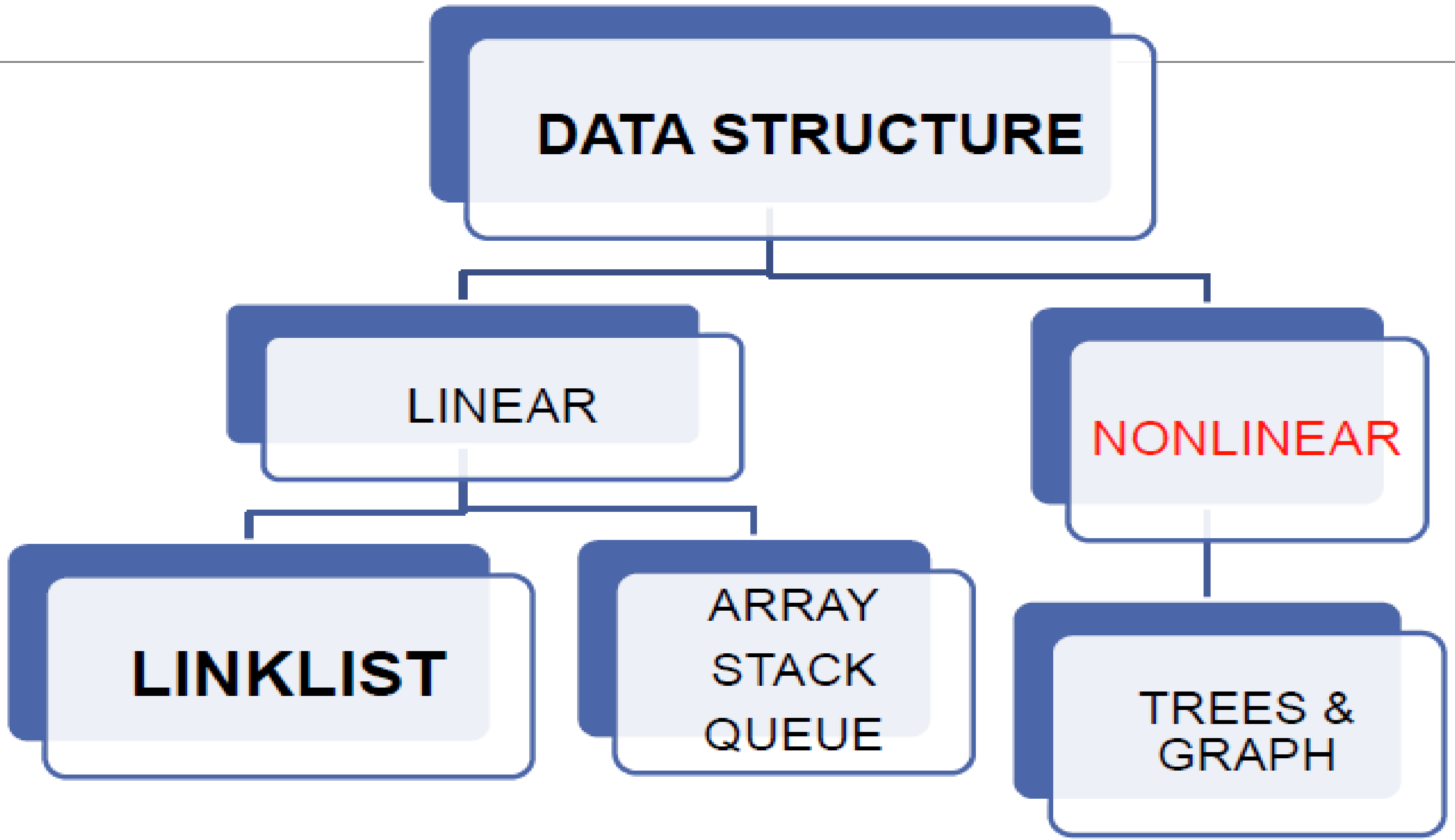
CSE2PM01A/ AID2PM01A Data Structures

S. Y. B. Tech CSE / AIDS

Semester – III

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

Types of Data Structures

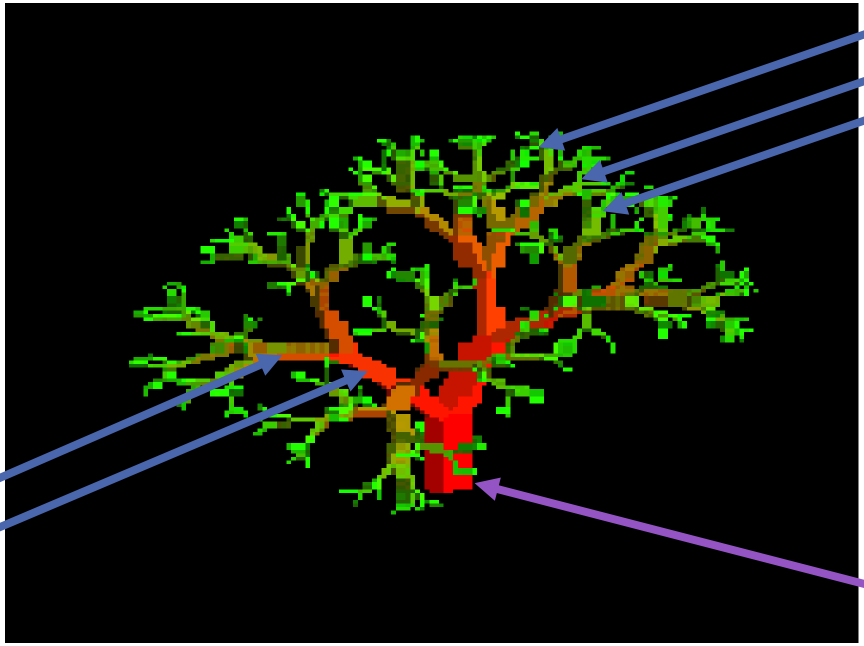


Tree

- Basic Terminology, Binary Tree- Properties
- Converting Tree to Binary Tree.
- Representation using Sequential and Linked organization .
- Binary tree creation and Traversals, Operations on binary tree.
- Binary Search Tree (BST) and its operations
- Threaded binary tree- Creation and Traversal of inorder Threaded Binary tree.
- **Case Study**- Expression tree.

Natural environment Tree

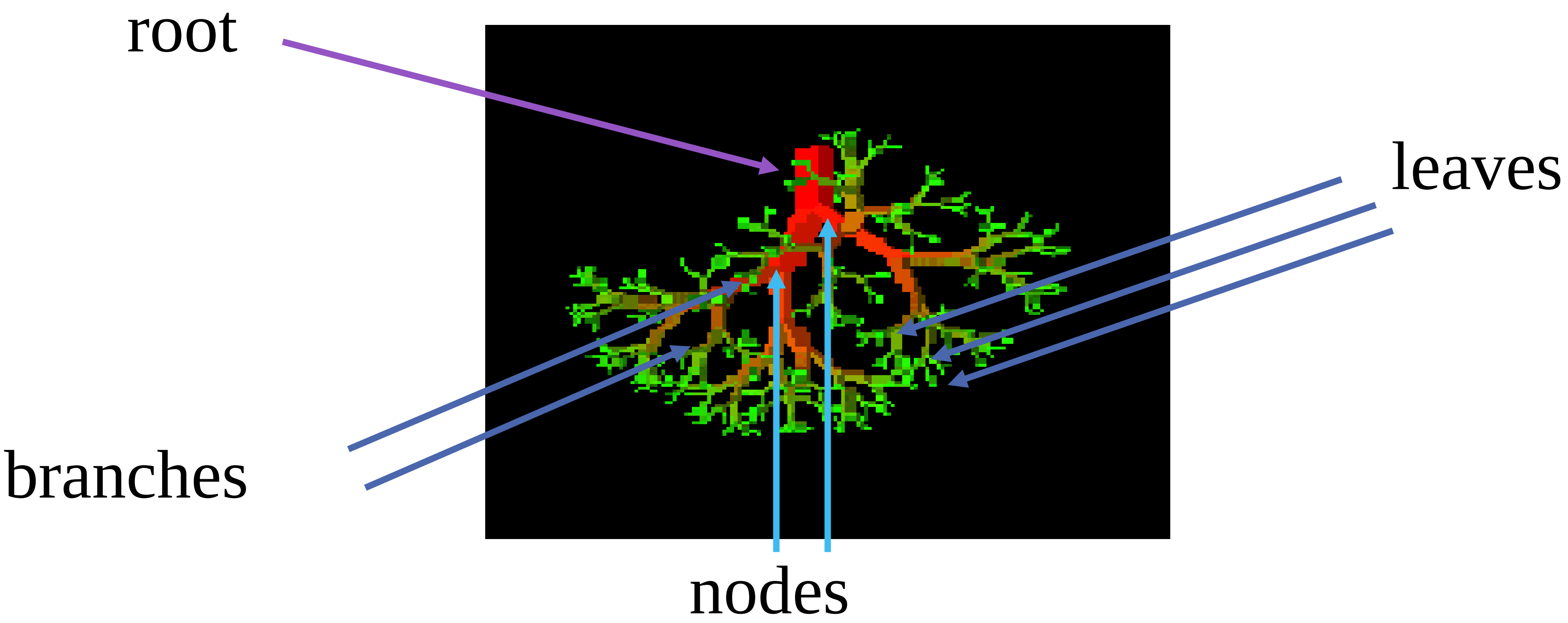
leaves



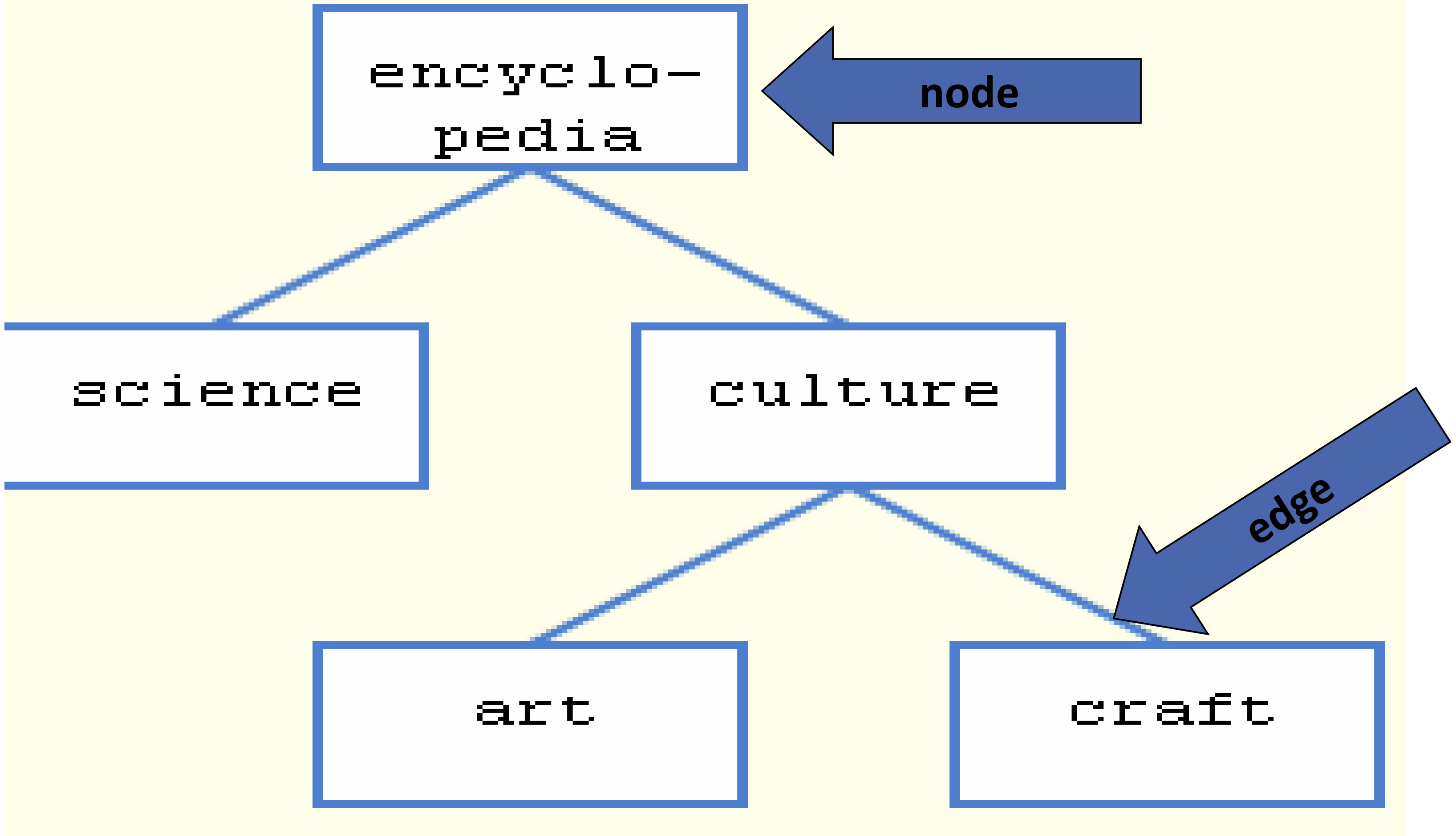
branches

root

Computer Scientist’s View



Tree (example)



General tree

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

(i)There is a specially designated node called the root;

(ii) The remaining nodes are partitioned into n >=0 disjoint sets T1, ...,Tn where each of these sets is a tree. T1, ...,Tn are called the subtrees of the root.

Sample Tree

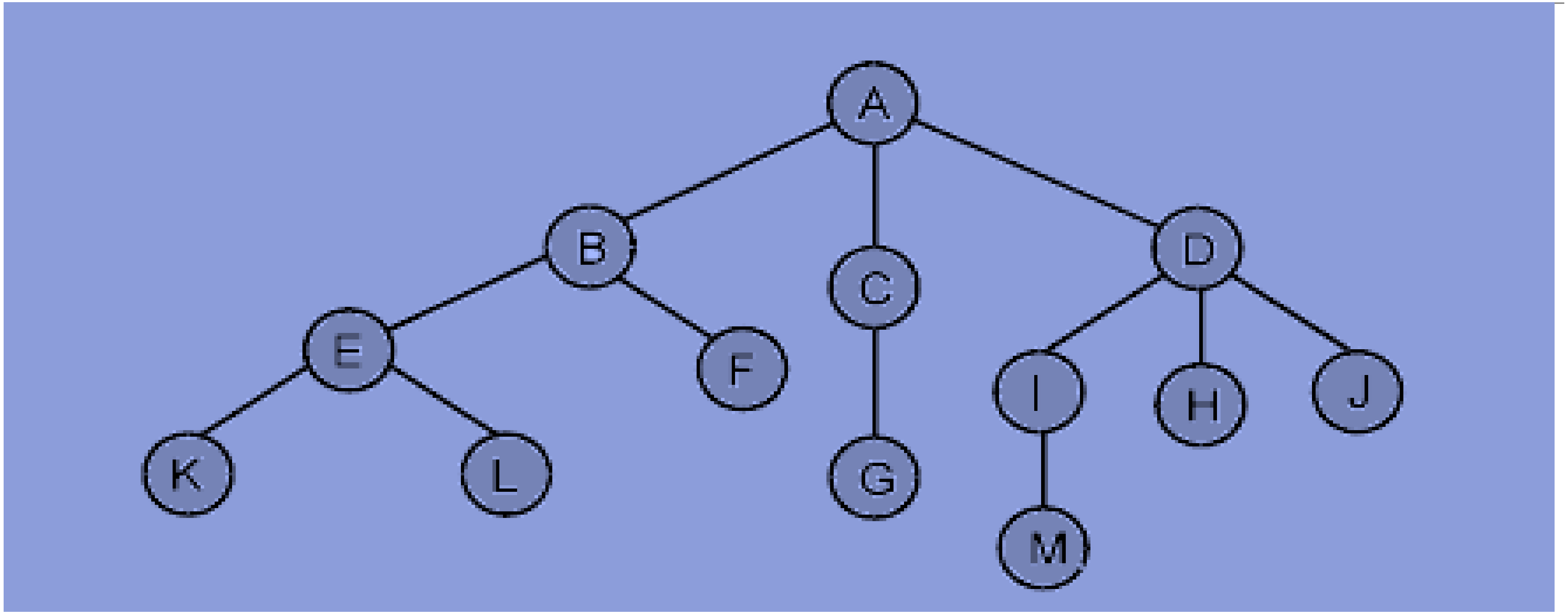


Figure 8: Sample Tree

Tree Terminology

Root: Node without parent (A)

Siblings: Nodes share the same parent

Ancestors of a node: all the nodes along the path from root to that node

Descendant of a node: child, grandchild, grand-grandchild, etc.

The height or depth of a tree is defined to be the maximum level of any node in the tree.(4)

Degree of a node: the number of subtrees(children) of a node is called degree

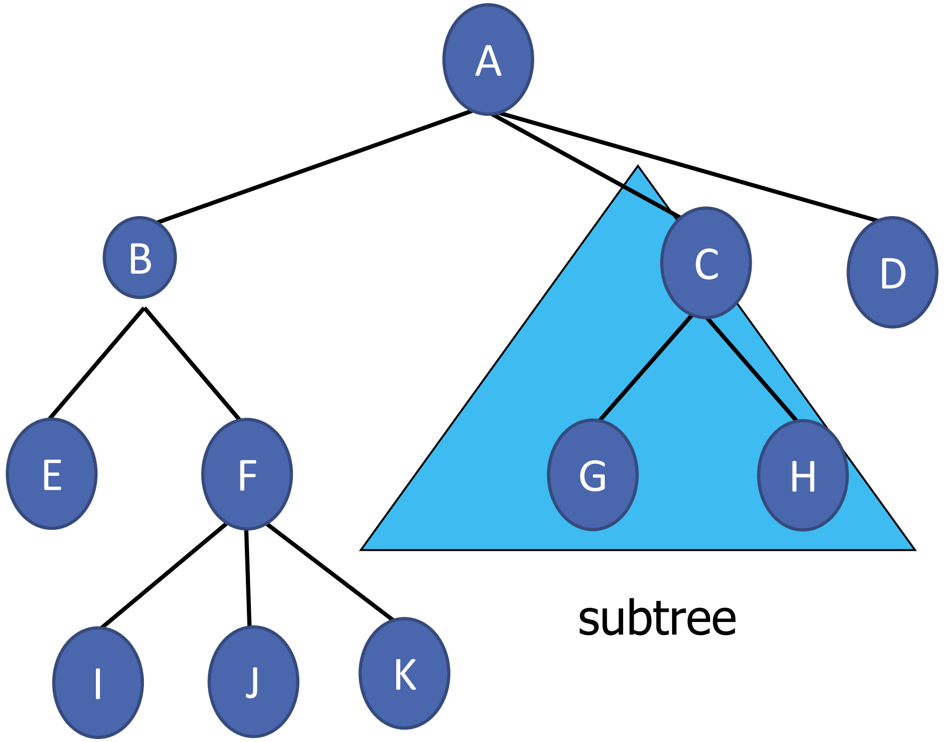
Degree of a tree: the maximum of the degree of the nodes in the tree.

Nonterminal nodes: other nodes

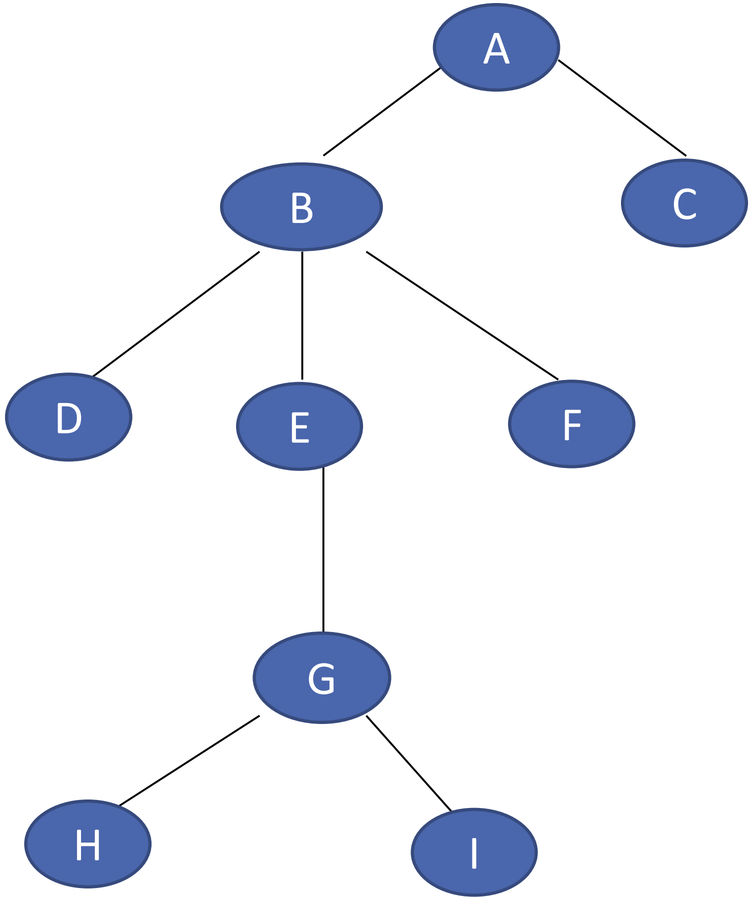
leaf or terminal node: Node that have degree zero (E, I, J, K, G, H, D)

The level of a node is defined by initially letting the root be at level one. If a node is at level l, then its children are at level l + 1.

Subtree: Tree consisting of a node and its descendants



Tree Properties



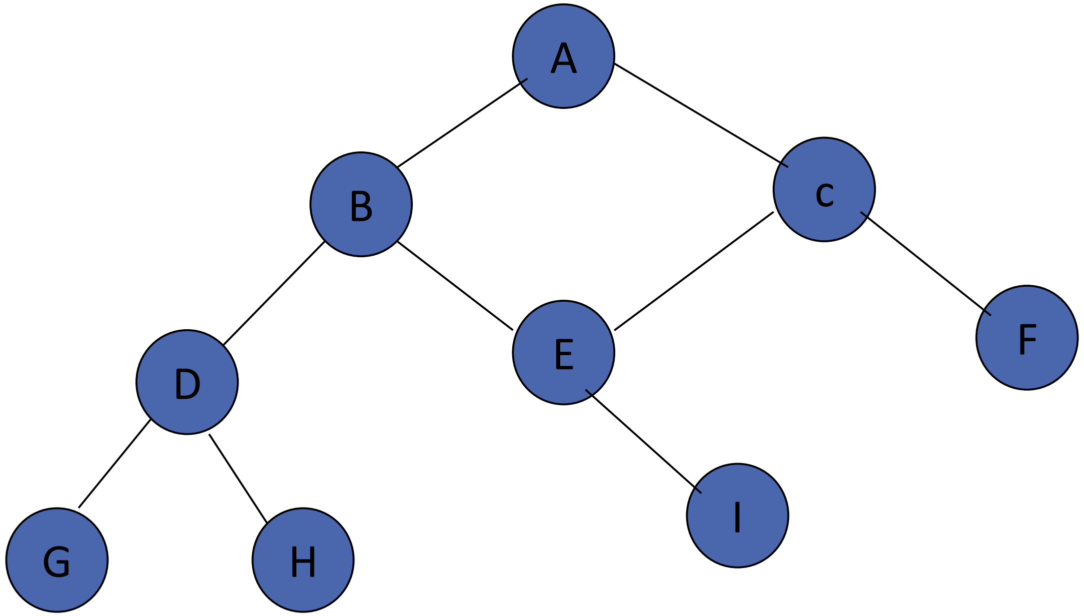
Property	Value
Number of nodes	9
Height	5
Root Node	A
Leaves	C,D,F,H,I
Interior nodes	B,E,G
Ancestors of H	A,B,E,G
Descendants of B	D,E,G,H,I,F
Siblings of E	D,F
Right subtree of A	A,C
Degree of this tree	3

Binary Tree

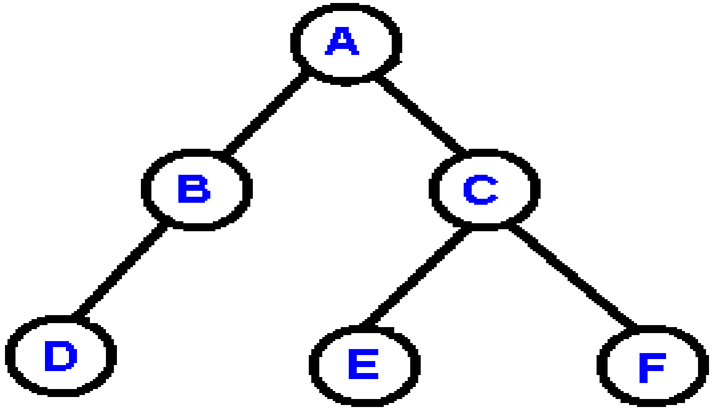
- Every node in a binary tree can have at most two children.

- 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*.

Structures that are not binary trees



Binary tree



Maximum Number of Nodes in BT

- 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.

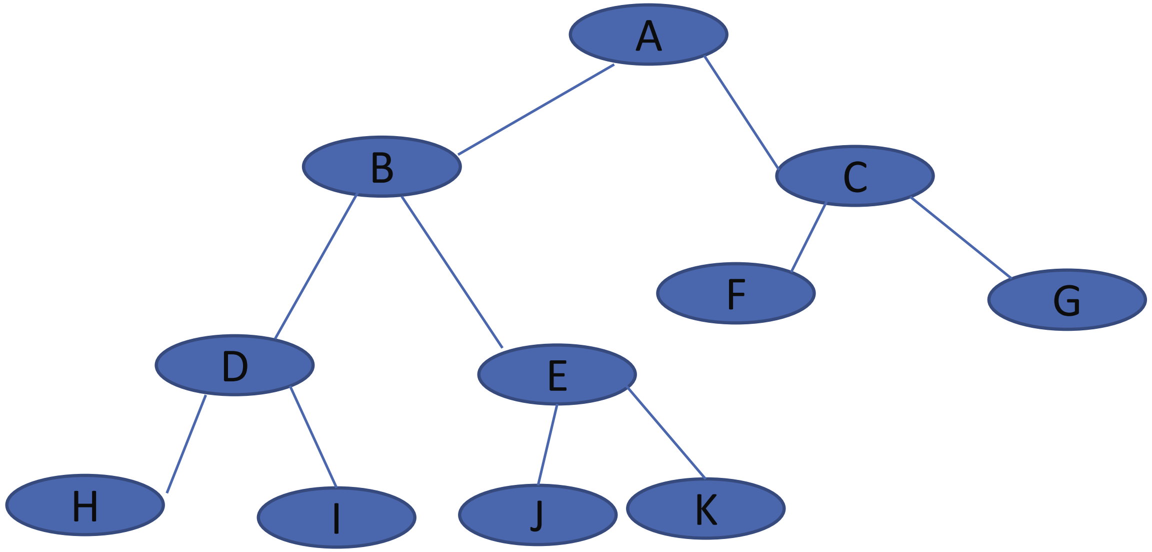
Binary Trees

- **A Full binary tree** of depth K is a binary tree of depth having 2^k-1 nodes k>=0

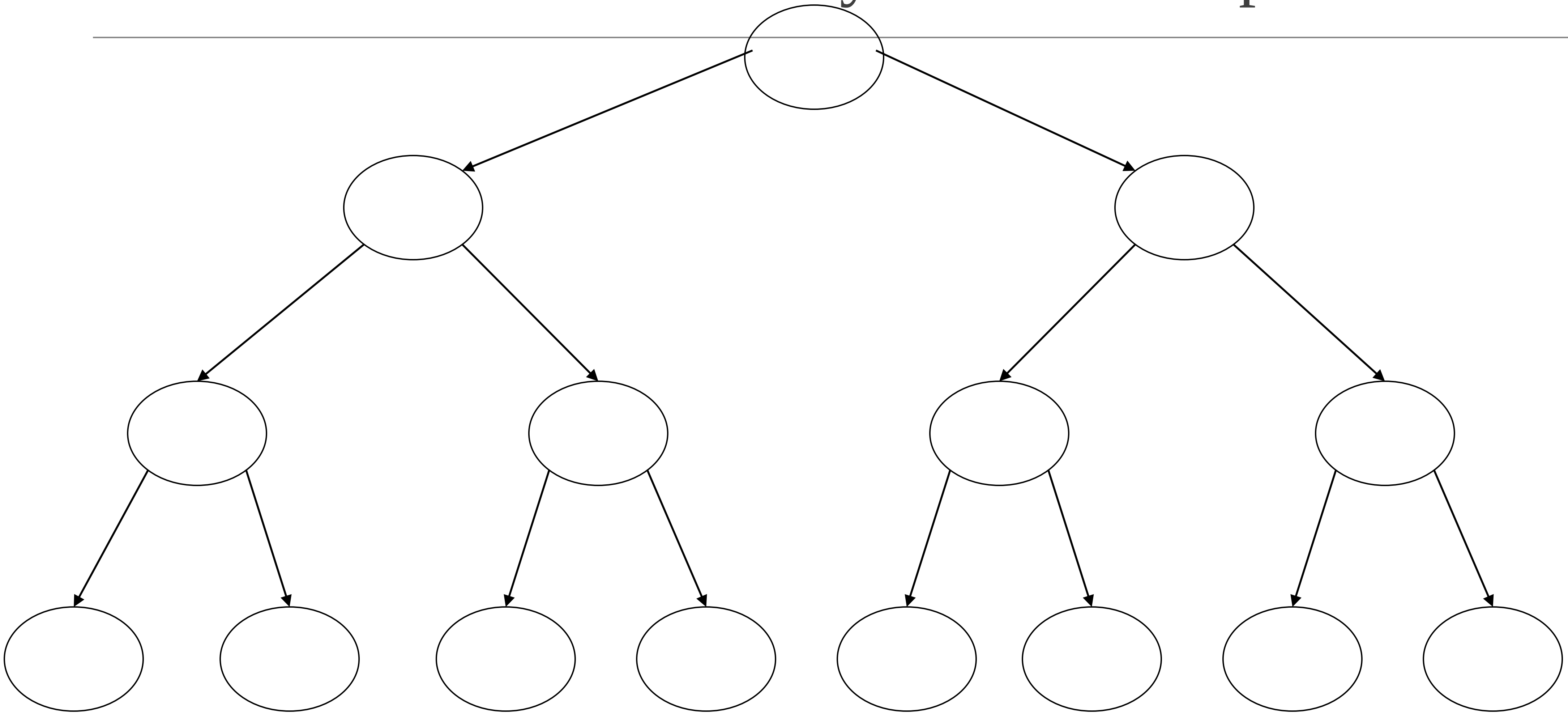
- **Complete Binary Tree**

  A binary tree T with n levels is *complete if all* levels except possibly the last are completely full, and the last level has all its nodes to the left side.

Complete Binary Trees - Example



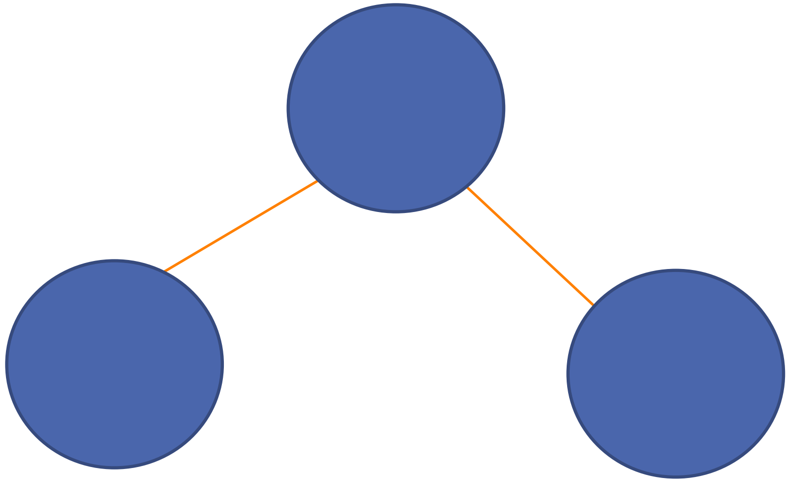
A Full Binary Tree - Example



Complete Binary Trees

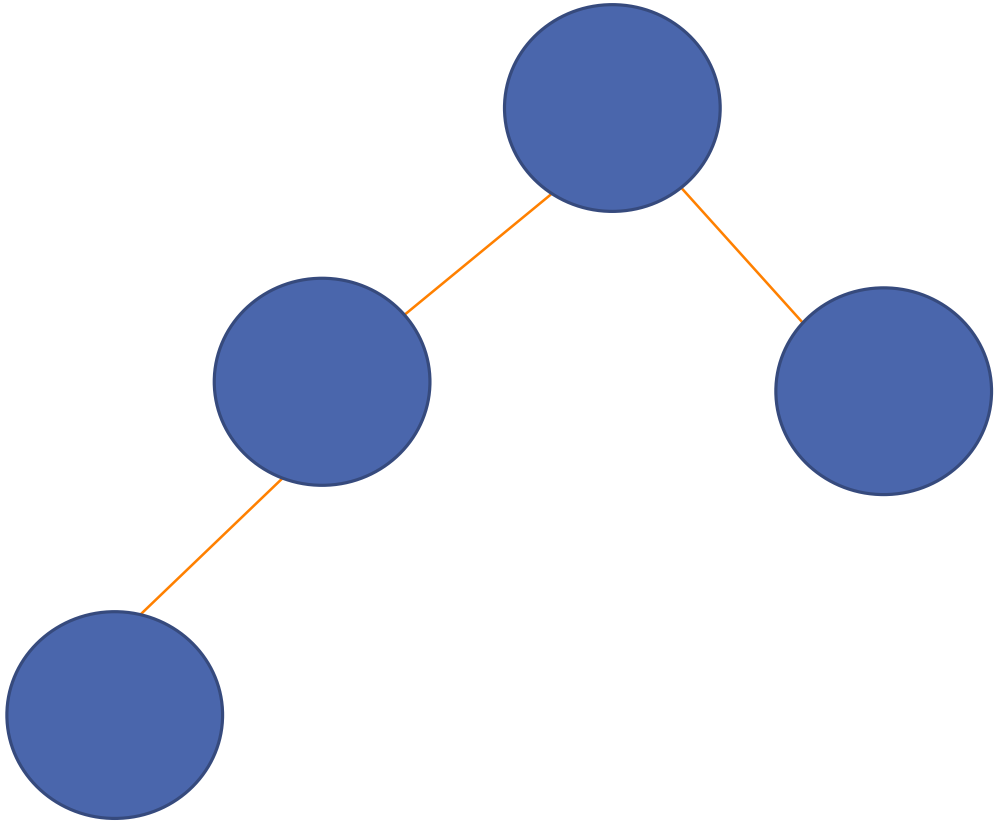
The second node of a complete binary tree is always the left child of the root...

... and the third node is always the right child of the root.



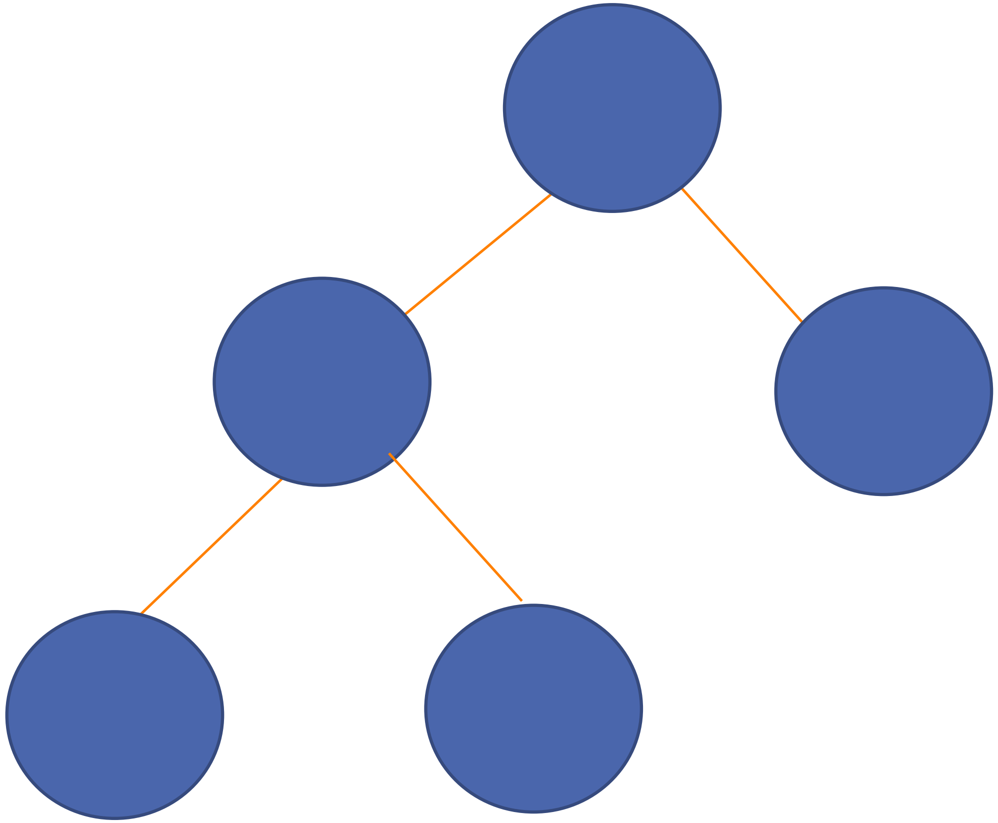
Complete Binary Trees

The next nodes must always fill the next level from left to right.



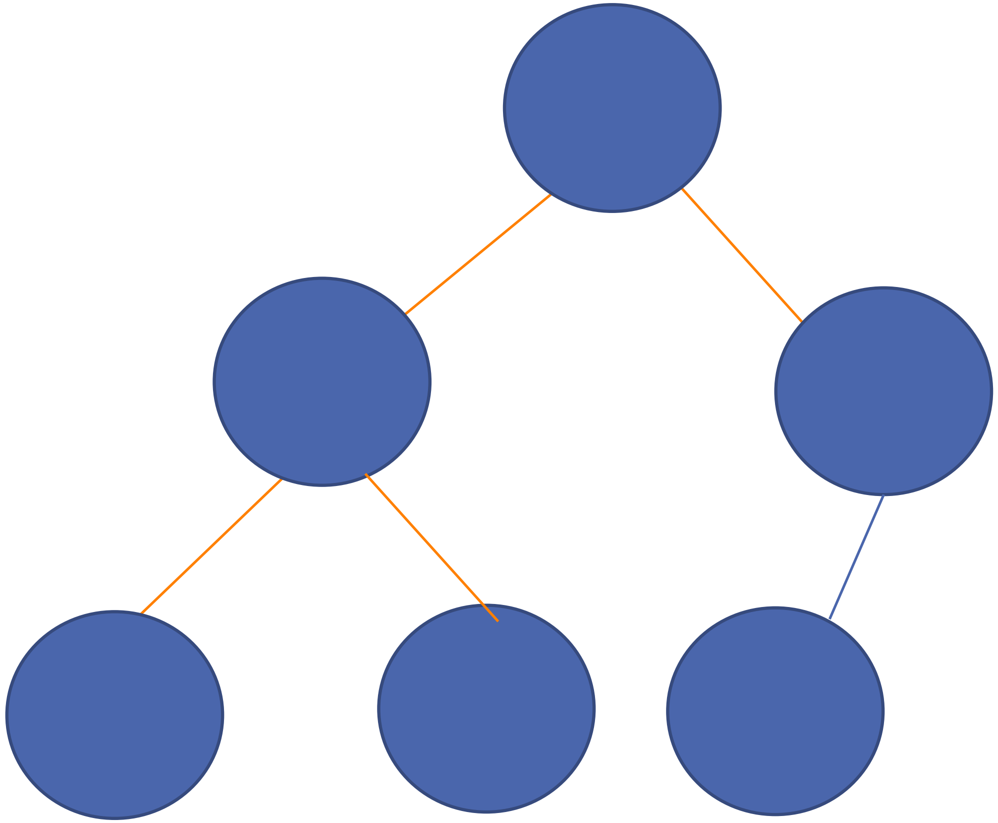
Complete Binary Trees

The next nodes must always fill the next level from left to right.



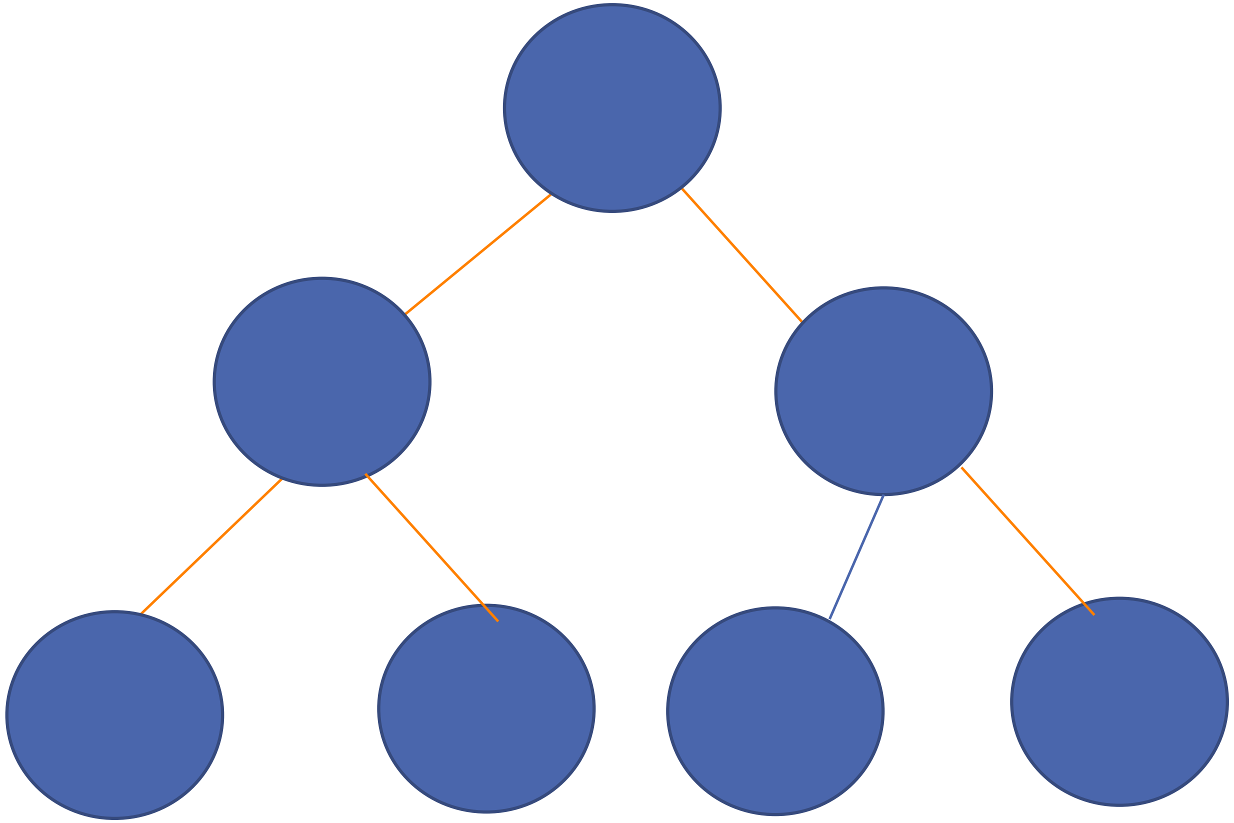
Complete Binary Trees

The next nodes must always fill the next level from left to right.



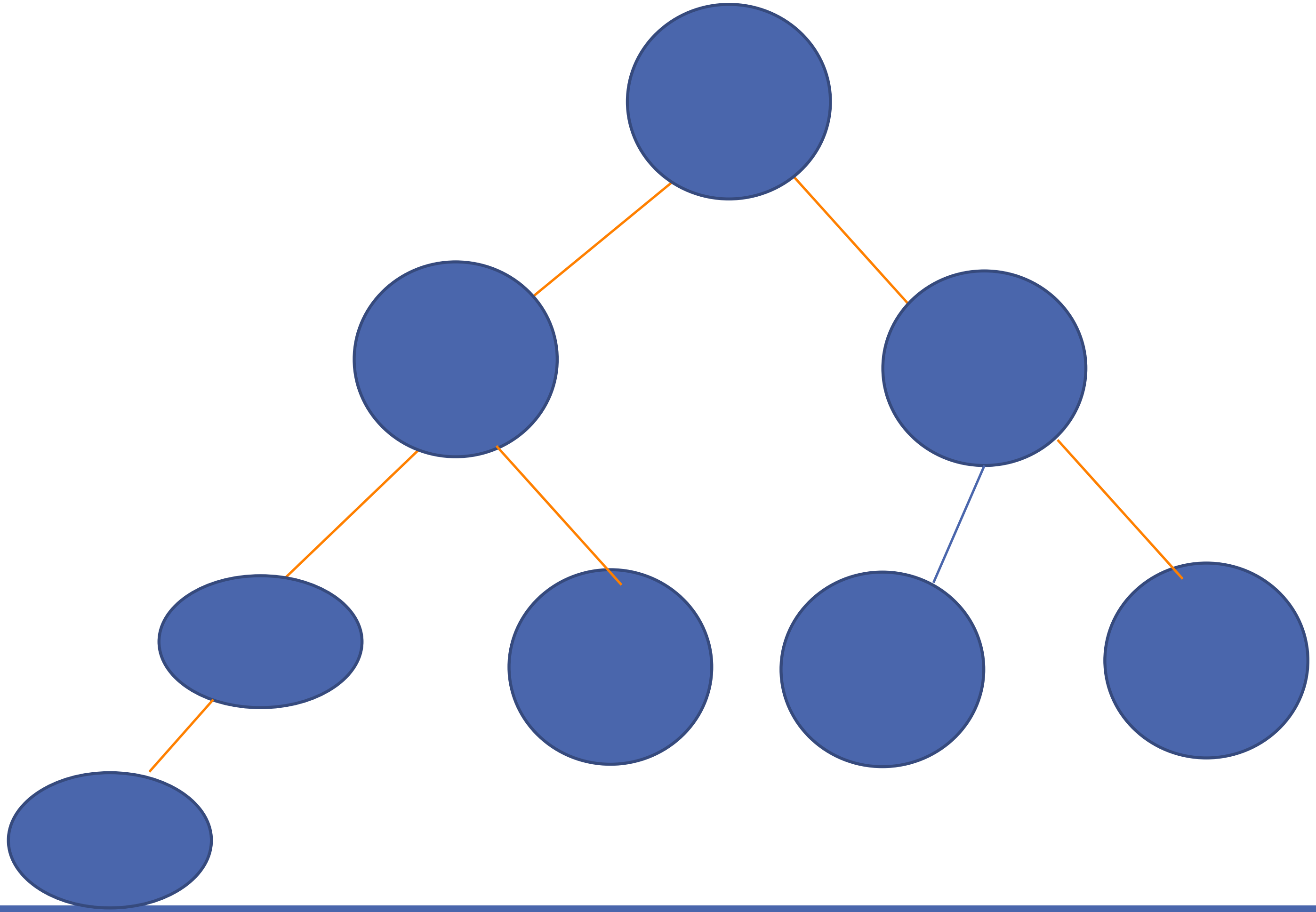
Complete Binary Trees

The next nodes must always fill the next level from left to right.



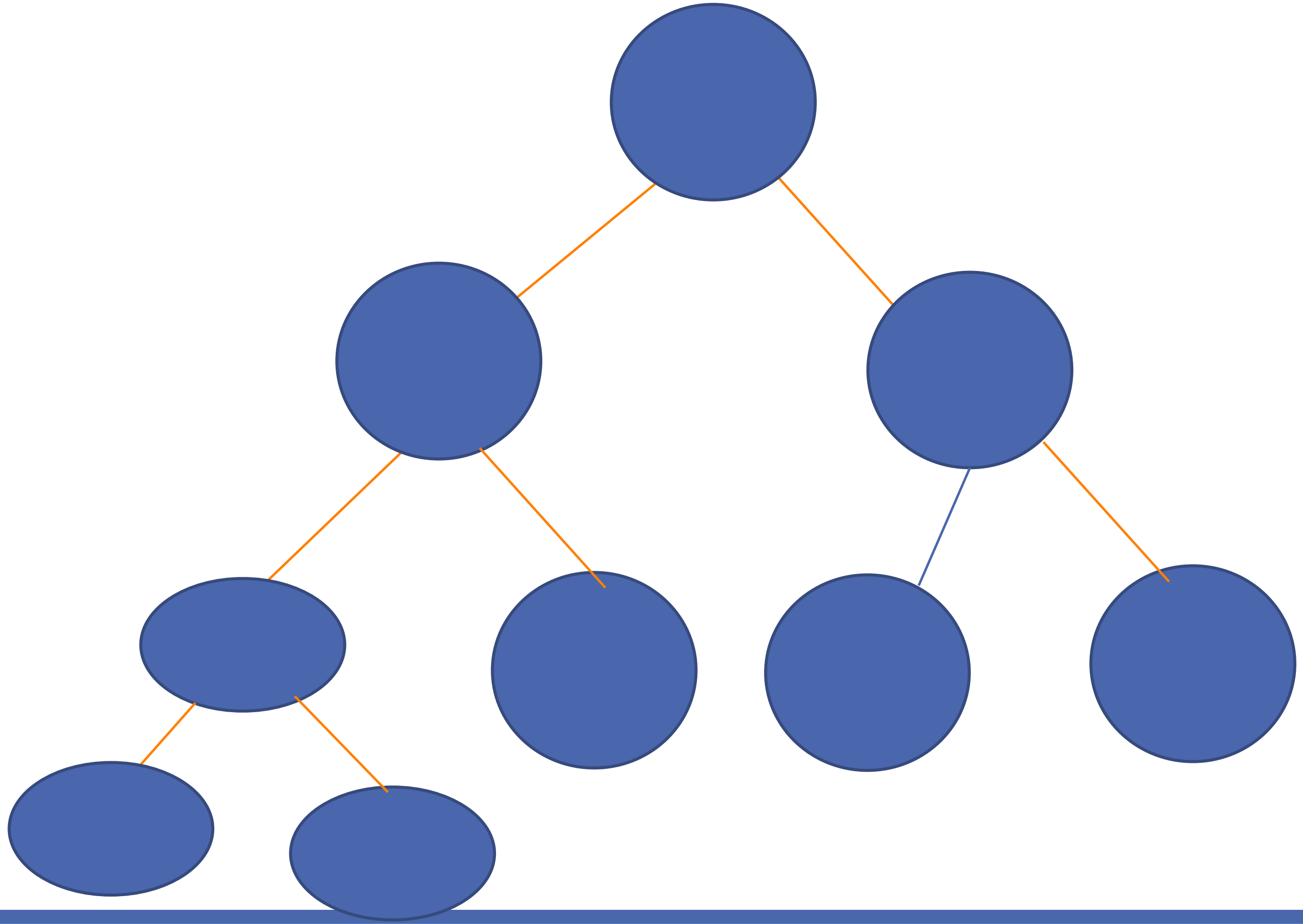
Complete Binary Trees

The next nodes must always fill the next level from left to right.

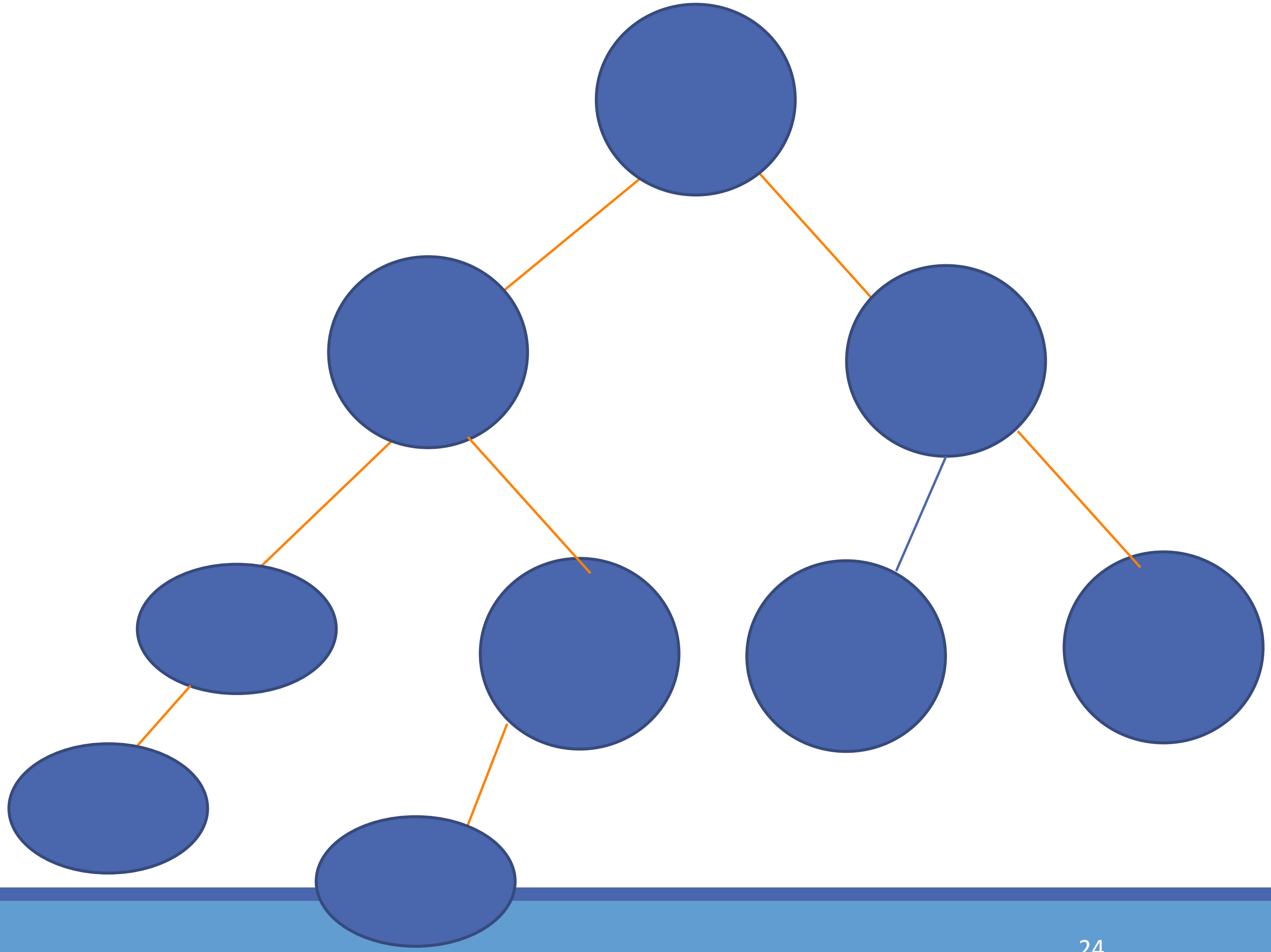


Complete Binary Trees

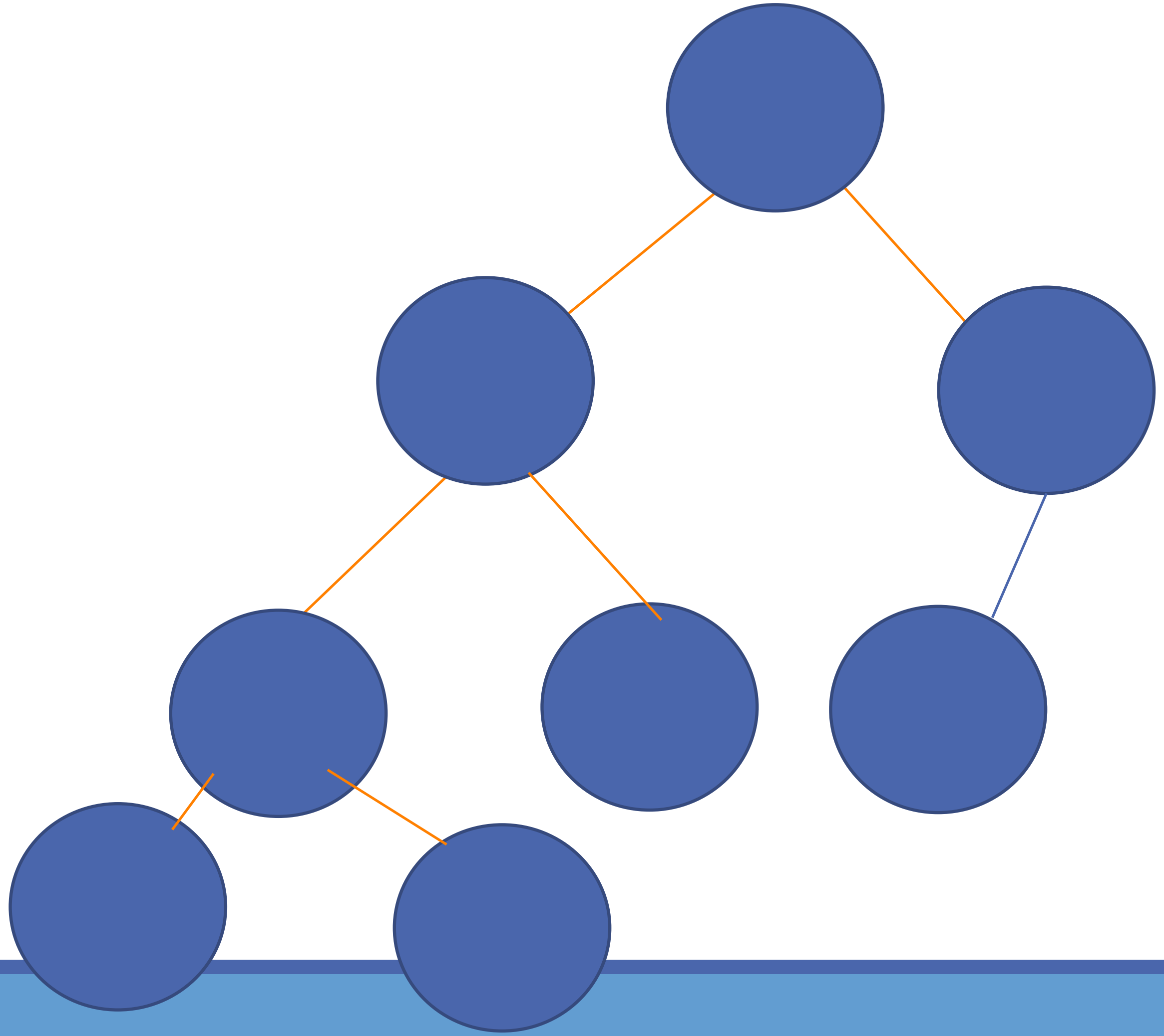
The next nodes must always fill the next level from left to right.



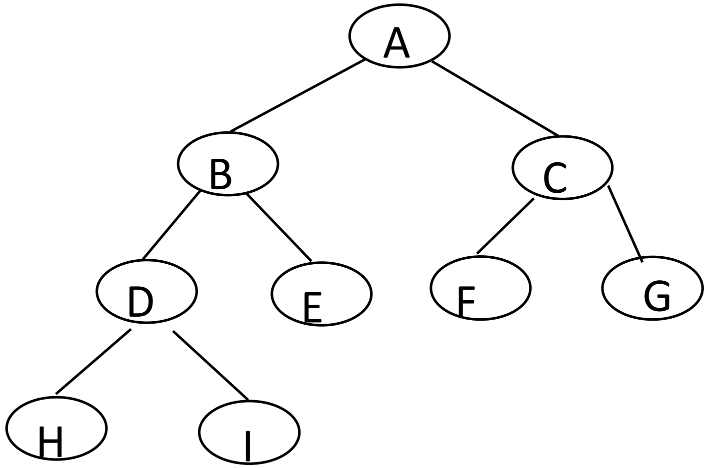
Is This Complete?

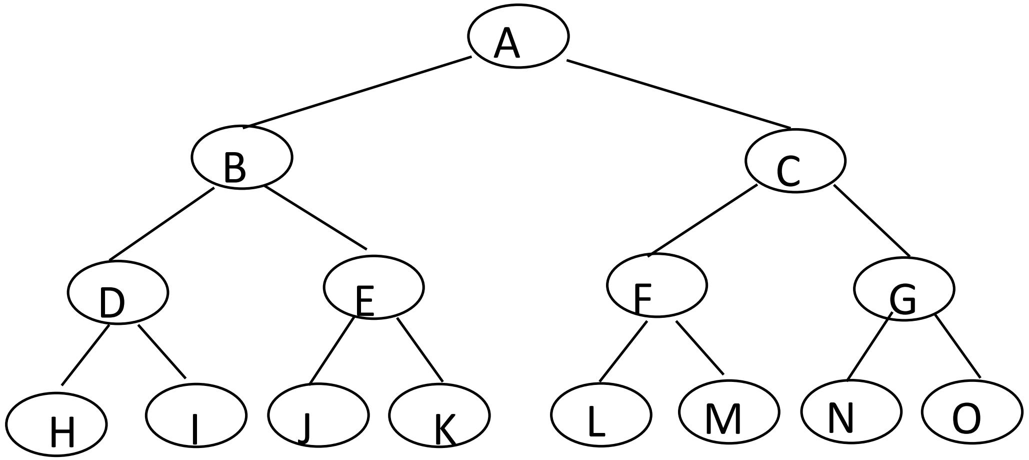


Is This Complete?



Full BT VS Complete BT

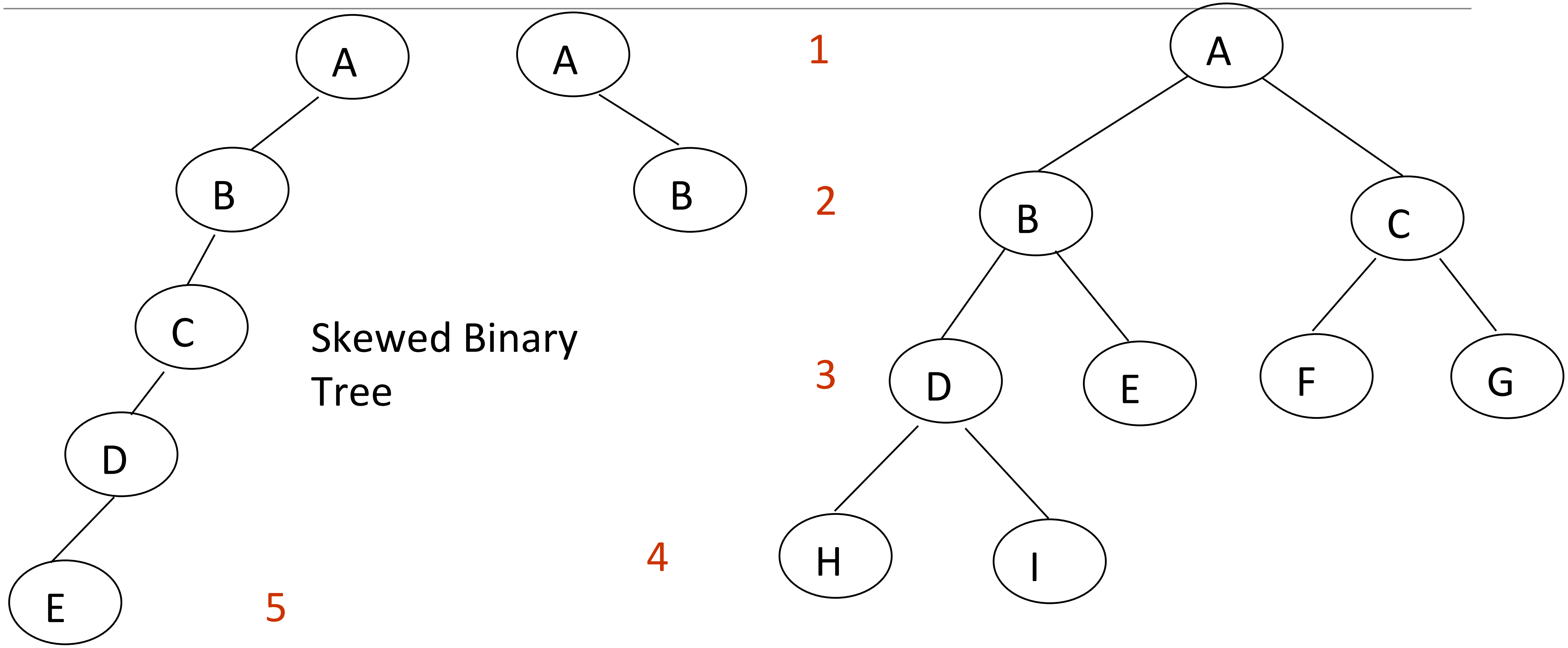




Complete binary tree

Full binary tree of depth 4

Samples of Trees



Skewed Binary Tree

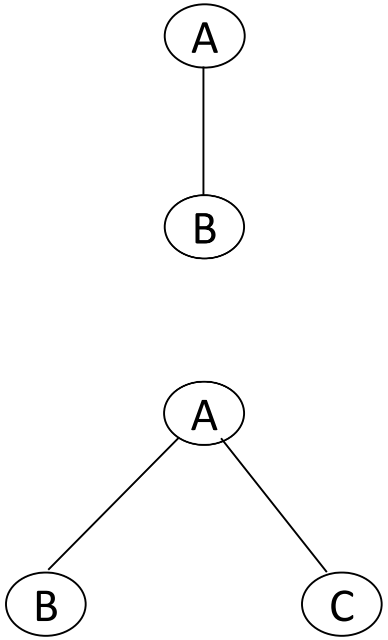
Complete Binary Tree

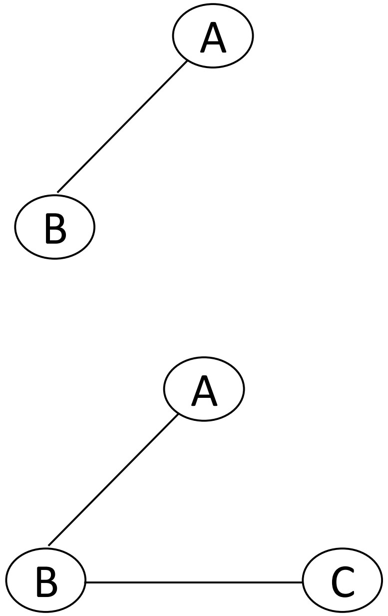
Converting tree to binary tree

- Any tree can be transformed into binary tree.
  - by left child-right sibling representation

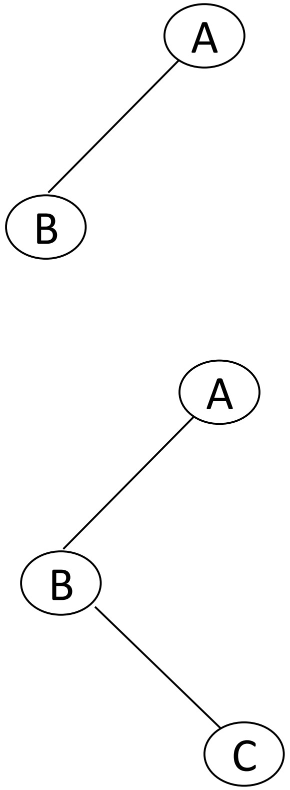
- The left subtree and the right subtree are distinguished.

Tree Representations





Left child-right sibling

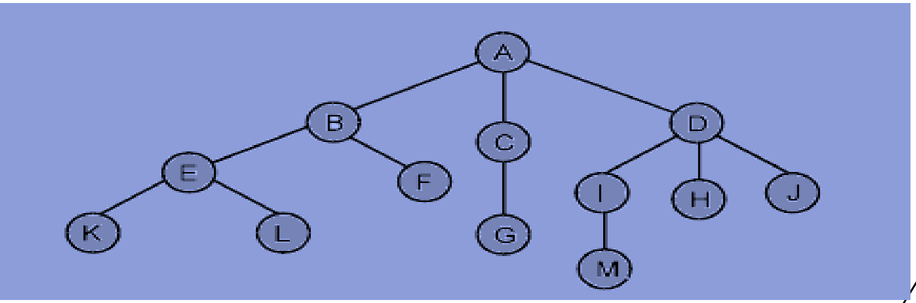


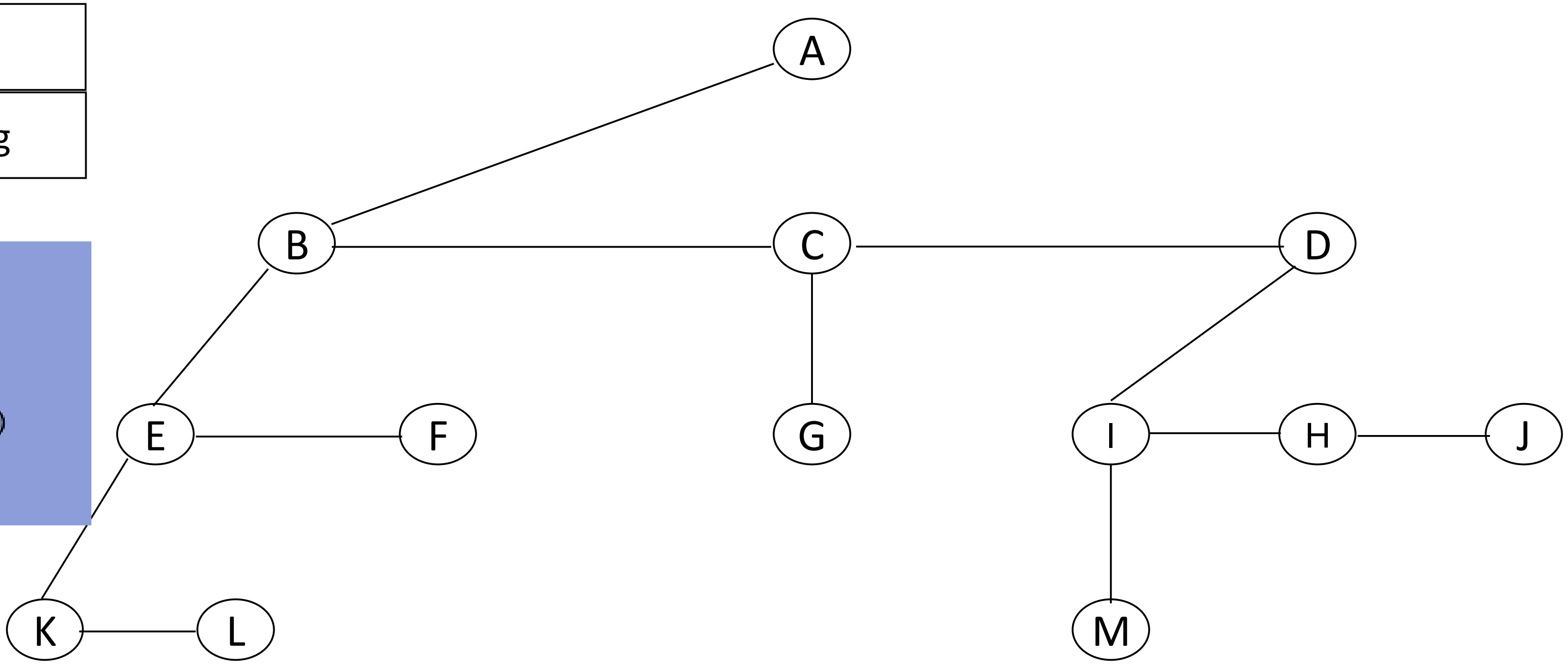
Binary tree

Representation of Trees

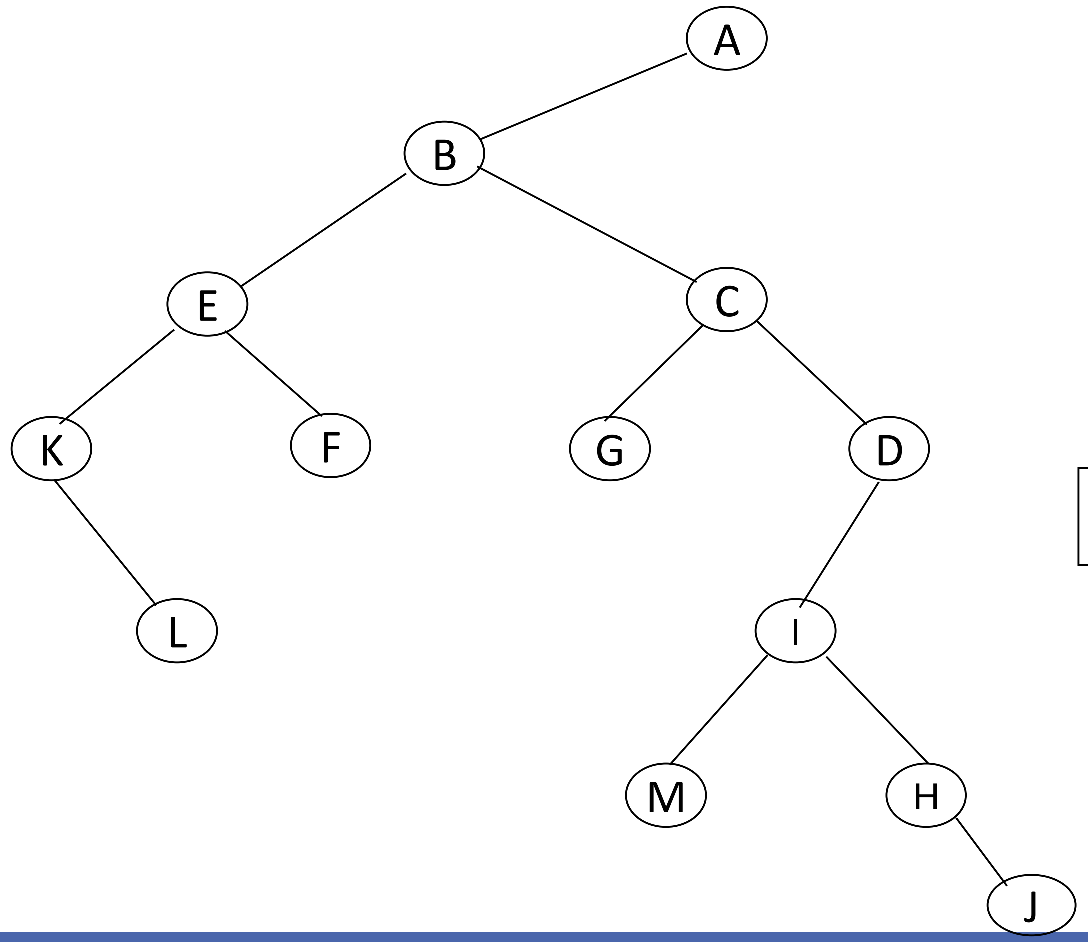
- Left Child-Right Sibling Representation
  - Each node has two links (or pointers).
  - Each node only has one leftmost child and one closest sibling.

<table>
<tr><td colspan="2">data</td></tr>
<tr><td>left child</td><td>right sibling</td></tr>
</table>





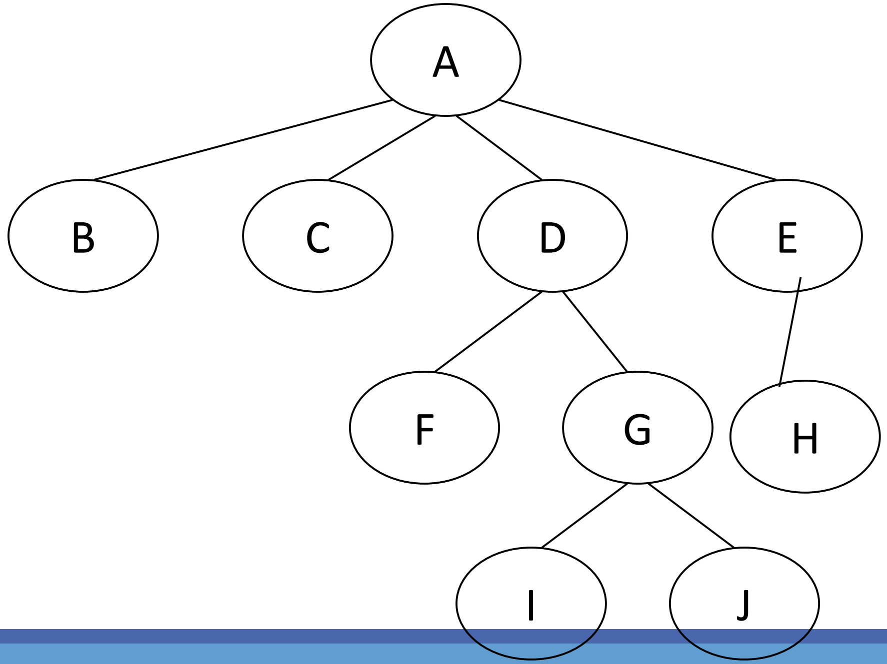
Degree Two Tree Representation

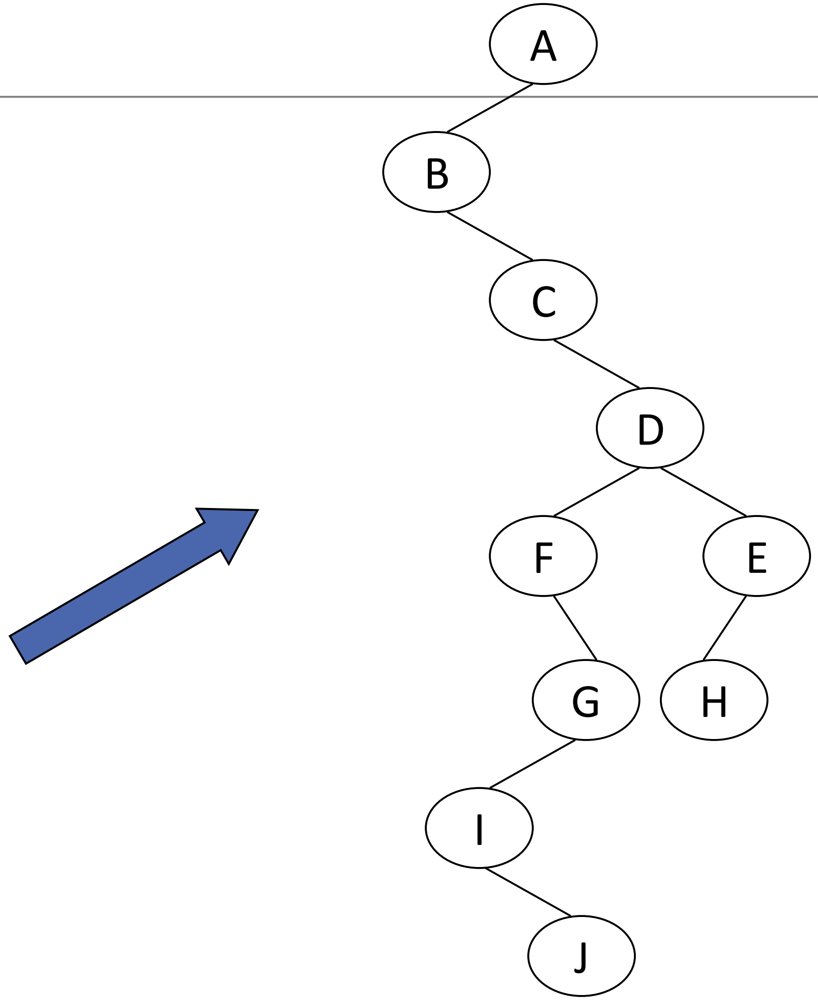


Binary Tree!

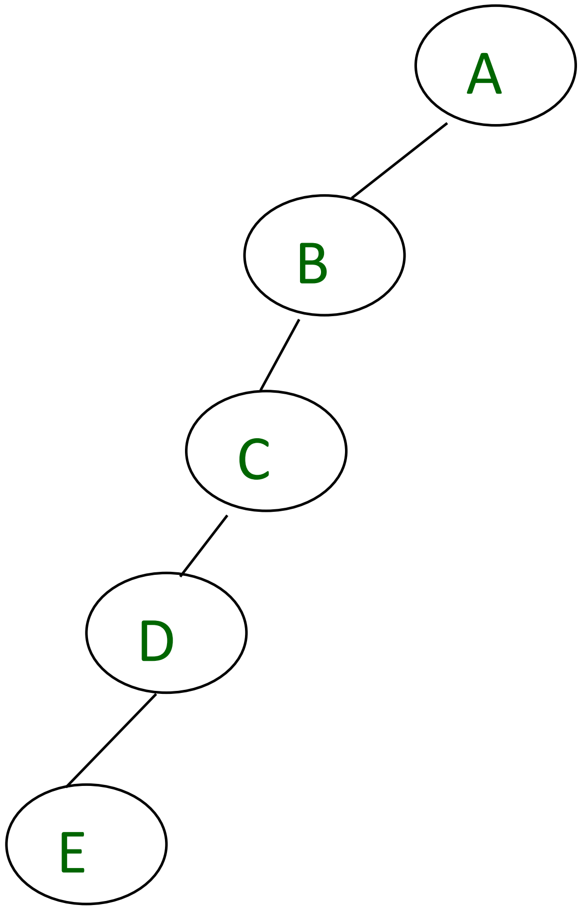
Converting to a Binary Tree

- Binary tree left child = leftmost child
- Binary tree right child= right sibling



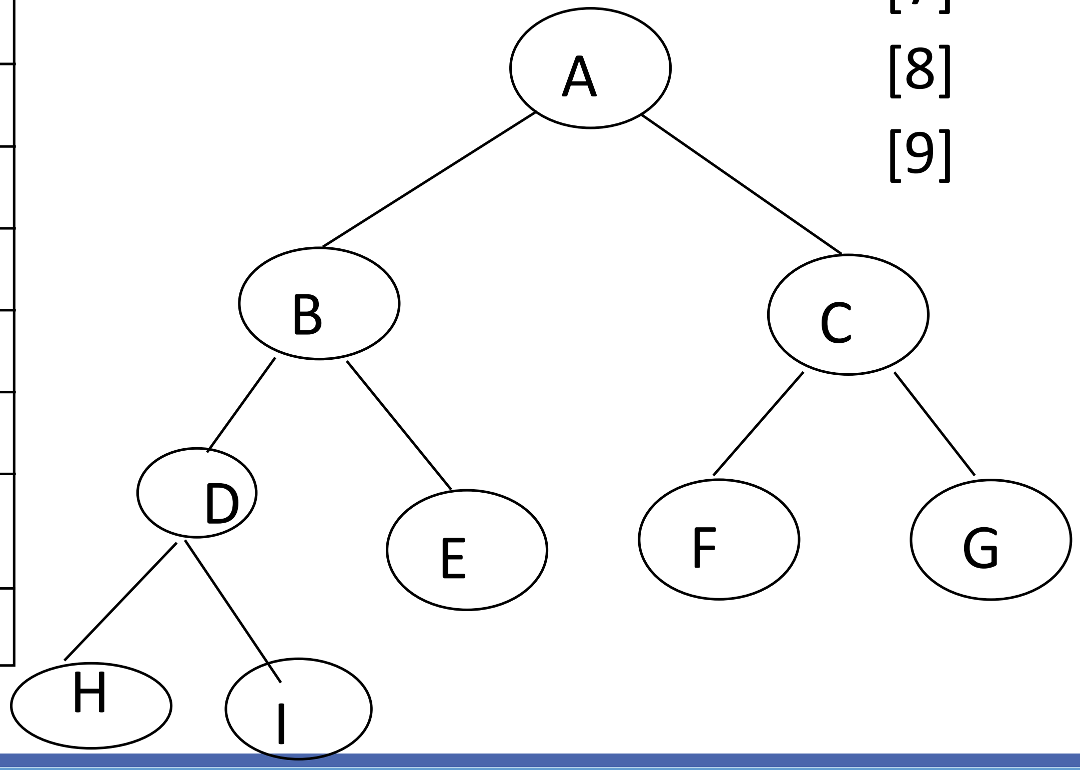


Sequential Representation



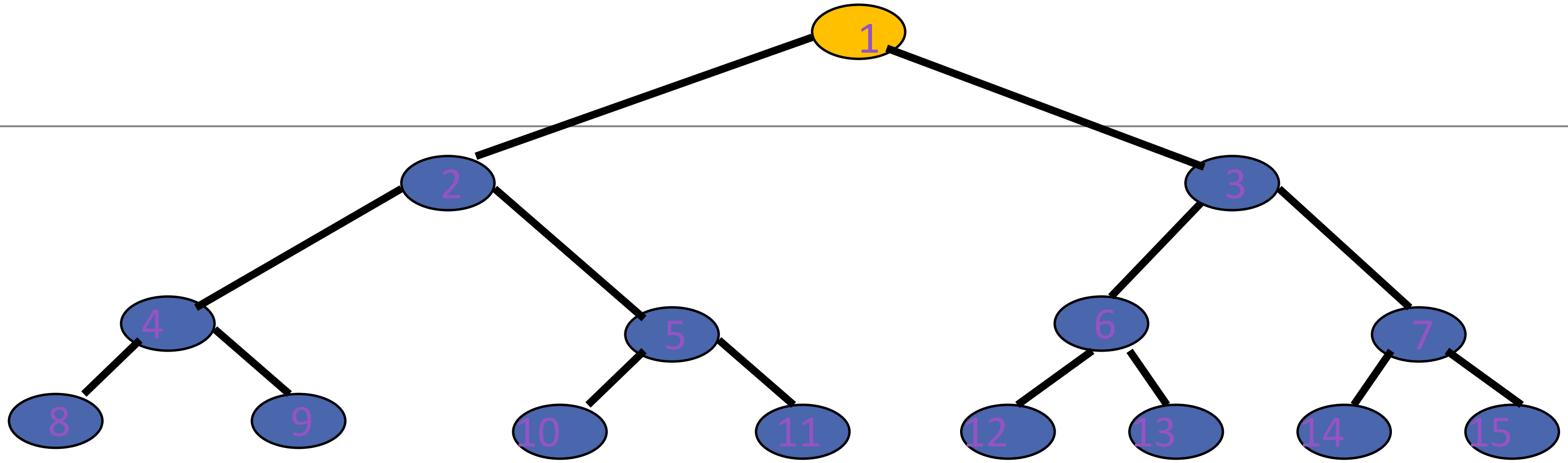
[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

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



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Node Number Properties



Parent of node i is node i/2

- But node 1 is the root and has no parent

Left child of node i is node 2i   if 2i is <=n

- But if 2i > n, node i has no left child

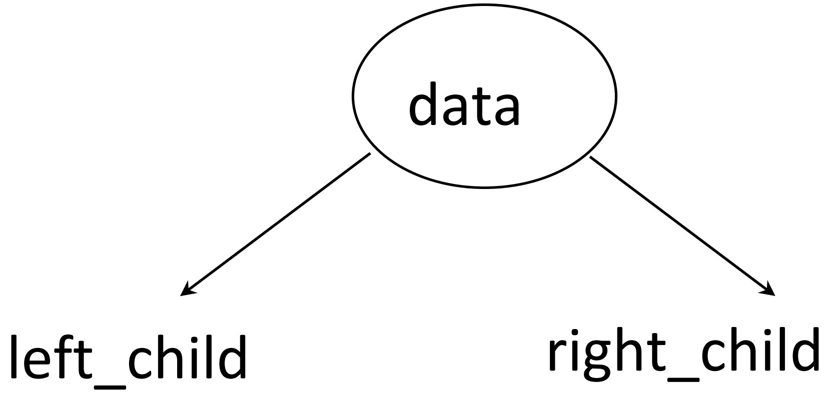
Right child of node i is node 2i+1 if 2i+1 is <=n

- But if 2i+1 > n, node i has no right child

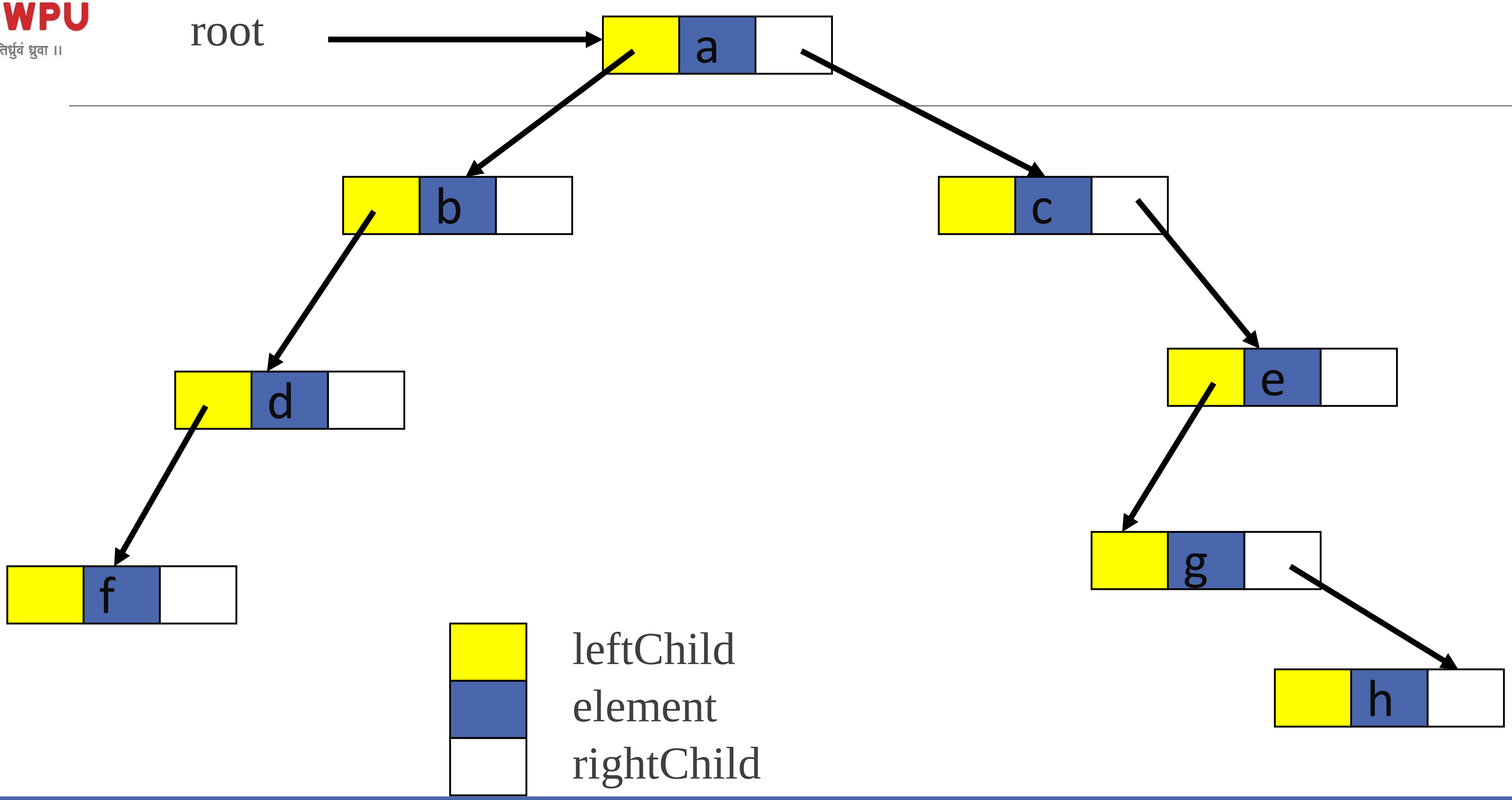
Linked Representation

```
Struct Node{
   int data;
   struct Node *lchild;
   struct Node *rchild;
};
```

left_child	data	right_child



Linked Representation Example



Binary Tree Creation

```
struct treenode
{
    char data[10];
    struct treenode *left;
    struct treenode *right;
}
```

```
int main()
{

        allocate the memory for root and read data ;
        Set Left and right of root node to NULL;
        create_r(root);
}
```

```
Algorithm create_r(struct treenode* root) {

    temp = root
    Accept choice whether data is added to left  of  temp->data;
    if ch='y'{
        Allocate a memory for curr and accept data;
         Set Left and right of curr node to NULL;
        temp->left=curr;
        create_r(curr);
    }

     Accept choice whether data is added to right  of  temp->data;
    if ch='y'   {
        Allocate a memory for curr and accept data;
        Set Left and right of curr node to NULL;
         temp->right=curr;
        create_r(curr);
    }
}
```

```
Algorithm create_nr(struct treenode* root)
{
   do
   {
     temp=root;
     flag=0;
     allocate memory for curr and accept data;
     while(flag==0)
     {
       Accept choice to add node(left or right);
       if  ch='l'
       {
          if temp->left=NULL
          {    temp->left=curr;
               flag=1;
          }
          temp=temp->left;
       }
       else {
            if ch='r'
            {
               if temp->right=NULL
               {
                    temp->right=curr;
                    flag=1;
               }
                    temp=temp->right;
            }
       }      //else end
     }//while flag
     Accept choice for continuation;
   }  // do while end
}// algo end
```

Binary Tree Traversals

- Let L, V/D 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

Binary Tree Traversals

- A traversal is where each node in a tree is visited once

- There are two very common traversals

  - Breadth First
  - Depth First

Breadth First

- In a breadth first traversal all of the nodes on a given level are visited and then all of the nodes on the next level are visited.

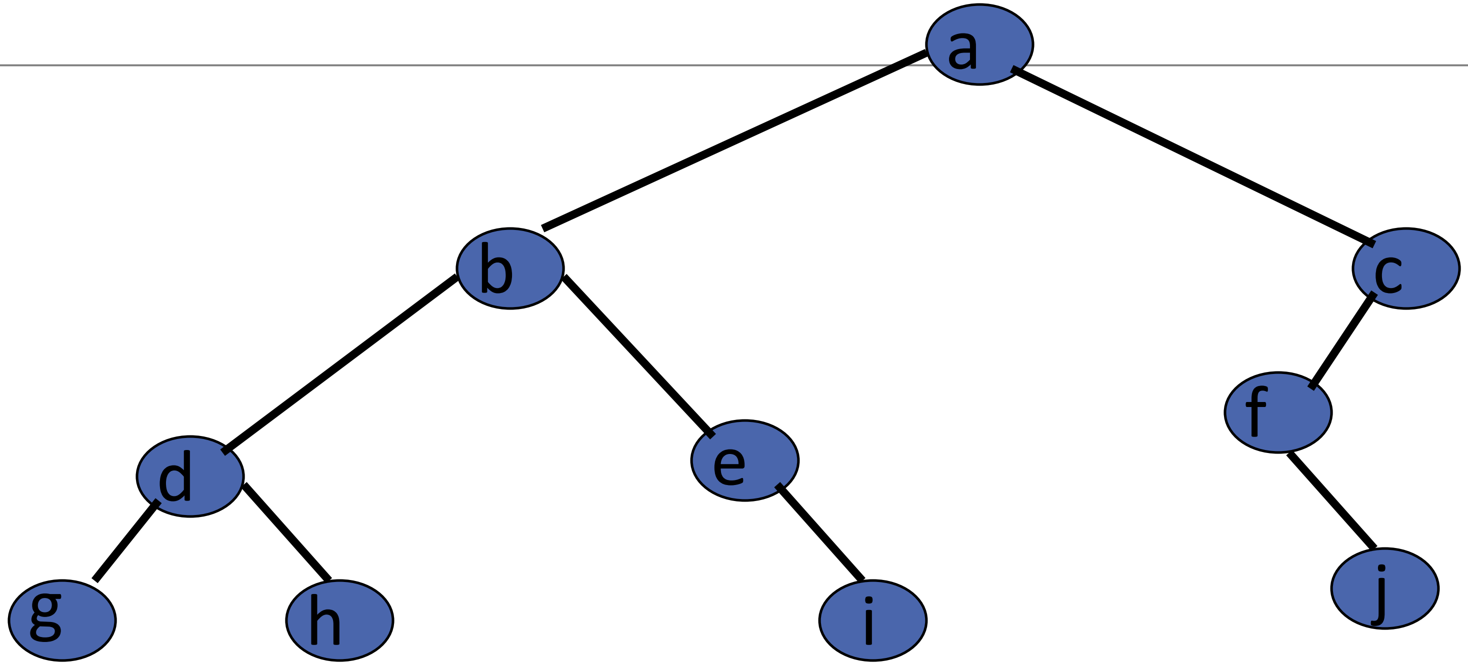
- Usually in a left to right fashion

- This is implemented with a queue

Depth First

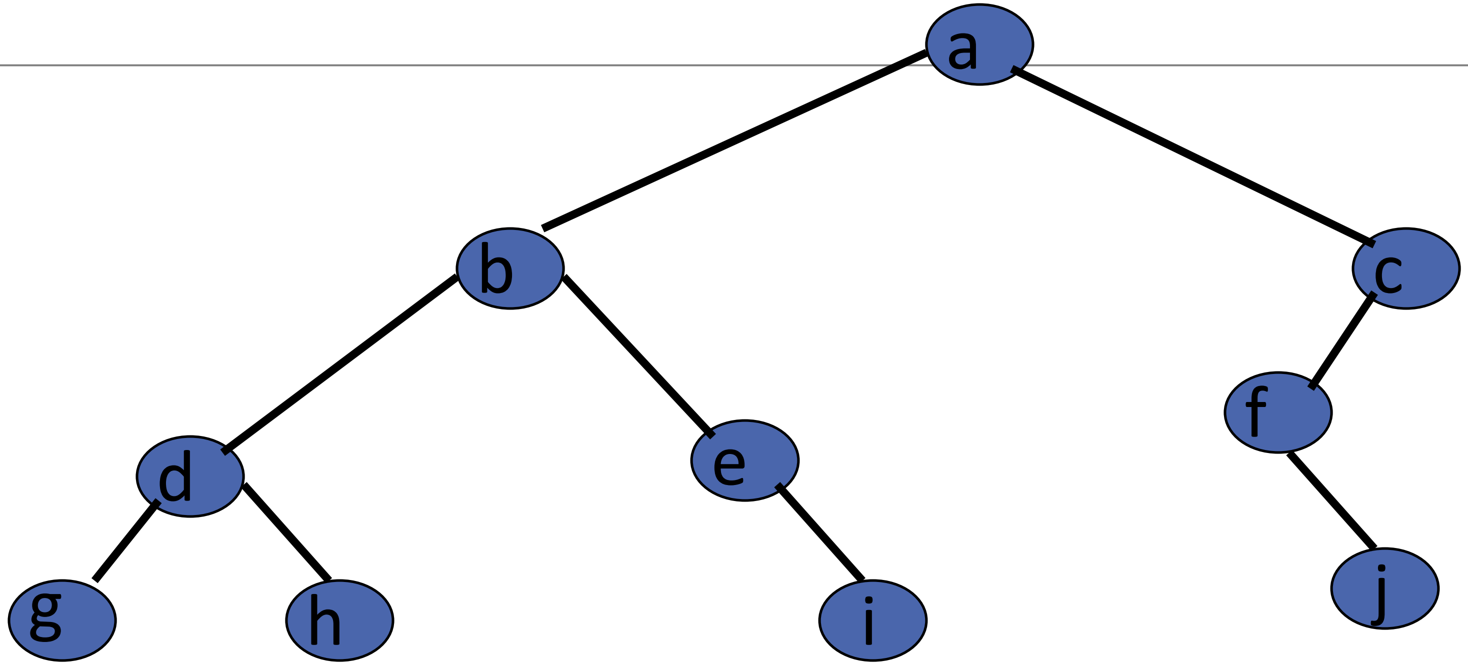
- In a depth first traversal all the nodes on a branch are visited before any others are visited
- There are three common depth first traversals
  - Inorder
  - Preorder
  - Postorder
- Each type has its use and specific application

Inorder Example (Visit = print)



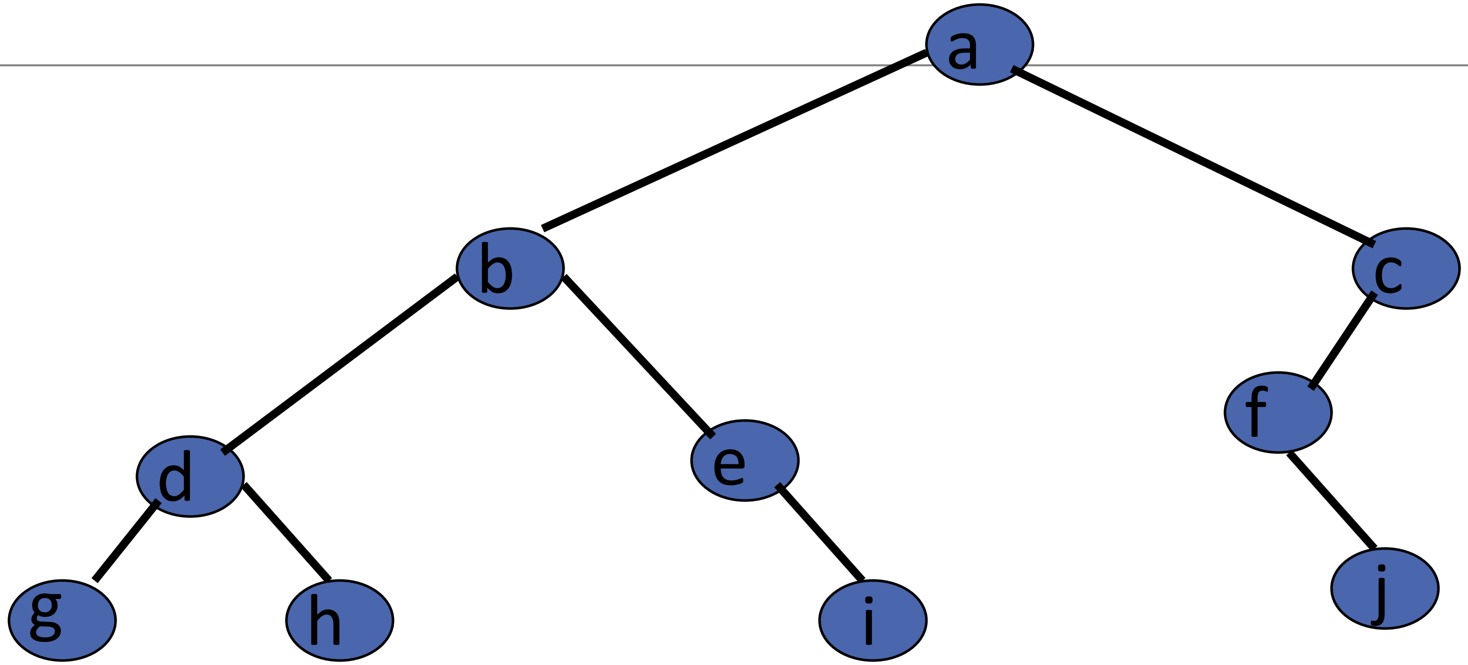
g d h b e i a f j c

Preorder Example (Visit = print)



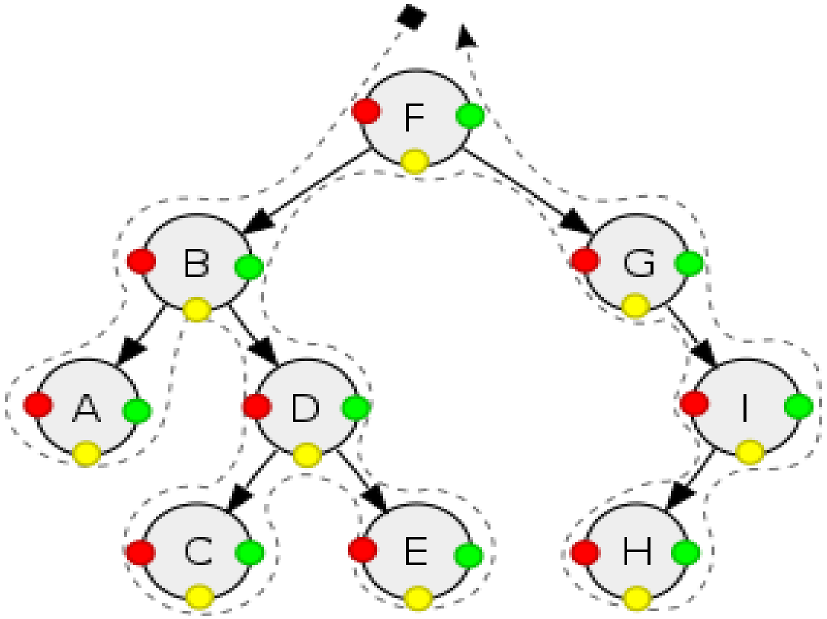
a b d g h e i c f j

Postorder Example (Visit = print)



g h d i e b j f c a

Depth first traversal

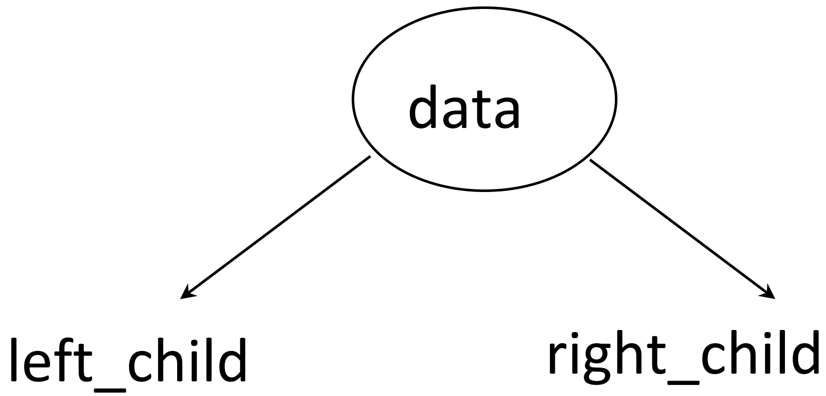


pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Linked Representation (using typedef)

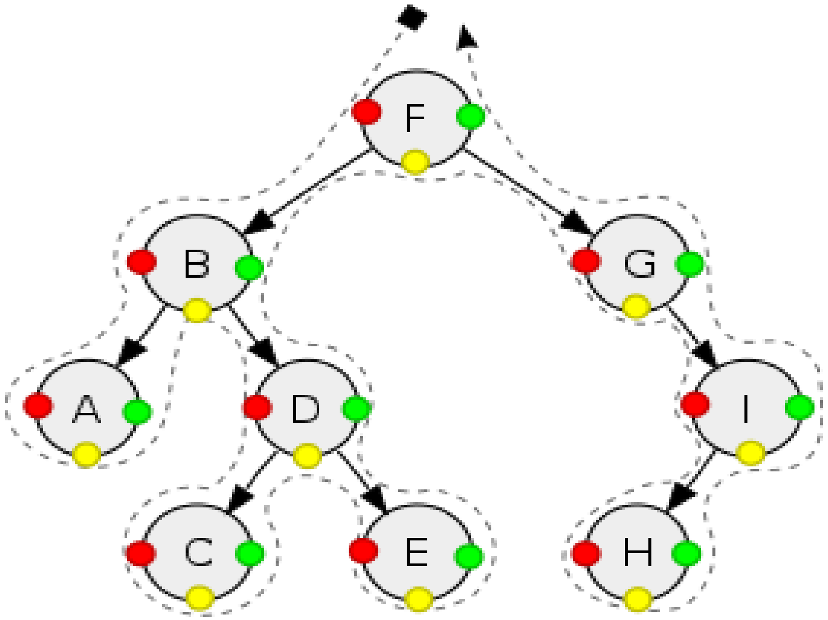
```
typedef struct TreeNode {
    char data[10];
    struct TreeNode *left_child;
    struct TreeNode *right_child;
} TreeNode;
```

left_child	data	right_child



Inorder Traversal (recursive version)

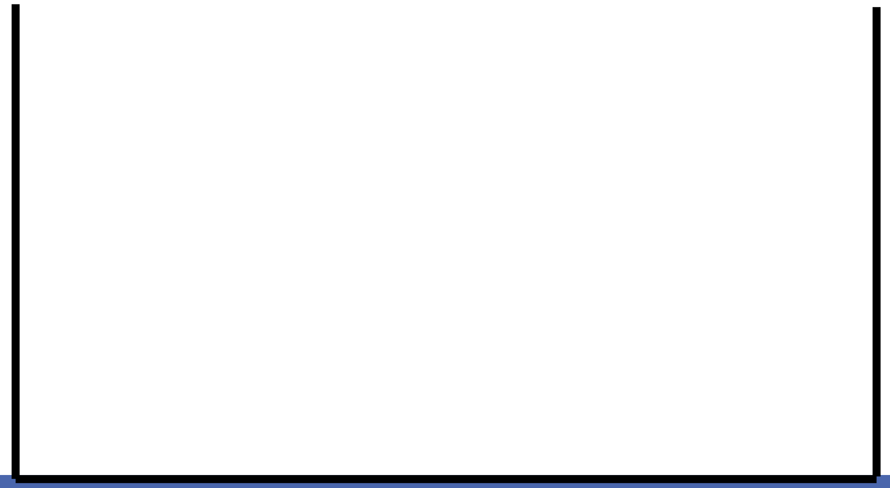
```
Algorithm inorder_r(Treenode *temp)
{
  if temp!=NULL
 {
    inorder_r(temp->left);
    Print temp->data;
    inorder_r(temp->right);
 }
}
```

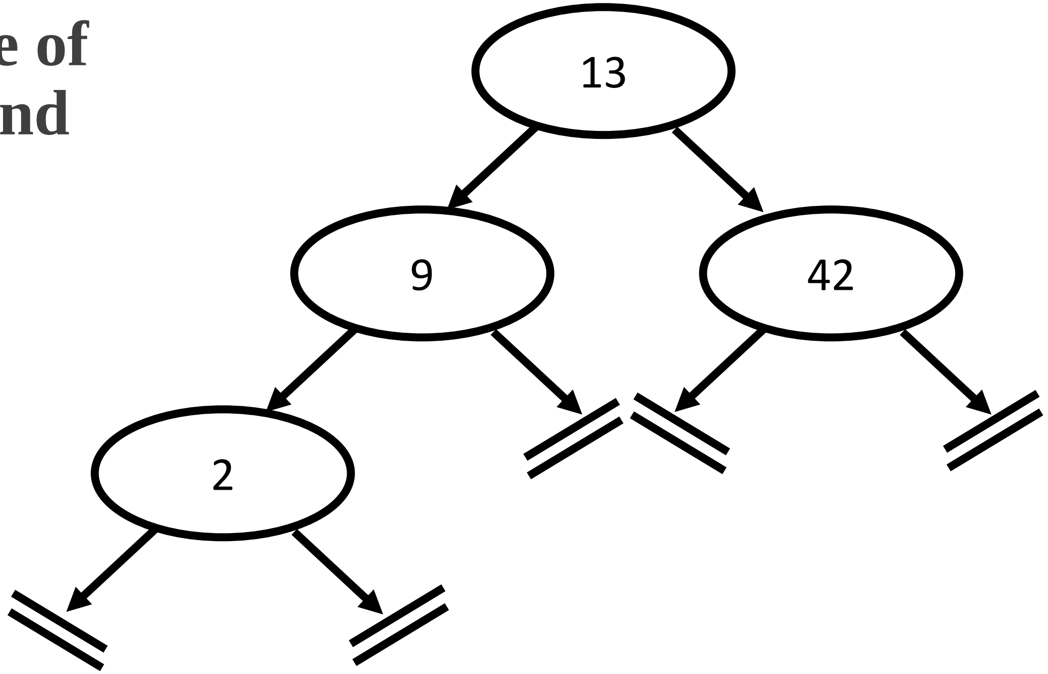


pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Use of the Activation Stack

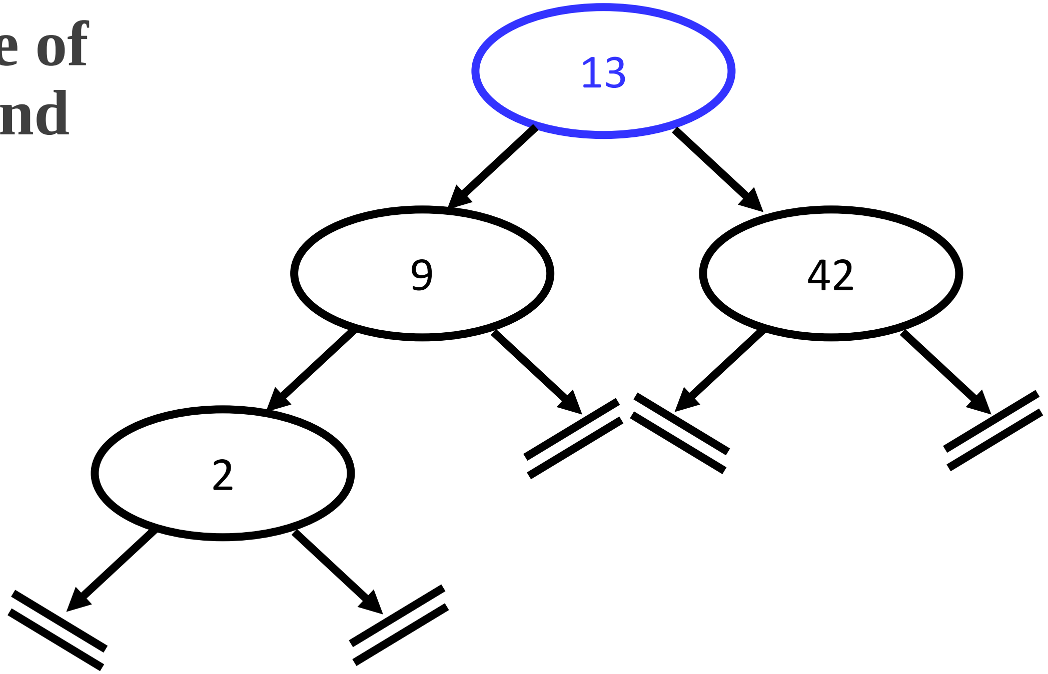
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

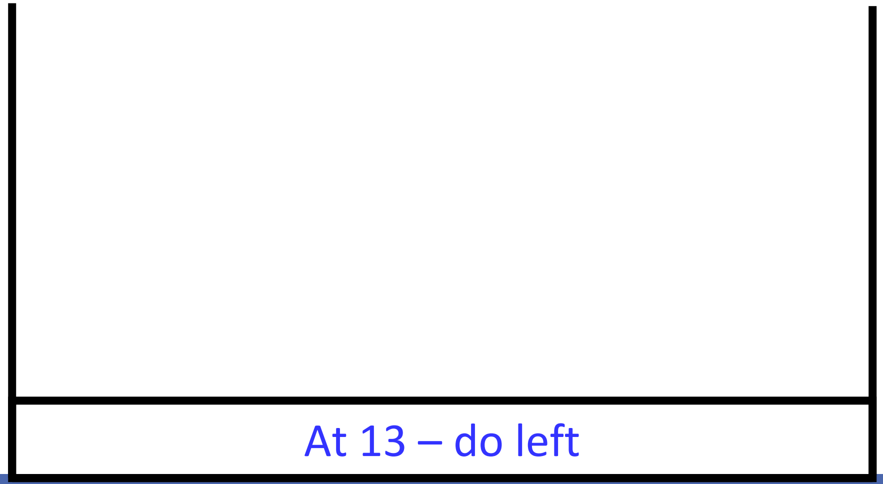




Use of the Activation Stack

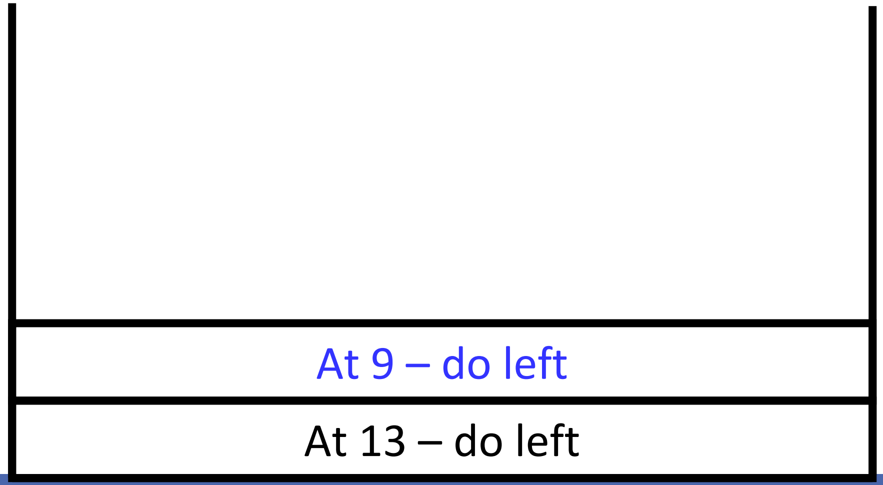
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

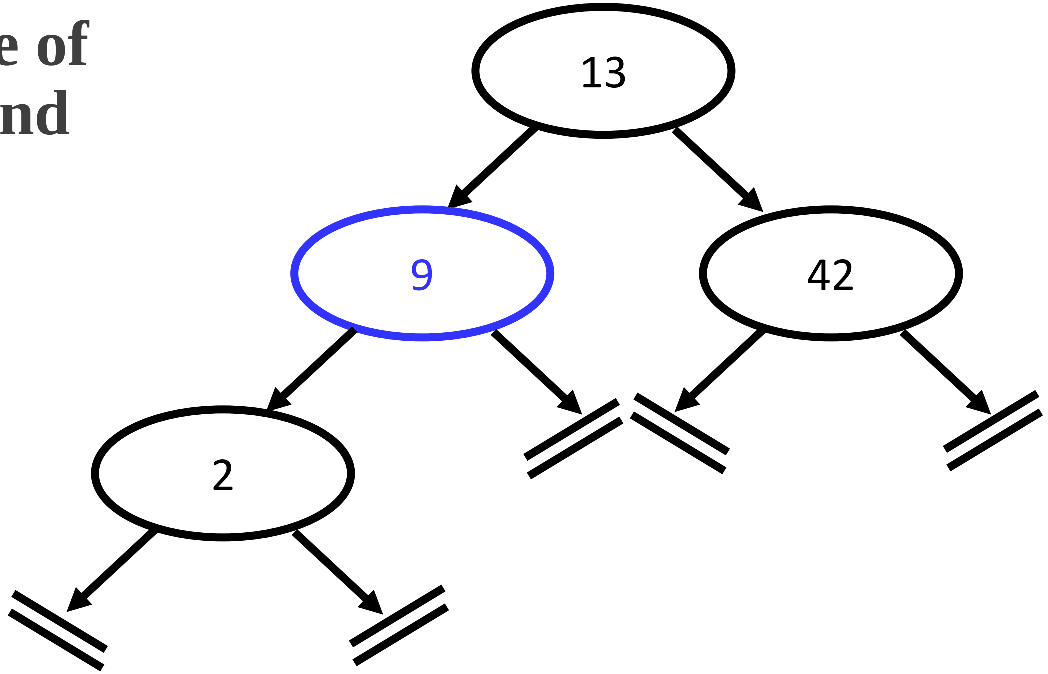




Use of the Activation Stack

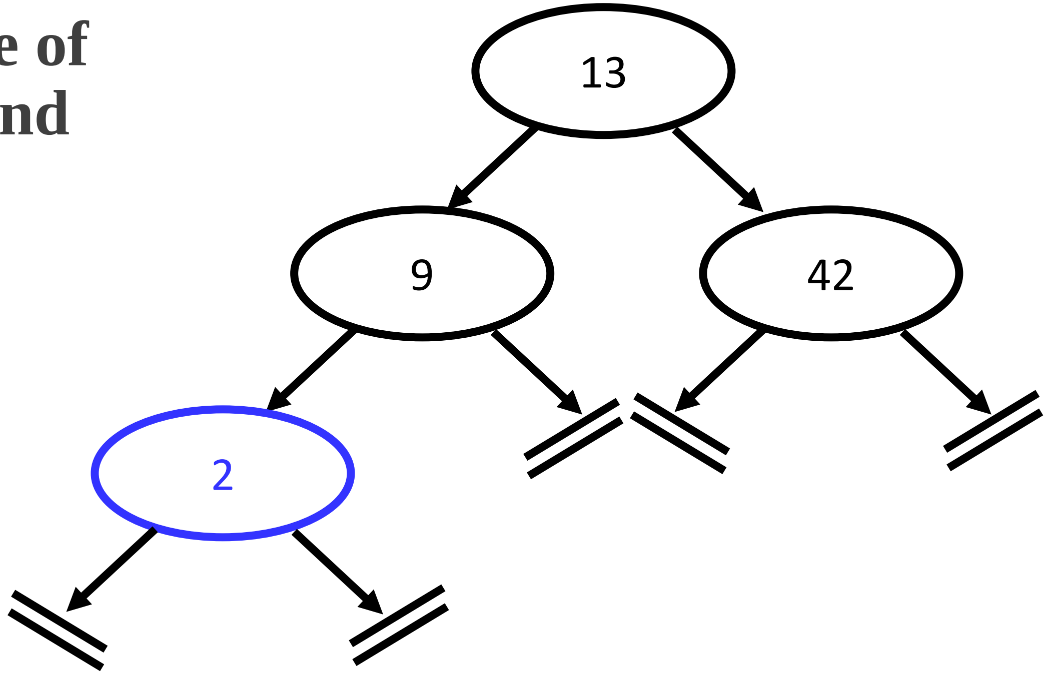
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.





Use of the Activation Stack

With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.



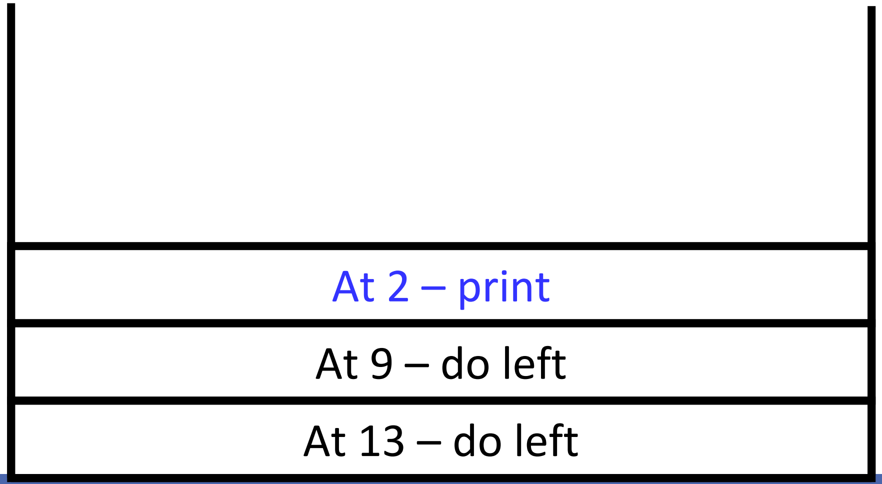
At 2 – do left

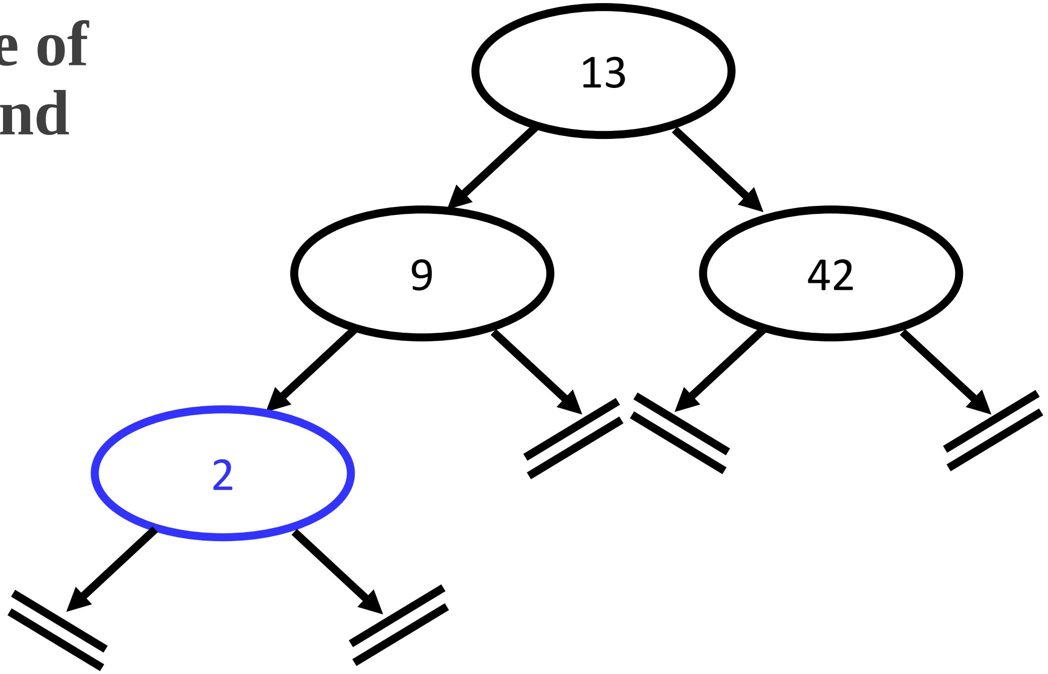
At 9 – do left

At 13 – do left

Use of the Activation Stack

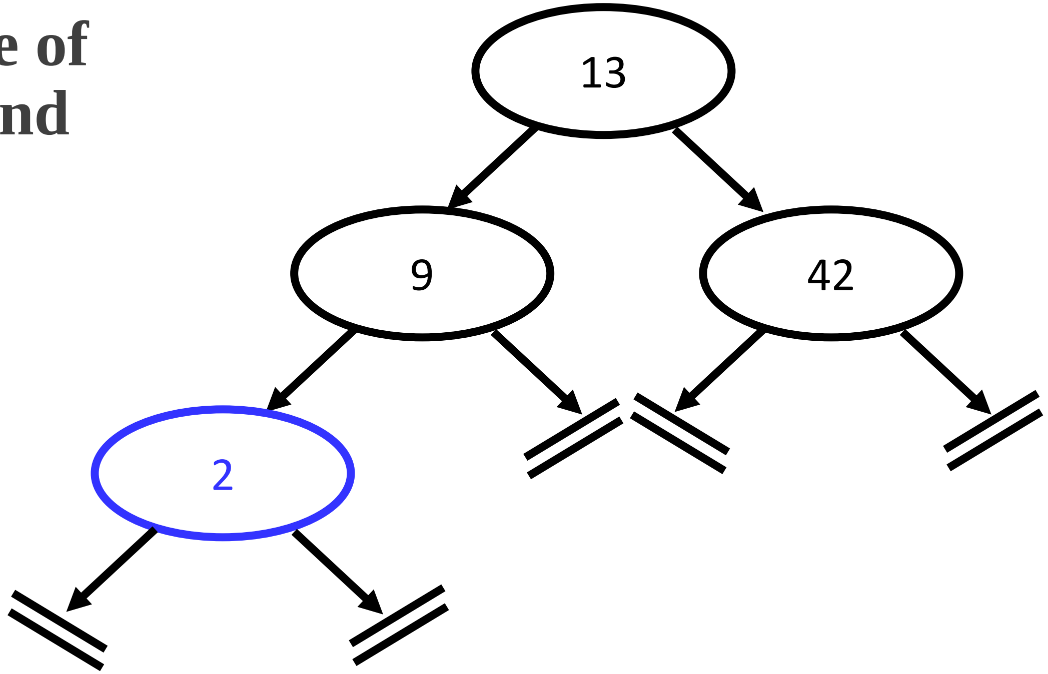
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

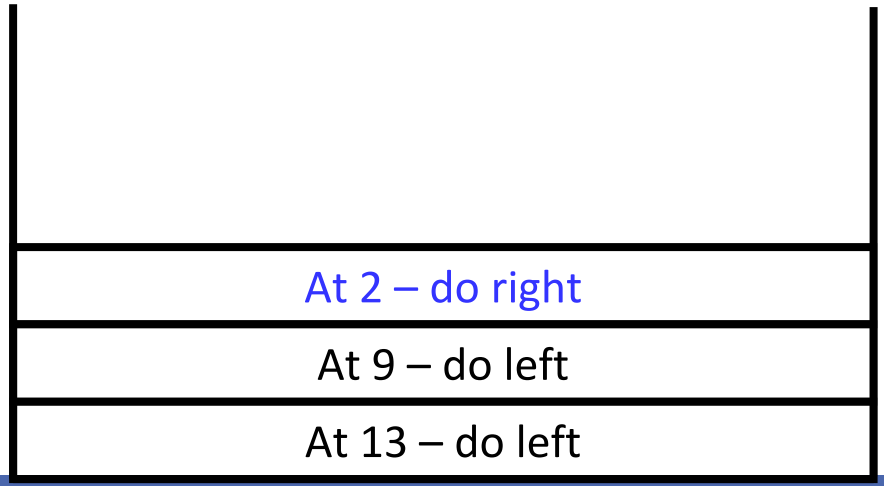




Use of the Activation Stack

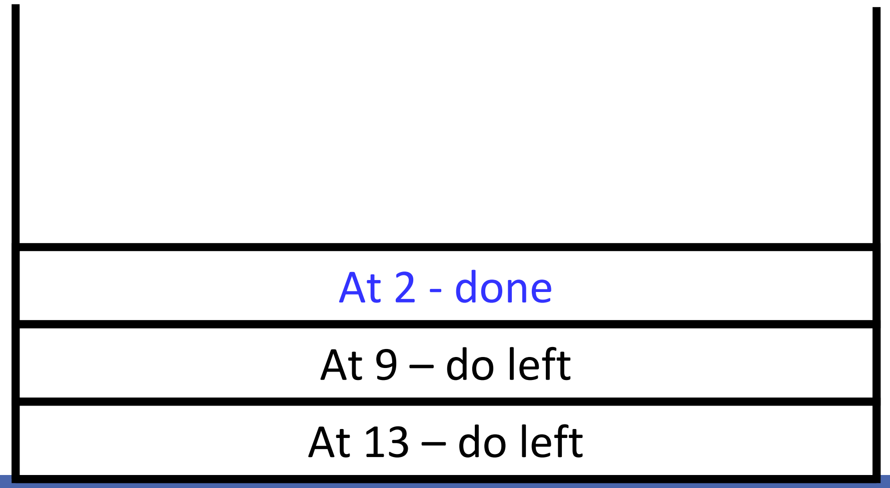
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

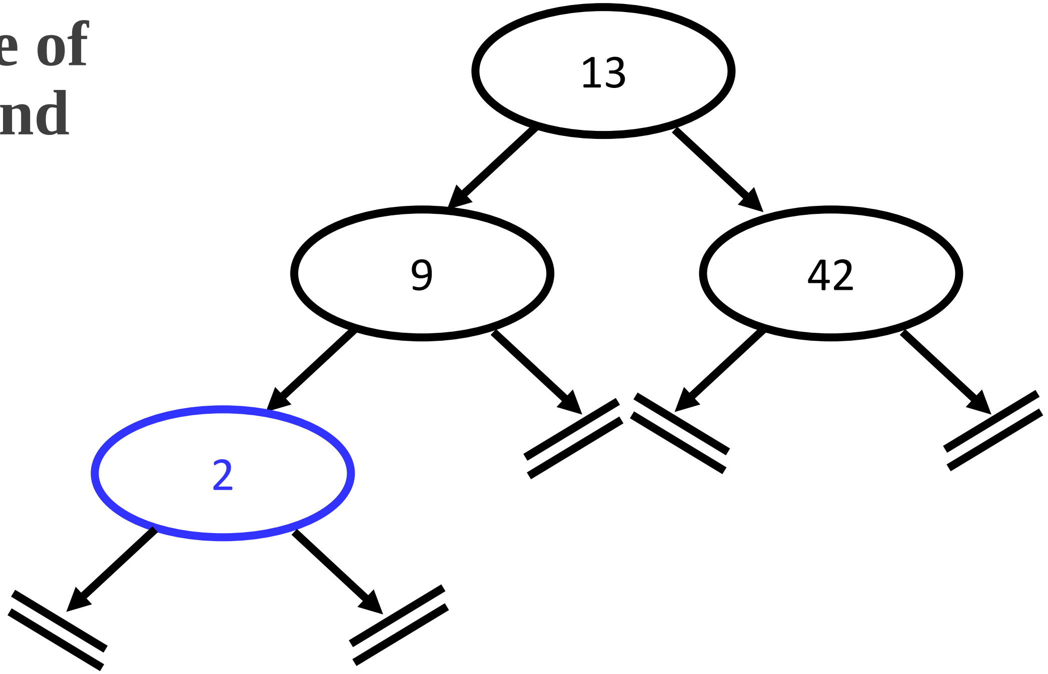




Use of the Activation Stack

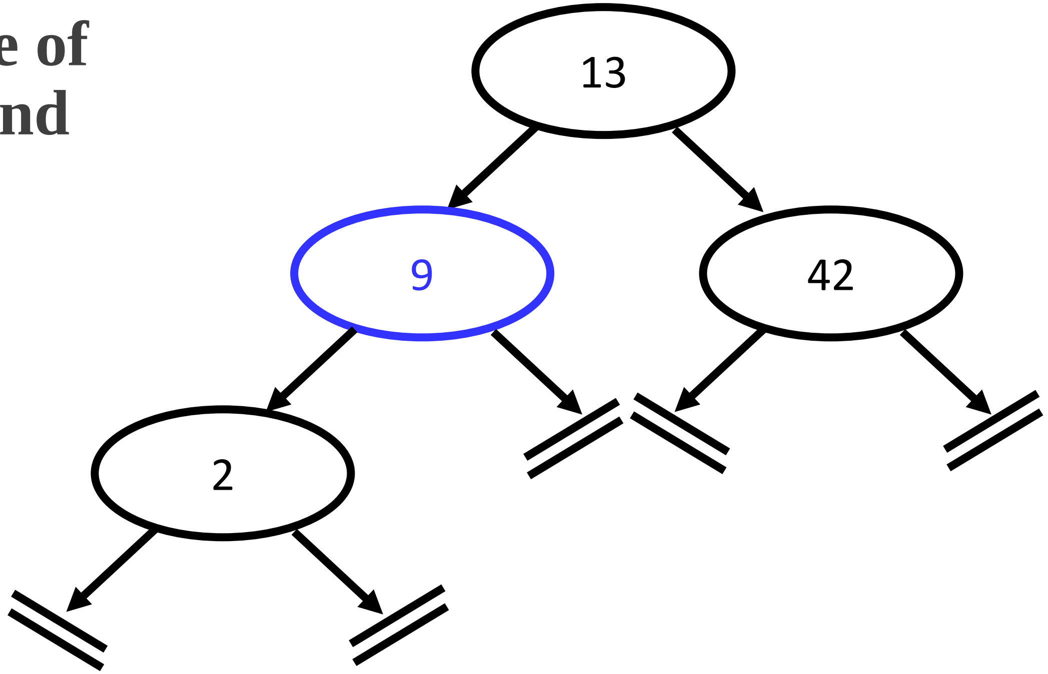
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

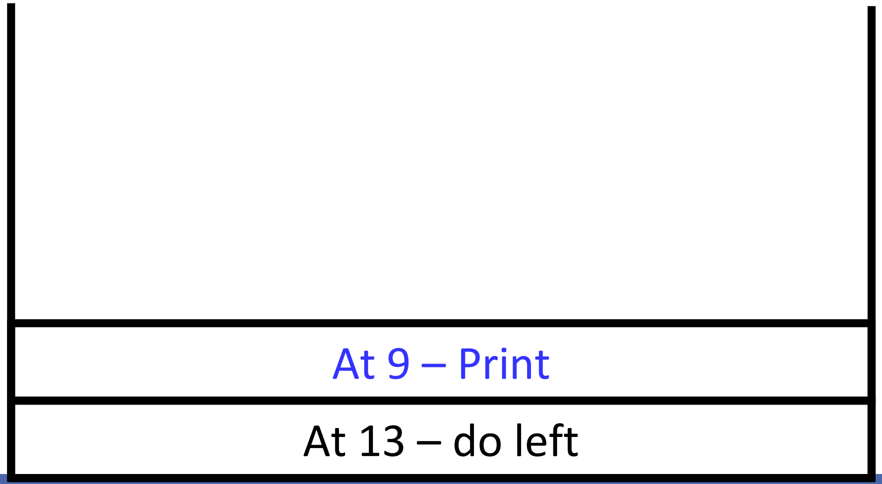




Use of the Activation Stack

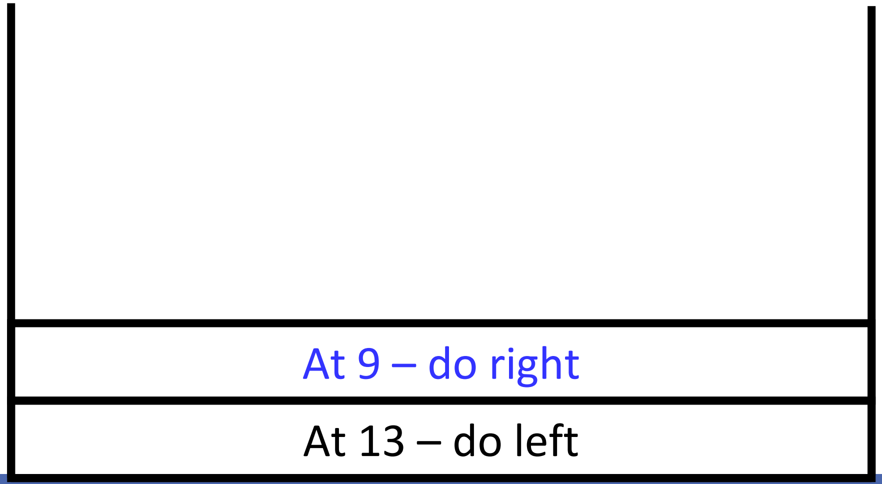
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

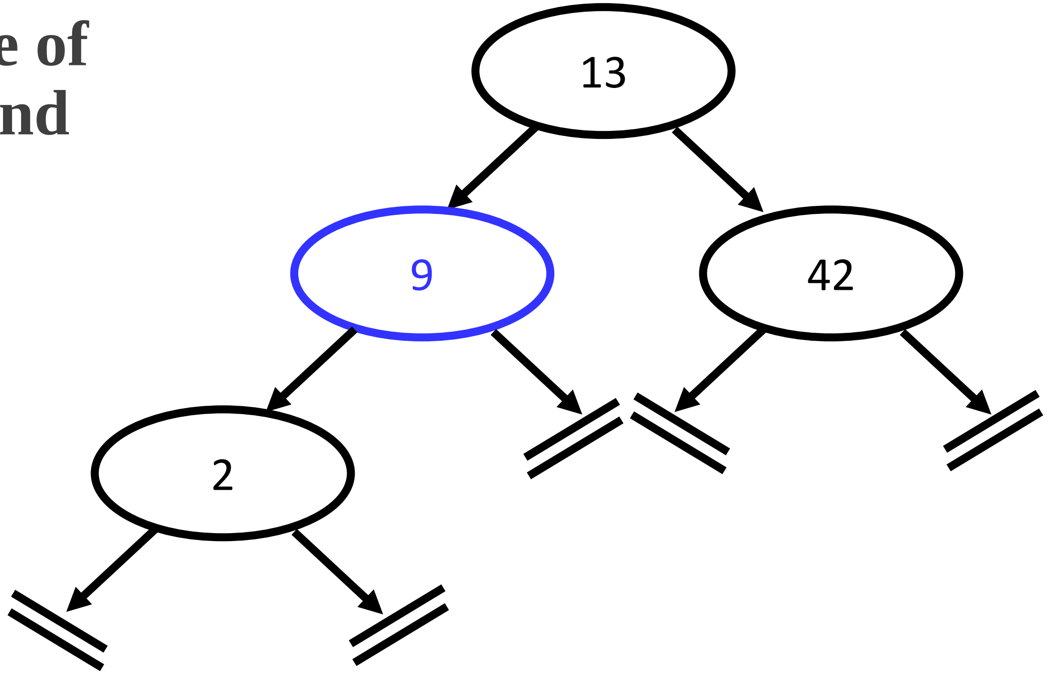




Use of the Activation Stack

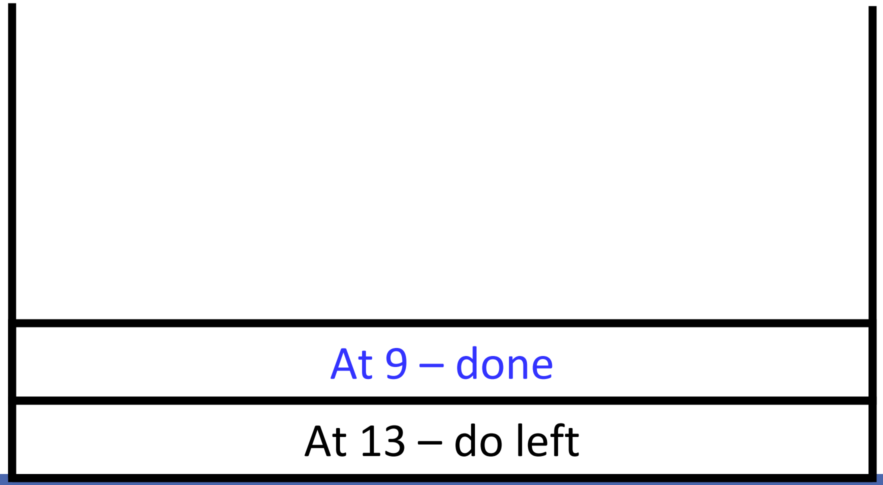
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

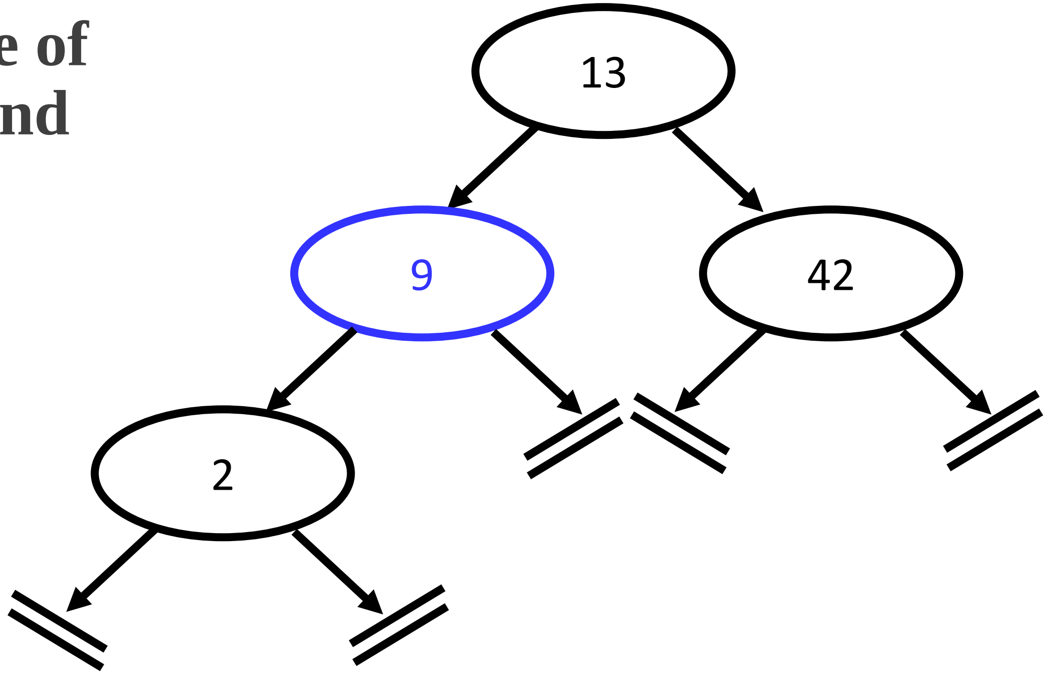




Use of the Activation Stack

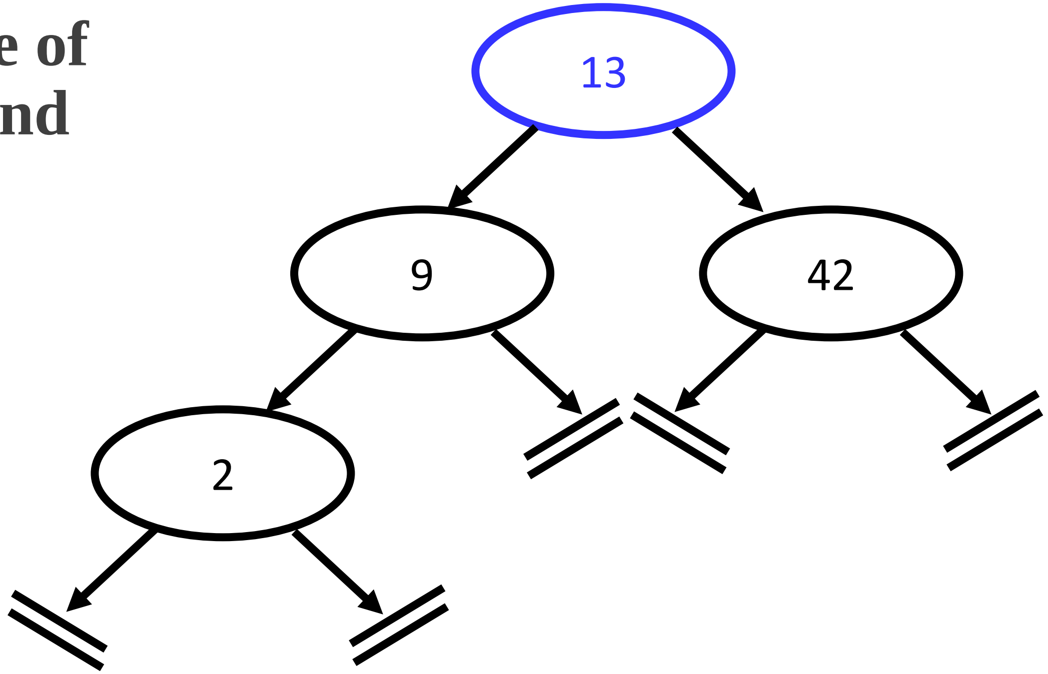
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

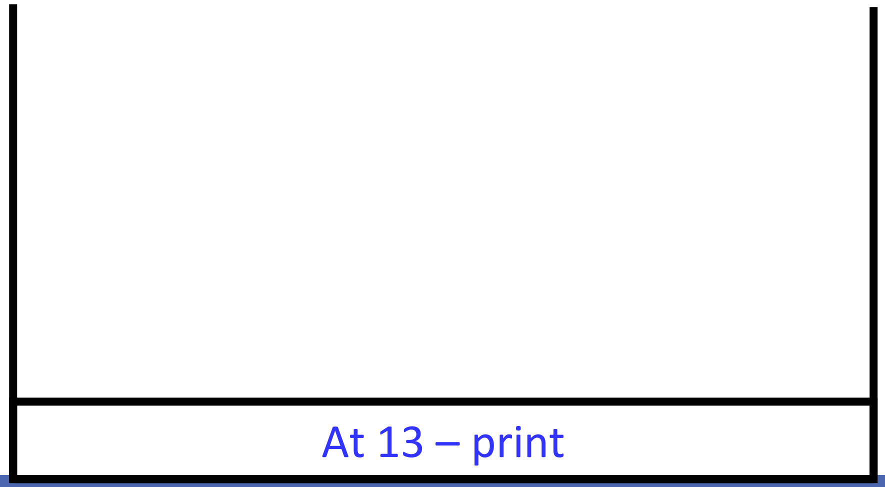




Use of the Activation Stack

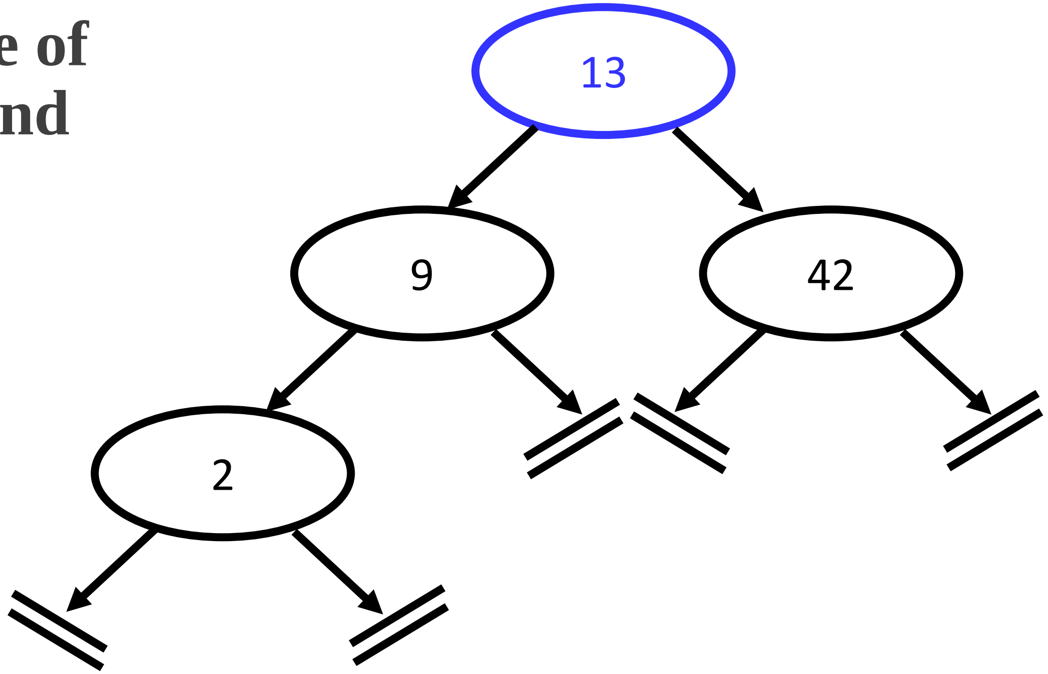
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

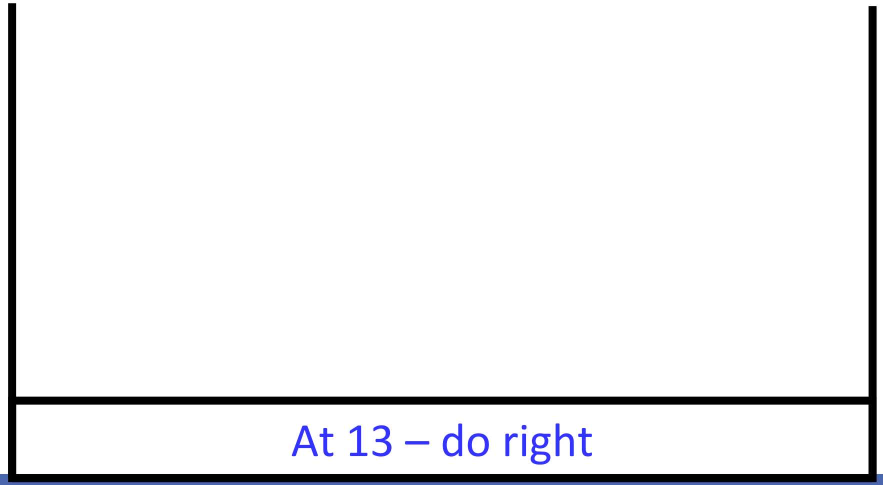




Use of the Activation Stack

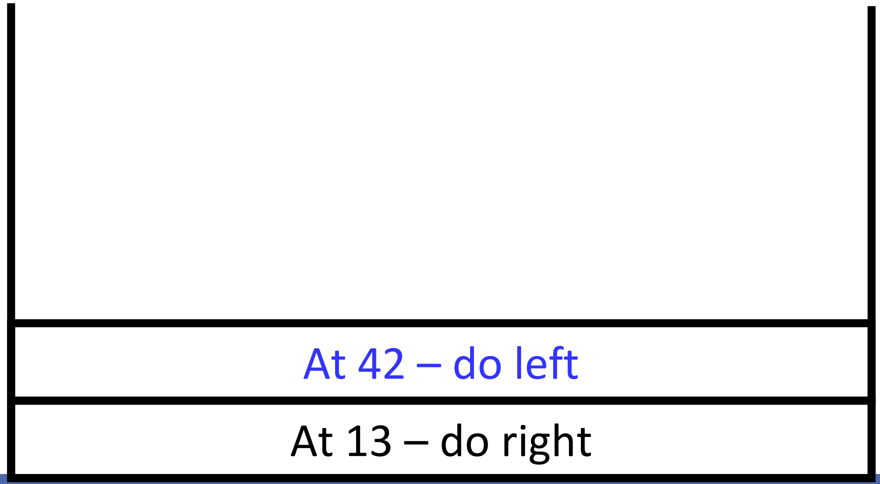
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

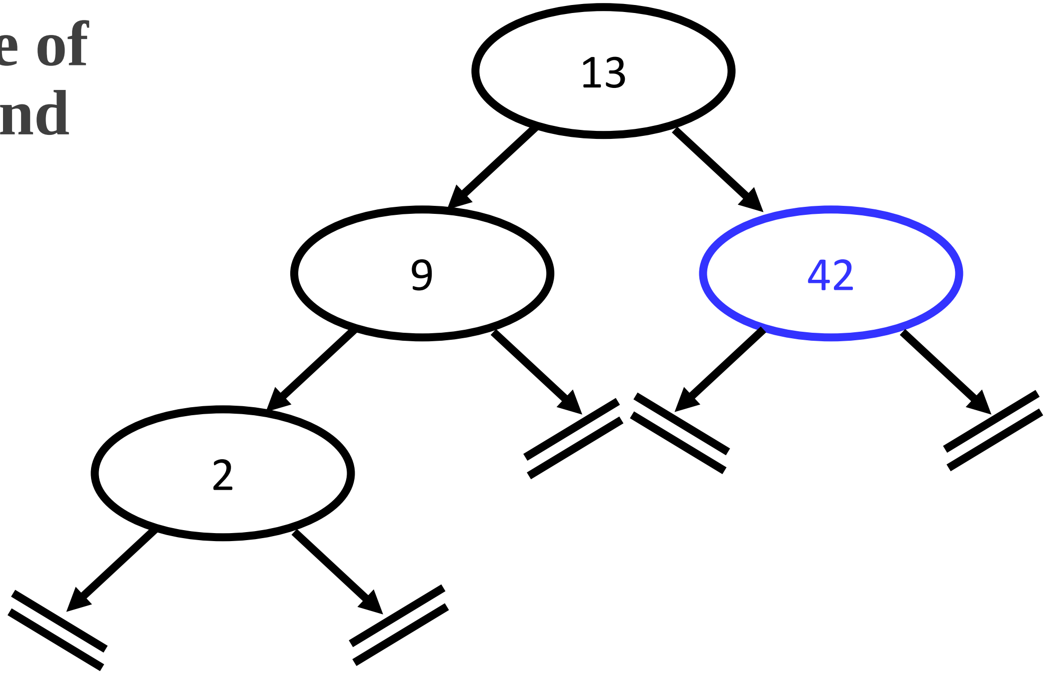




Use of the Activation Stack

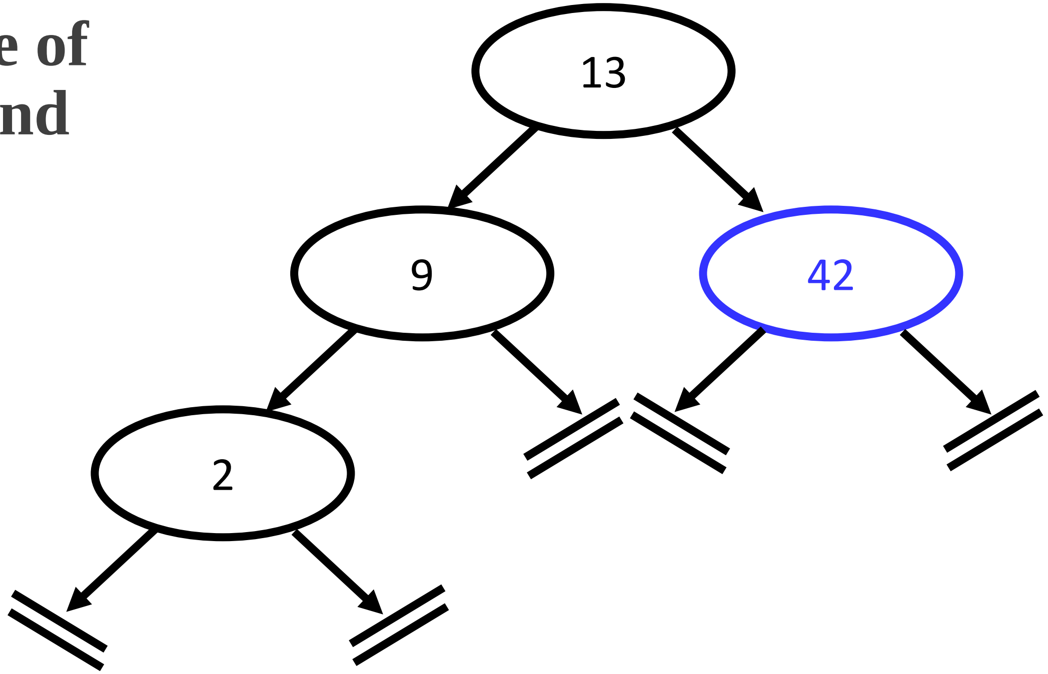
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

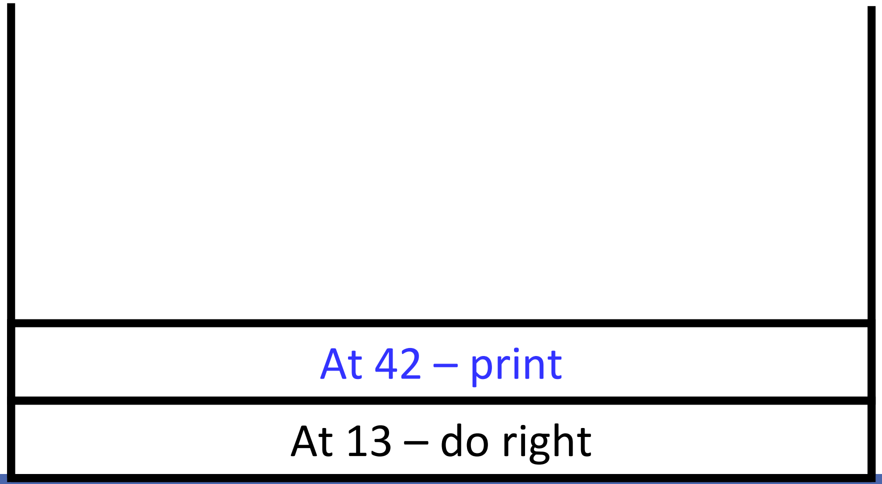




Use of the Activation Stack

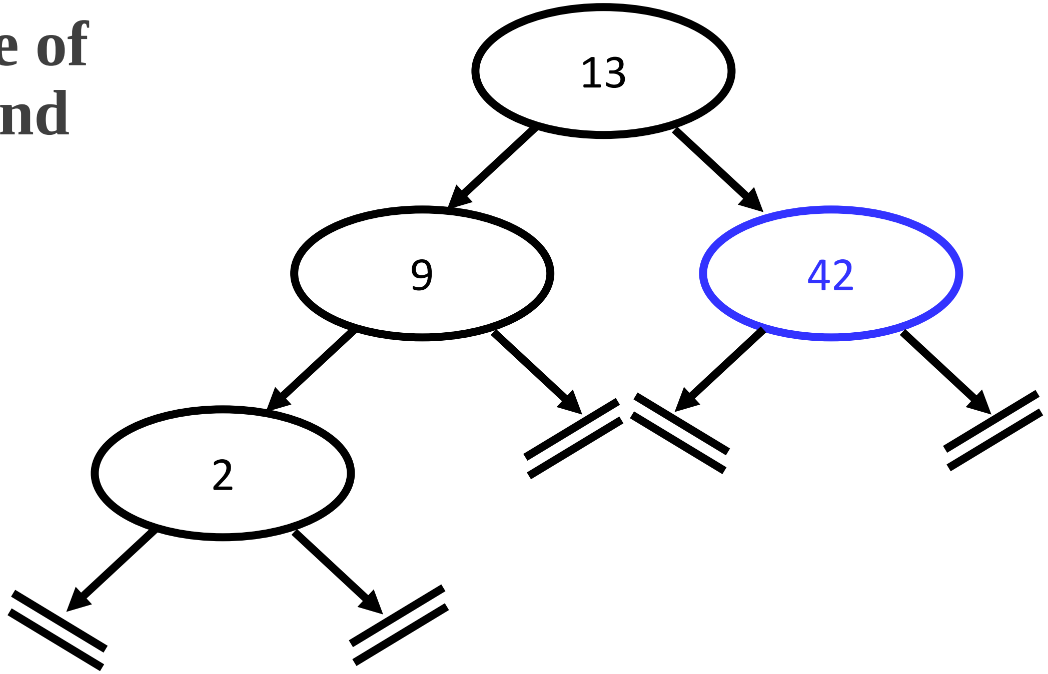
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

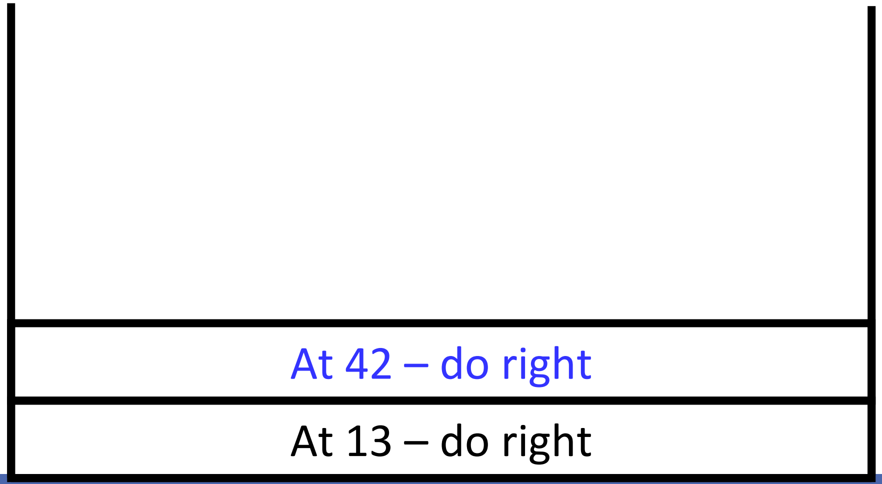




Use of the Activation Stack

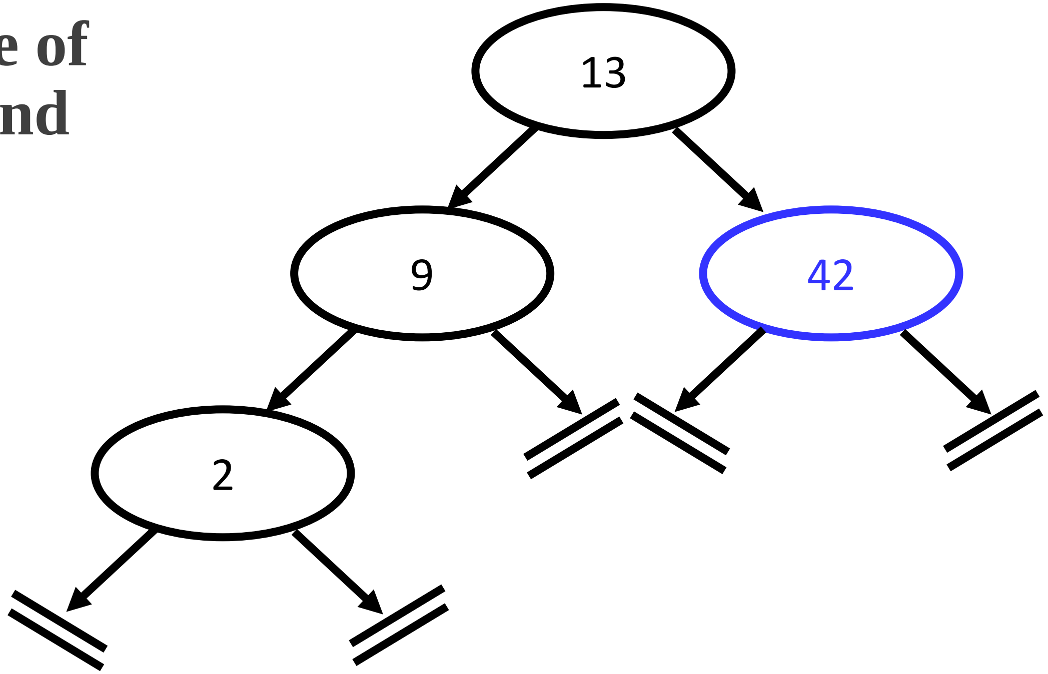
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

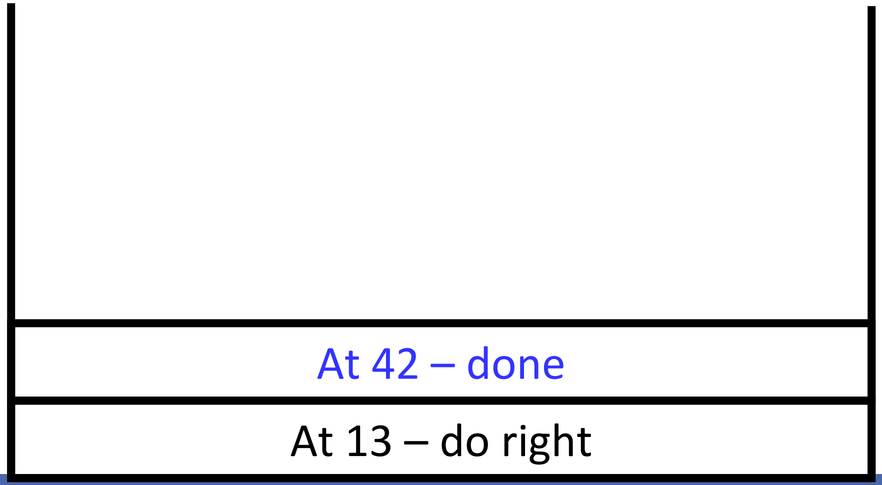




Use of the Activation Stack

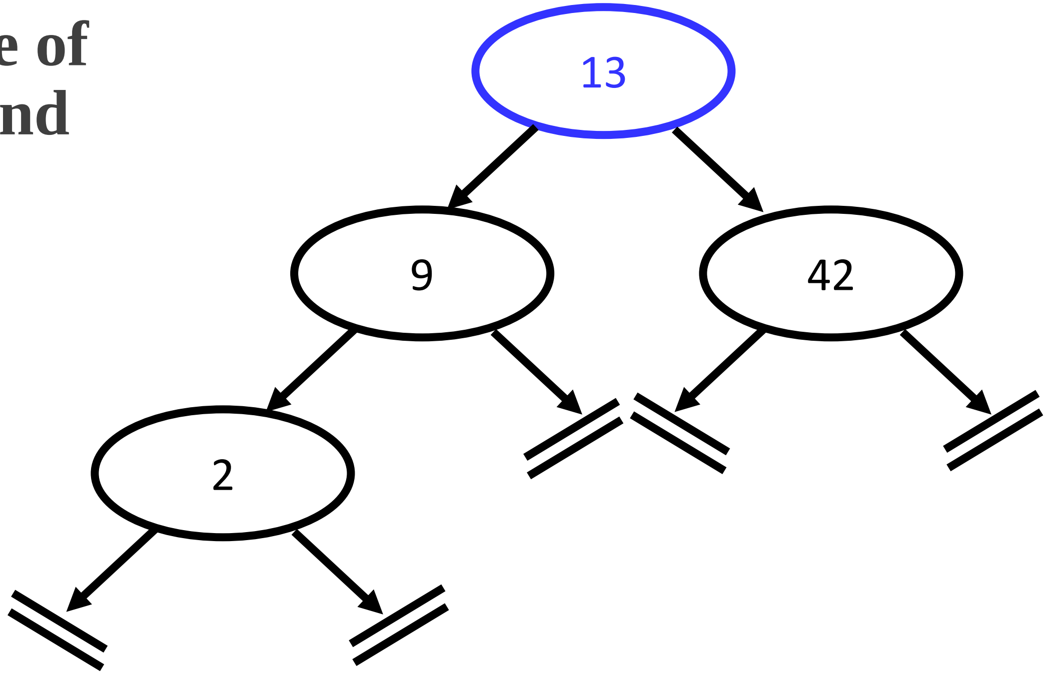
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

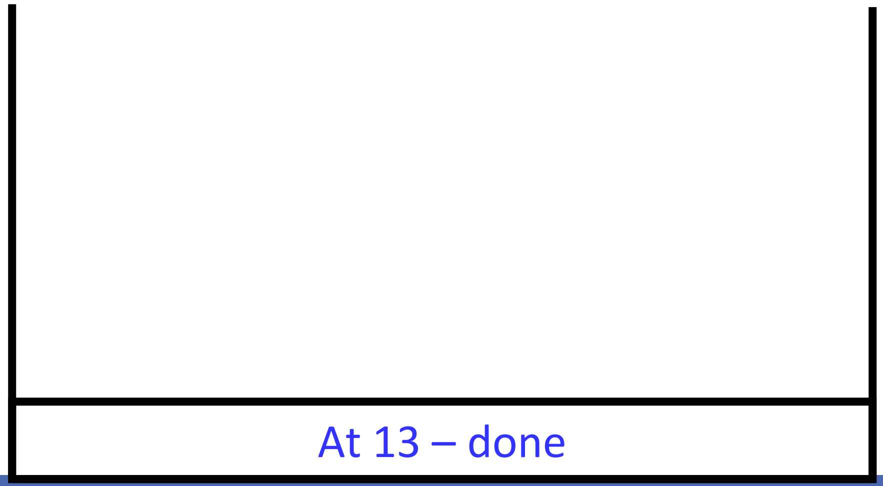




Use of the Activation Stack

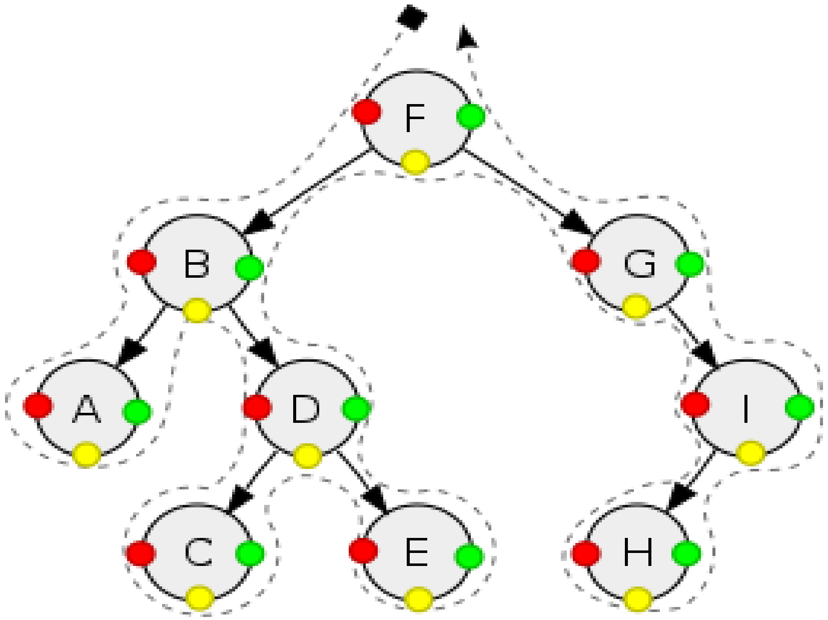
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.





Preorder Traversal (recursive version)

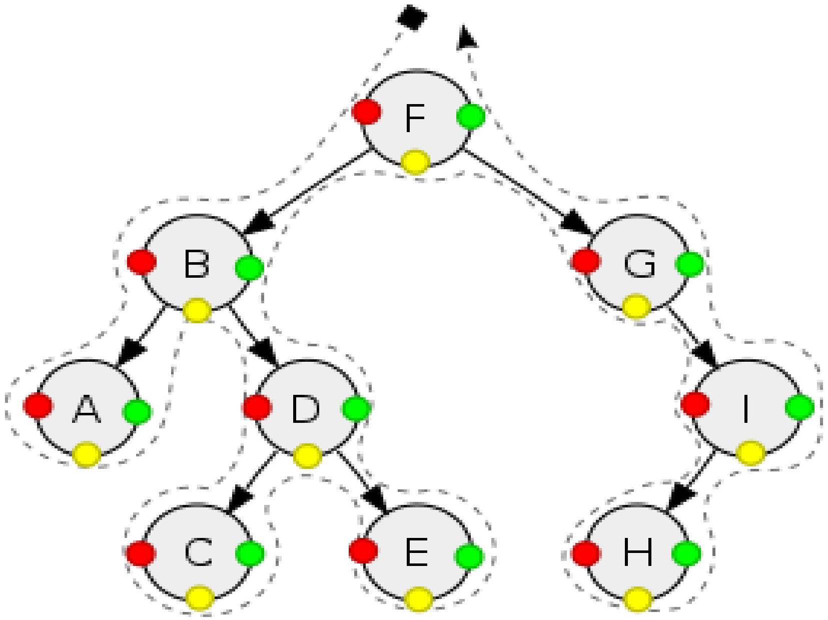
```
Algorithm preorder(Treenode * temp)
/* preorder tree traversal */
{
    if (temp!=NULL) {
        print(temp->data);
        preorder(temp->left);
        predorder(temp->right);
    }
}
```



pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Postorder Traversal (recursive version)

```
Algorithm postorder(Treenode * temp)
/* postorder tree traversal */
{
    if (temp!=NULL) {
        postorder(temp->left);
        postdorder(temp->right);
        print(temp->data);
    }
}
```



pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

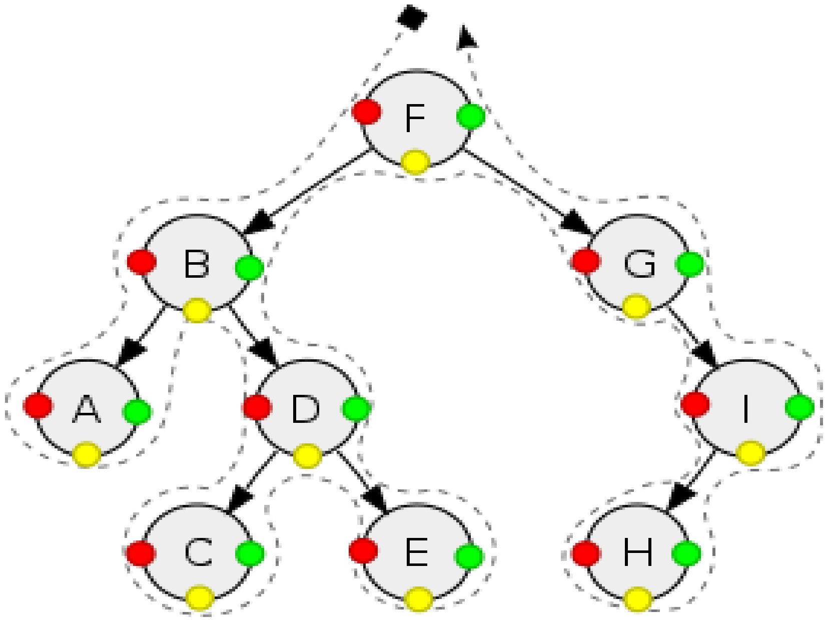
Stack for tree traversal

```
typedef struct TreeNode {
    char data[10];
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

// Global stack variables
#define STACK_SIZE 100
TreeNode* stack[STACK_SIZE]; // Array to store stack
  elements
int top = -1; // Index of the top element

// Function to check if the stack is empty
int isStackEmpty() {
    return top == -1;
}
```

```
void push(TreeNode* node) {
    if (top == STACK_SIZE - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = node;
}
TreeNode* pop() {
    if (isStackEmpty()) {
        printf("Stack underflow\n");
        return NULL;
    }
   TreenNode *node = stack[top--];
    return node;
}
```

Nonrecursive Inorder Traversal

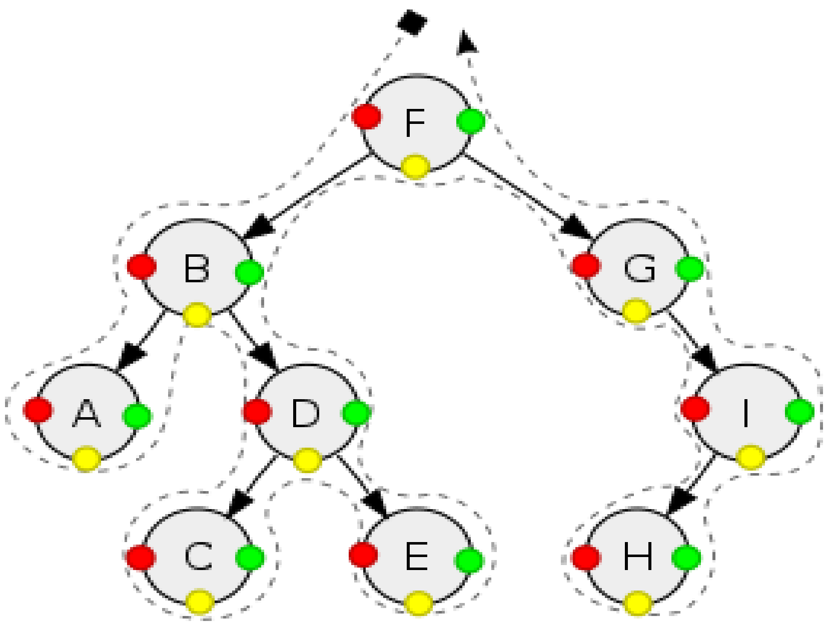
```
Algorithm inorder_nonrec(TreeNode* root) {
    temp = root;  //start traversing the binary tree at the root node
    while(1)  {
      while(temp is not NULL)
      {
        push temp onto stack;
        temp = temp ->left;
      }
      if stack  empty
        break;
      pop stack into temp;
      visit temp;    //visit the node
      temp = temp ->right;     //move to the right child
    }   //end while
}   //end algorithm
```



pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Nonrecursive Preorder Traversal

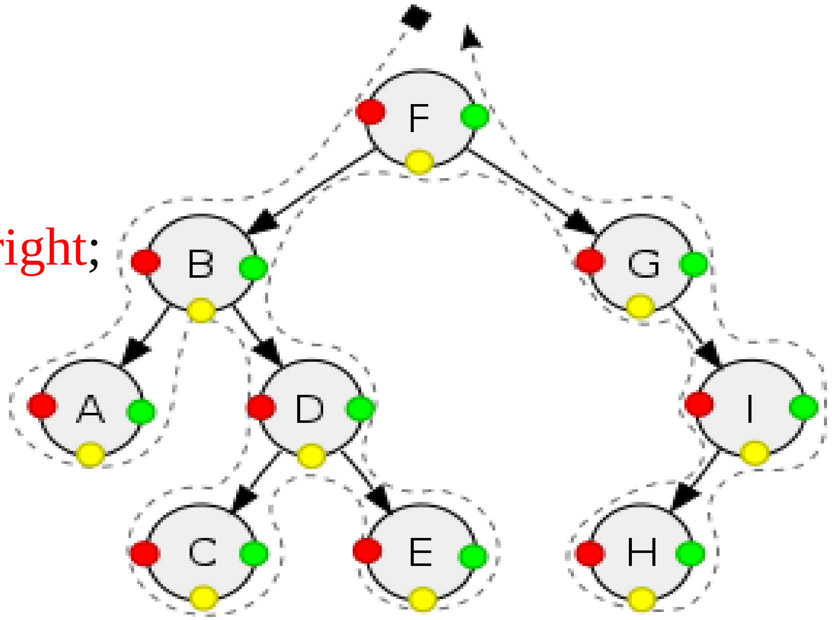
```
Algorithm preorder_nonrec(TreeNode* root) {
        temp = root;  //start the traversal at the root node
        while(1) {
          while(temp is not NULL)
          {
              visit temp;
              push temp onto stack;
            temp = temp ->left;
           }
          if stack  empty
             break;
         pop stack into temp;
         temp = temp ->right;    //visit the right subtree
      }  //end while
  }     //end algorithm
```



pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

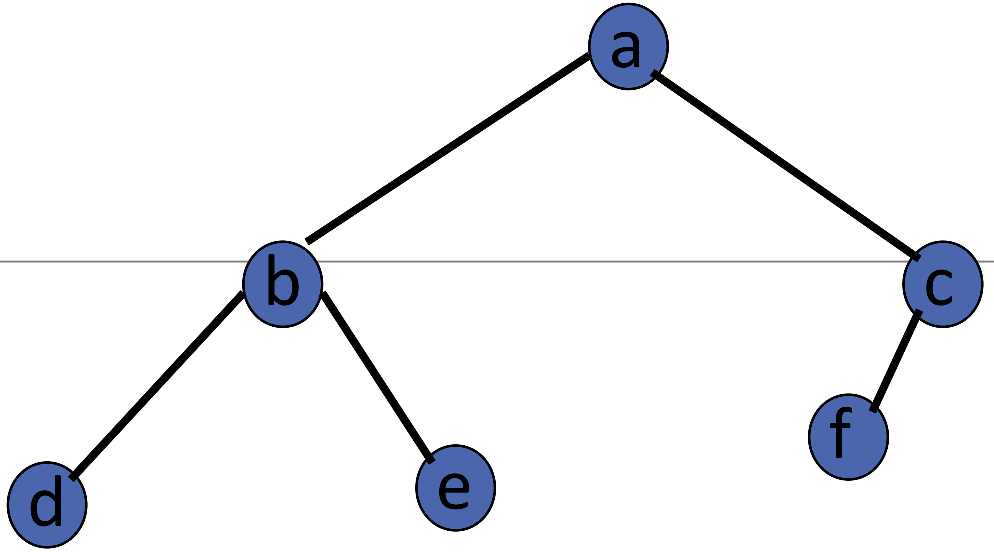
Nonrecursive Postorder Traversal

```
Algorithm postorder_nonrec(TreeNode* root)
{
   temp=root;
   while(1)
  {
     while(temp is not NULL)
    {
     push temp onto stack;
     temp = temp ->left;
    }
     if  stack top right is NULL
    {
      pop stack into temp;
      visit temp;
     }
     while(stack not empty && stack top right is temp)
     {
        pop stack into temp;
        visit temp
     }
      if stack  empty
          break;

          temp= stack[top]->right;
   }    // end while

}   // end algorithm
```



pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while Queue not empty
 {

     Remove from queue into temp;
      visit temp;
    if(temp->left is not NULL)
          insert temp->left into queue;
    if(temp->right is not NULL)
       insert temp->right into queue;

   }
 }
```

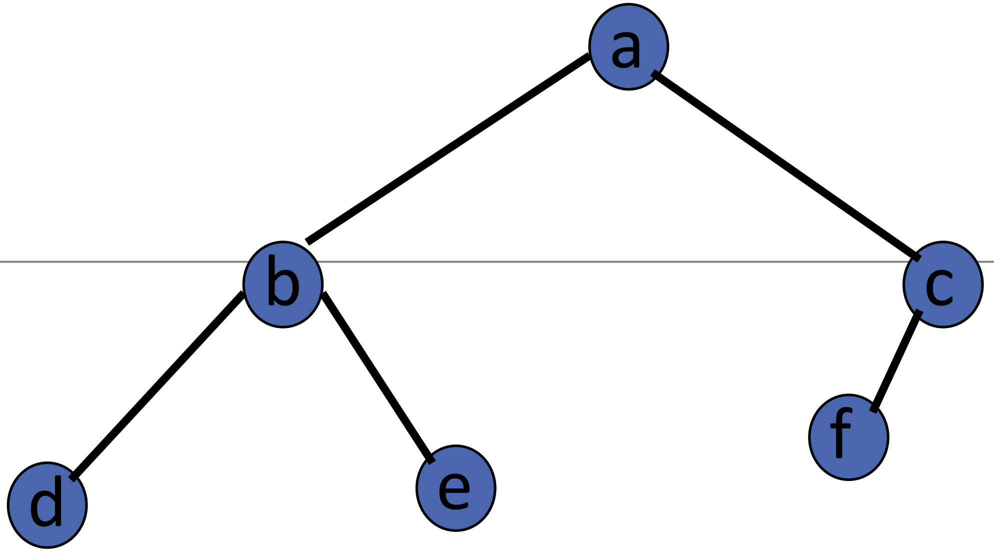


Queue

a									

Answer

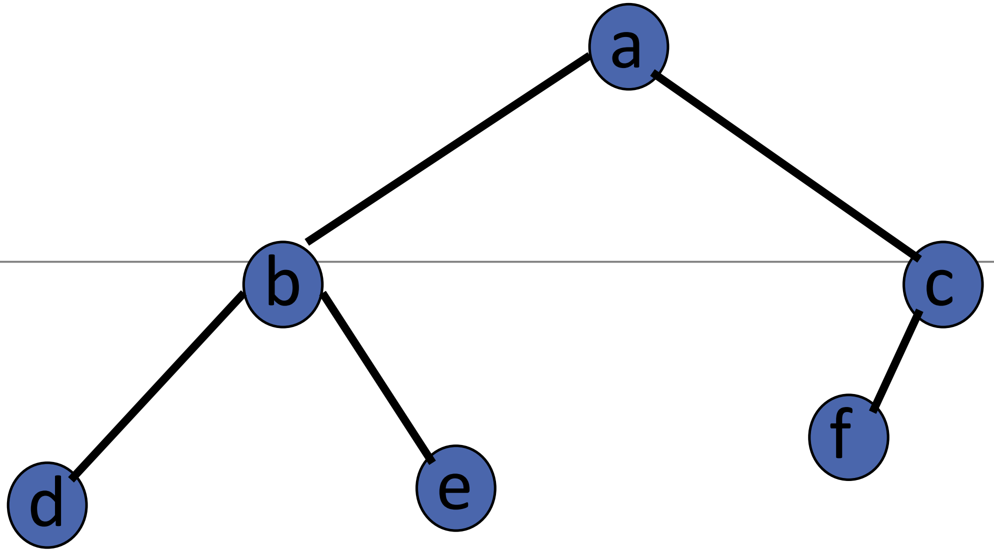
```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

    Remove from queue into temp;
     visit temp;
   if(temp->left is not NULL)
        insert temp->left into queue;
   if(temp->right is not NULL)
      insert temp->right into queue;

   }
 }
```



Queue

Answer
a

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

     Remove from queue into temp;
      visit temp;
   if(temp->left is not NULL)
         insert temp->left into queue;
   if(temp->right is not NULL)
       insert temp->right into queue;

   }
 }
```



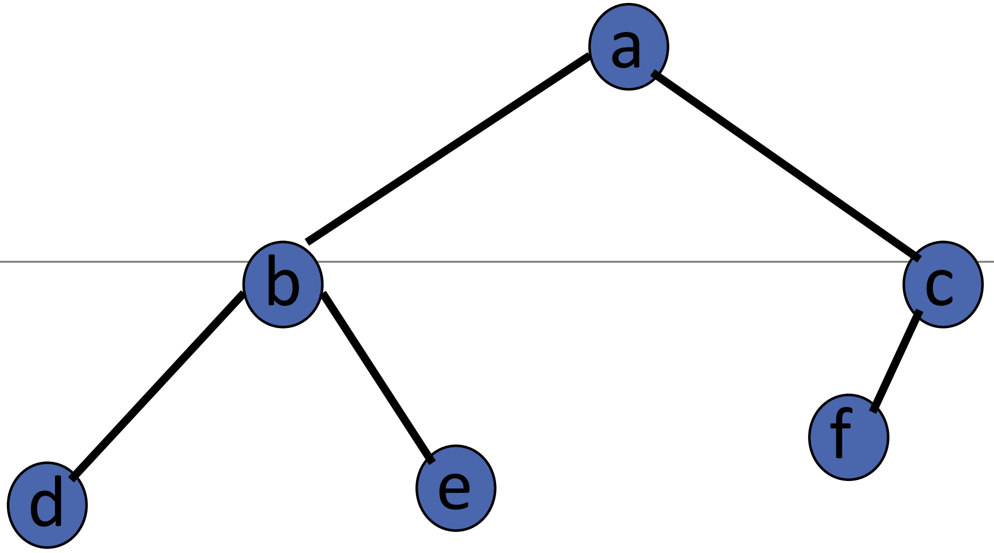
Queue

b	c								

Answer

a

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

    Remove from queue into temp;
     visit temp;
   if(temp->left is not NULL)
         insert temp->left into queue;
   if(temp->right is not NULL)
      insert temp->right into queue;

   }
 }
```



Queue

	c								

Answer

a b

Algorithm BFS(TreeNode *root)

{

temp=root;

Insert temp into queue;

while Queue not empty

{

Remove from queue into temp;

visit temp;

if(temp->left is not NULL)

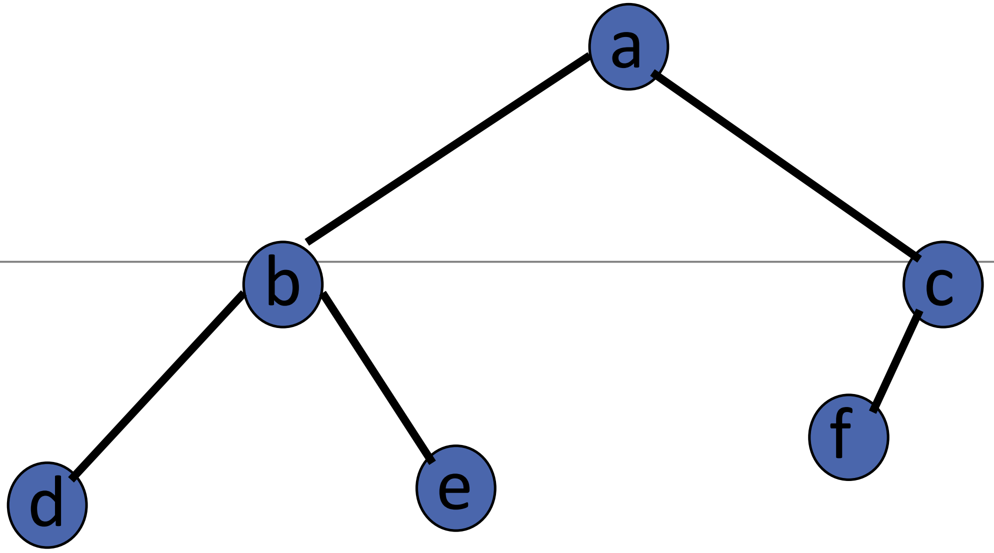
insert temp->left into queue;

if(temp->right is not NULL)

insert temp->right into queue;

}

}



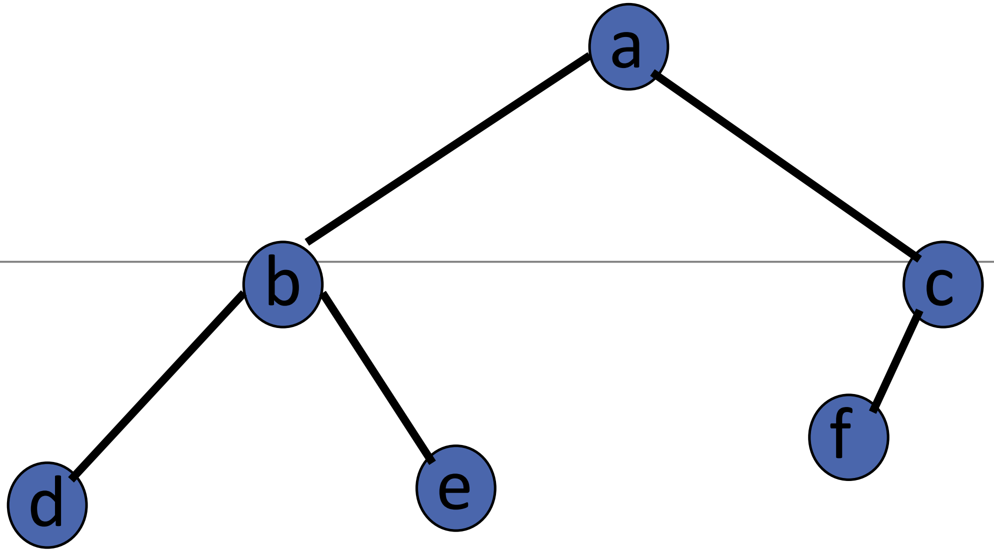
Queue

	c	d	e						

Answer

a b

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

    Remove from queue into temp;
     visit temp;
   if(temp->left is not NULL)
          insert temp->left into queue;
   if(temp->right is not NULL)
       insert temp->right into queue;

   }
 }
```



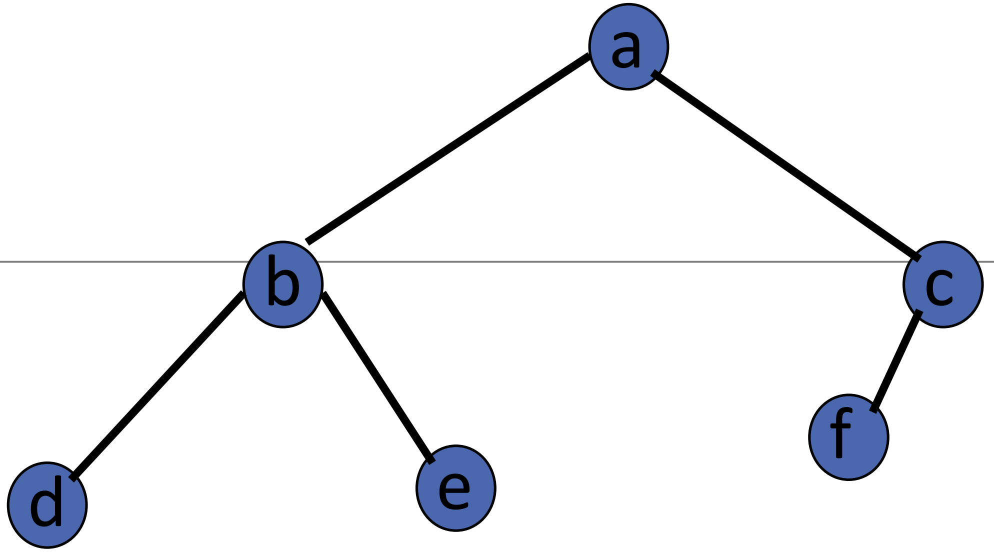
Queue

		d	e						

Answer

a b c

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

     Remove from queue into temp;
      visit temp;
   if(temp->left is not NULL)
         insert temp->left into queue;
   if(temp->right is not NULL)
       insert temp->right into queue;

   }
 }
```



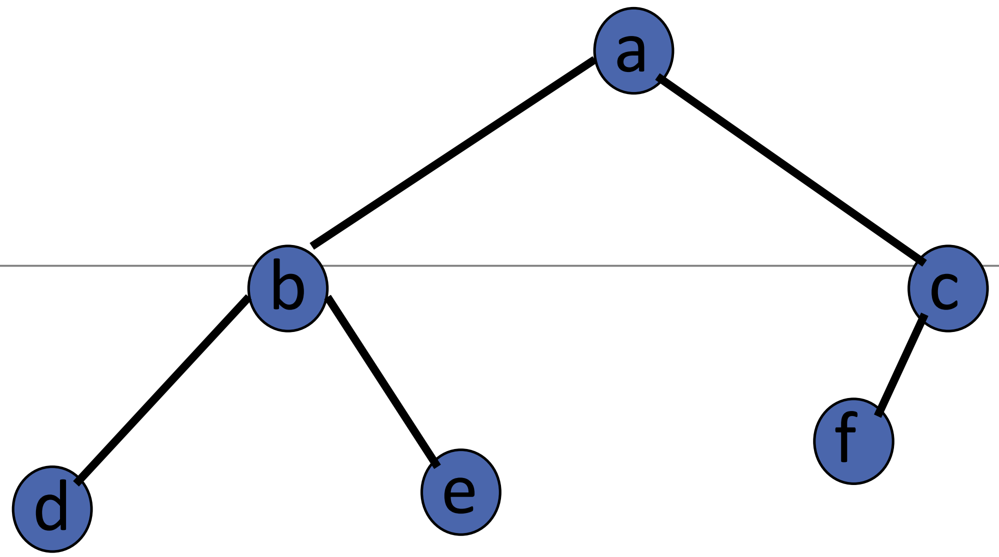
Queue

		d	e	f					

Answer

a b c

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

     Remove from queue into temp;
      visit temp;
   if(temp->left is not NULL)
         insert temp->left into queue;
   if(temp->right is not NULL)
       insert temp->right into queue;

   }
 }
```



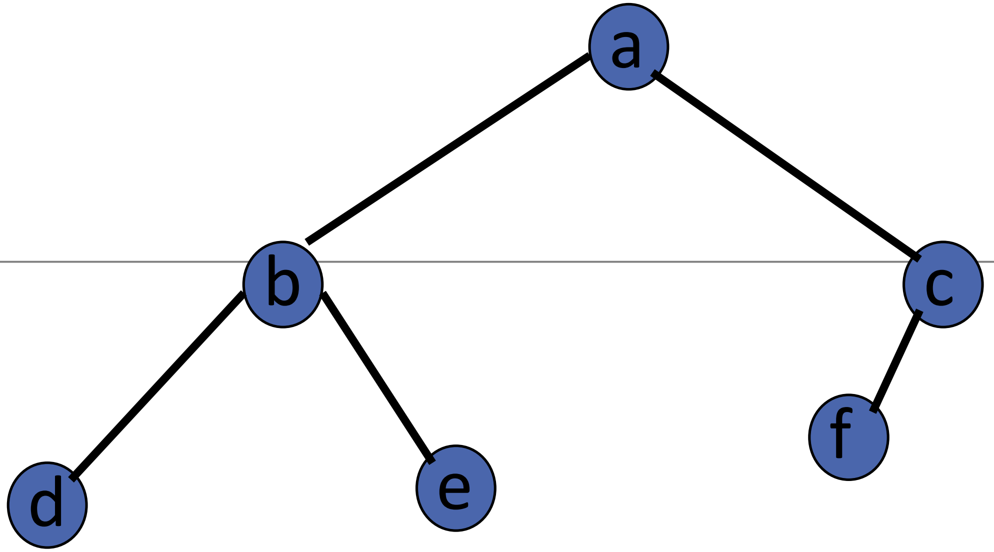
Queue

			e	f					

Answer

a b c d

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

     Remove from queue into temp;
      visit temp;
   if(temp->left is not NULL)
          insert temp->left into queue;
   if(temp->right is not NULL)
        insert temp->right into queue;

   }
  }
```



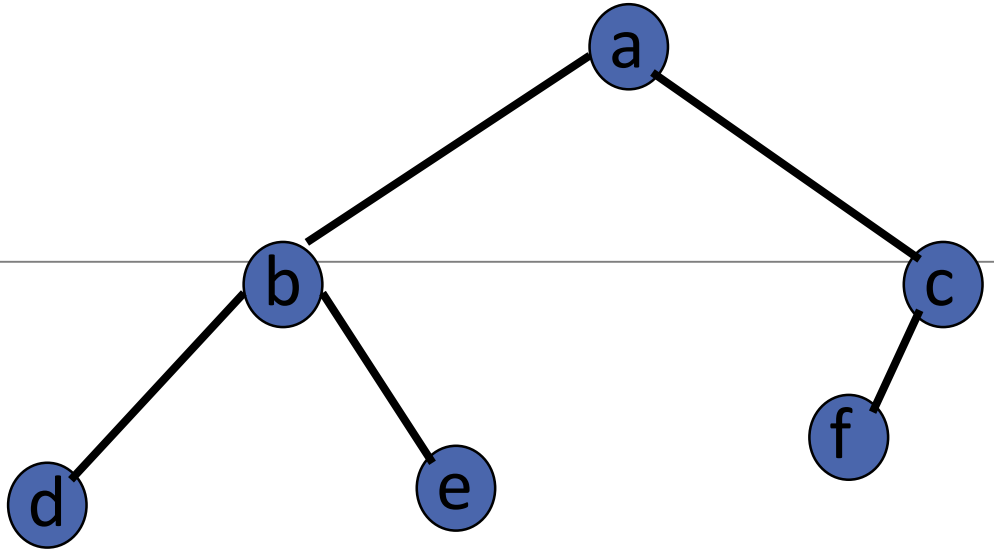
Queue

				f					

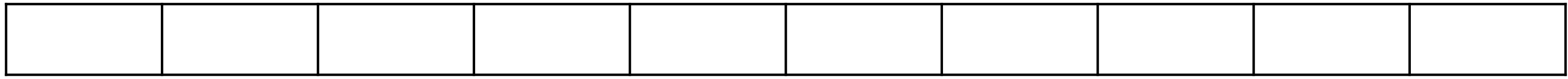
Answer

a b c d e

```
Algorithm BFS(TreeNode *root)
{
temp=root;
Insert temp into queue;
 while  Queue not empty
 {

    Remove from queue into temp;
     visit temp;
   if(temp->left is not NULL)
          insert temp->left into queue;
   if(temp->right is not NULL)
       insert temp->right into queue;

   }
  }
```



Queue



Answer

a b c d e f

Assignment no 1

2. Implement binary tree and perform following operations: Creation of binary tree and traversal recursive and non-recursive.

Operations on binary tree

Copying Binary Tree (recursive)

Copy of binary tree using non recursive is done through preorder

```
treenode  *copy(TreeNode *root)
{
    temp=NULL
    if (root!=NULL) {

       Allocate memory for temp
       temp->data=root->data;
       temp->left=copy(root->left);
       temp->right=copy(root->right);
    }
    return temp;
}
```

```
Algorithm copy_nr(TreeNode *root2)
{  //root2 is original tree
  Allocate memory for root1
temp1=root1;
temp2=root2;
copy(temp1->data,temp2->data);
while(1)
{
 while(temp2!=NULL)
 {
   if(temp2->left!=NULL)
   {
      Allocate memory for temp1->left;
    copy (temp1->left->data,temp2->left->data);
   }
   if(temp2->right!=NULL)
   {
      Allocate memory for temp1->right;;
     copy temp1->right->data,temp2->right->data);
   }
   push(temp1);
   push(temp2);
   Move temp1 to temp1->left
   Move temp2 to temp2->left
 }
 if stack empty     break;
  else
  {
     Pop to temp1
     Pop to temp2
     temp1=temp1->right;
     temp2=temp2->right;
  }
}            //end while
 }
```

Erasing nodes in binary tree

Use postorder

```
Algorithm depth_nr(TreeNode *root)
{
Initialize d to 0;
temp=root;
while(1)
{
 while(temp!=NULL)
 {
    push temp;
    move temp to temp->left;
  if(d<top)
   d=top;    }
 if(stack top right is NULL)
 {
      pop to temp;    }
  while(stack not empty && stack top right is temp)
  {
    pop to temp ;   }
 if  stack empty
   break;
 move temp to stack top right;
}
cout<<"\nDepth is "<<d+1; }
```

```
int depth_r(TreeNode *root)
 {
   Initialize t1=0,t2=0;
   if(root==NULL)
      return 0;
  else
  {
   t1=depth_r(root->left);
   t2=depth_r(root->right);
  if(t1>t2)
   return ++t1;
  else
   return ++t2;
 }
}
```

```
Algorithm mirror_r(TreeNode *root)
{
  swap left and right;
  if(root->left!=NULL)
  mirror_r(root->left);
  if(root->right!=NULL)
  mirror_r(root->right);
}
```

```
Algorithm mirror_nr(TreeNode *root)
{
   temp=root;
   insqueue(temp);
   while(!qempty())
  {
     temp=delqueue();
     swap left and right;
     if(temp->left!=NULL)
    insqueue(temp->left);
    if(temp->right!=NULL)
    insqueue(temp->right);
  }
  dispbfs(root);
}
```

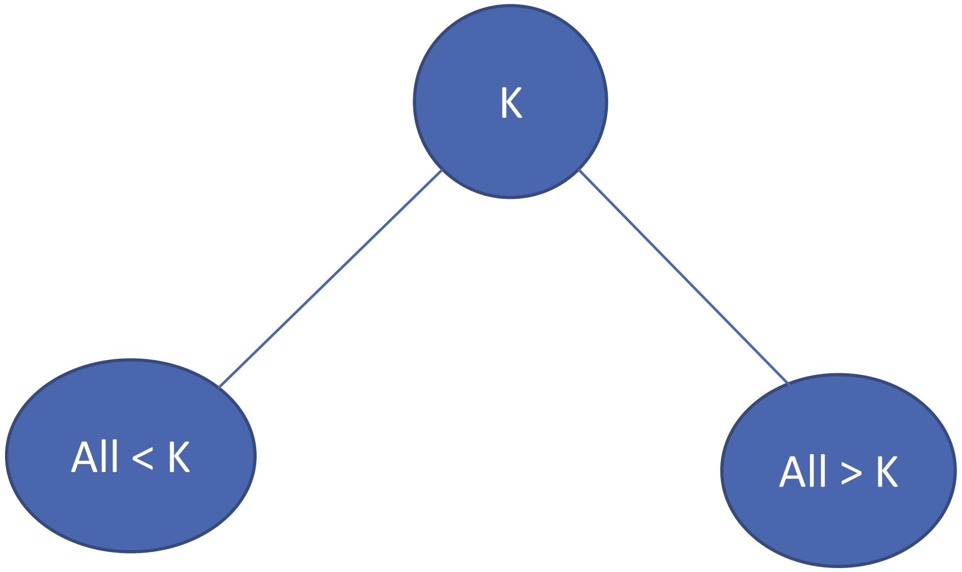
Binary search Trees

It is a binary tree.It may be empty.If it is not empty then it satisfies the following properties

- Every element has a unique key.
- The keys in a nonempty left subtree are smaller than the key in the root of subtree.
- The keys in a nonempty right subtree are larger than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

- *Binary search trees provide an excellent structure for searching a list and at the same time for inserting and deleting data into the list.*

Binary Search Tree



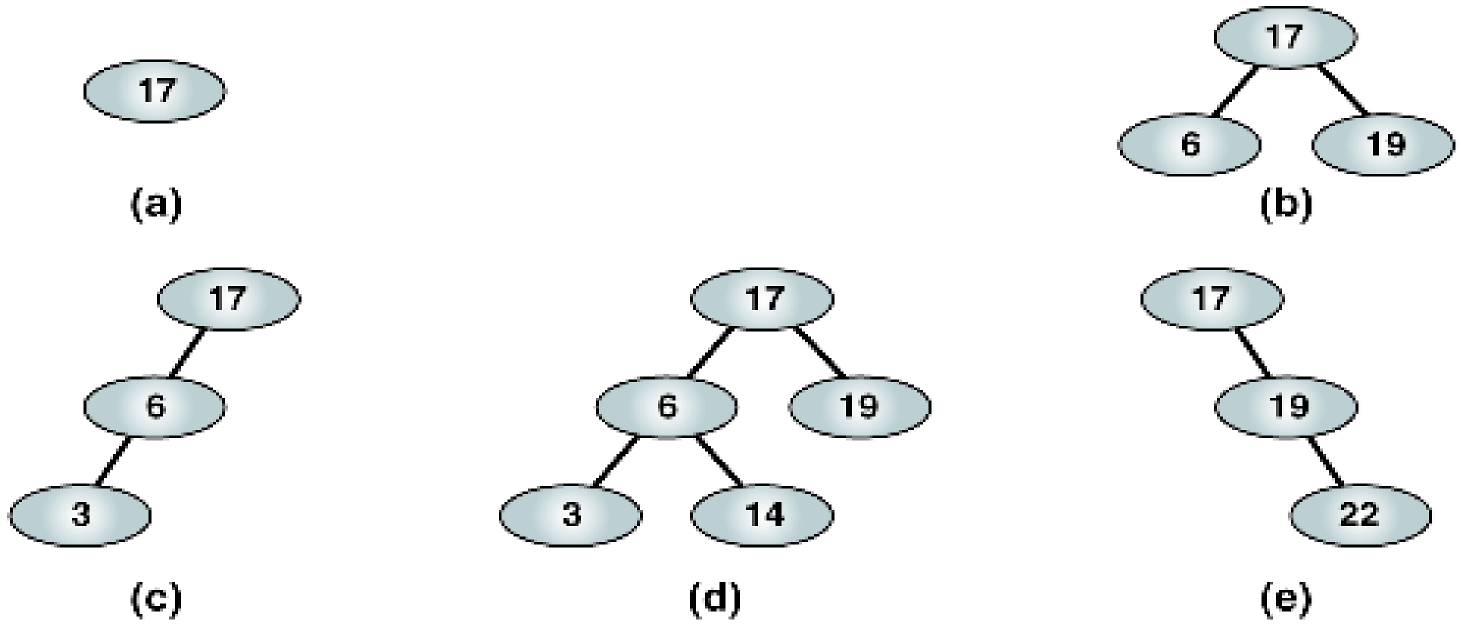


FIGURE 7-2 Valid Binary Search Trees

(a), (b) - complete and balanced trees;

(d) – nearly complete and balanced tree;

(c), (e) – neither complete nor balanced trees

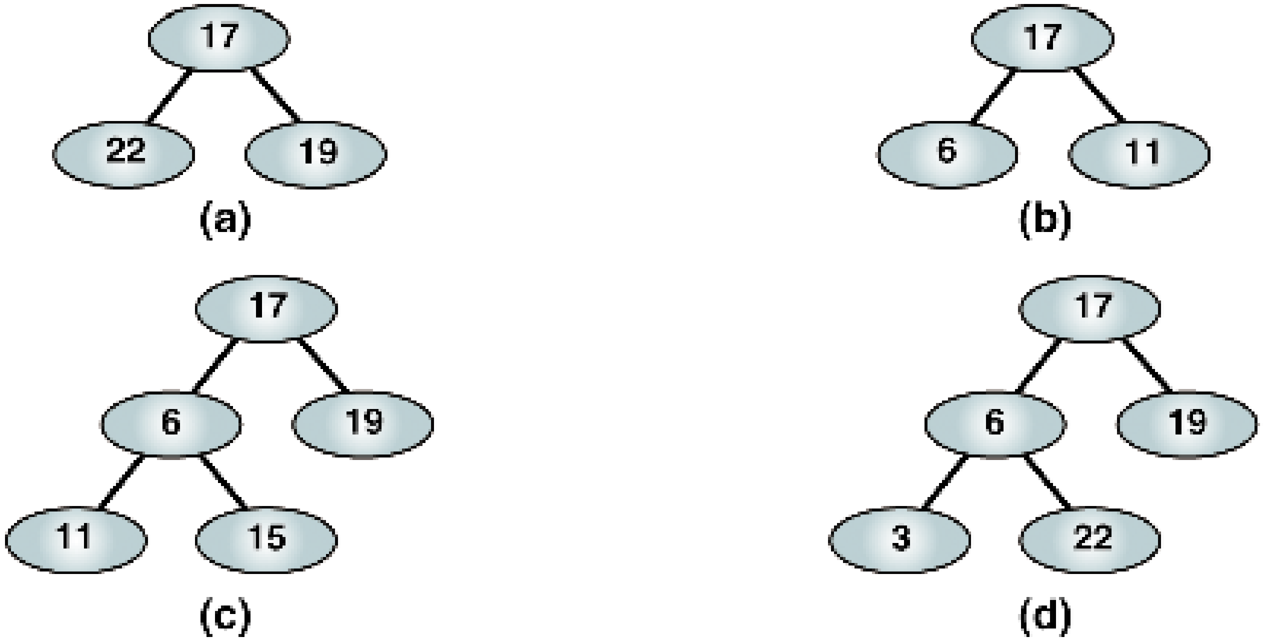


FIGURE 7-3 Invalid Binary Search Trees

Binary search Trees

```
typedef struct BST
{
    char word[10];
    char meaning[20];
    struct BST *left;
    struct BST *right;
}BST;
```

```
Algorithm  create()
{
     allocate memory and accept the data for root node;
    do
    {
        temp=root;
        flag=0;
        allocate memory and accept the data for curr node;
        while(flag==0)
        {
          if(curr->data < temp->data)
          {
            if(temp->left=NULL)
            {
              temp->left=curr;
              flag=1;
            }
        else
          move temp to temp->left
      }  //end if compare
else {
       if(temp->right=NULL)
       {
         temp->right=curr;
         flag=1;
       }
     else
       move temp to temp->right;
     }    //end else
   }  //end while flag
   Accept choice for adding more nodes;
   }while(choice =yes);   //end do
}   //end algorithm
```

binary search tree creation

Jyoti, Deepa, Rekha, Amit, Gilda, Anita, Abolee, Kaustubh, Teena, Kasturi, Saurabh

```
Algorithm search (BST *root)
{
  Initialize  flag = 0;
  Accept string to be searched ;
  flag = search_r(root, str);
  if(flag = 1)
    print found;
 else
   print not found;
}
```

```
Algorithm search_r(BST *temp, char str[10])
{
  Initialize f to 0;
  if(temp != NULL)
 {
    if(str = temp->data)
          return 1;
    if(str < temp->data)
          f = search_r(temp->left, str);
     if(string > temp->data)
         f = search_r(temp->right, str);
  }
  return f;
}
```

```
Algorithm search_nr(BST *root)
{
  Initialize  flag to 0;
 temp = root;
 Accept string to be searched;
 while(flag = 0 && temp != NULL)
 {
   if(string = temp->data)
  {
   flag=1;   break;
  }
  else if(string < temp->data)
     move temp to temp->left;
   else
     move temp to temp->right;
  }   //end while
  if(flag = 1)
     Print found;
 else
    Print not found;
}   //end algorithm
```

Function DeleteItem

First, find the item; then, delete it

Important: binary search tree property must be preserved!!

We need to consider following different cases:

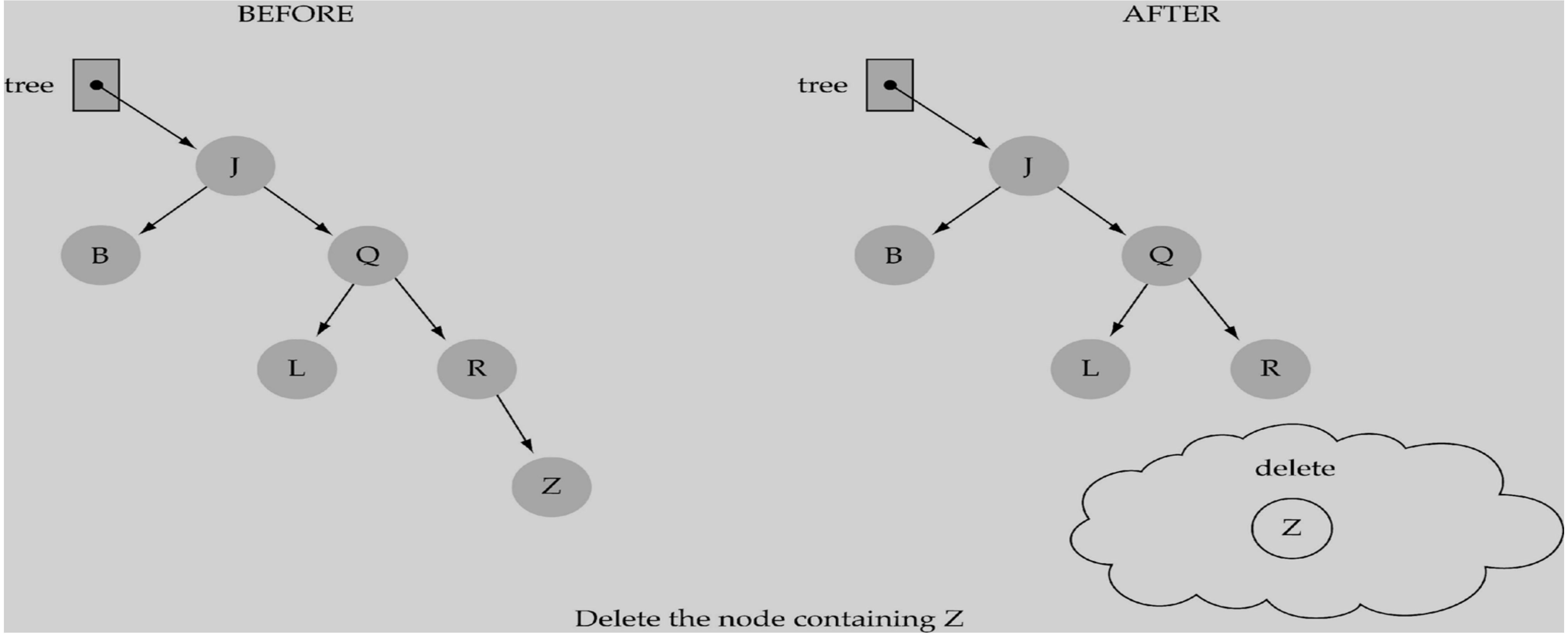
(1) Deleting a leaf

(2) Deleting a node with only one child

(3) Deleting a node with two children

(4) Deleting the root node

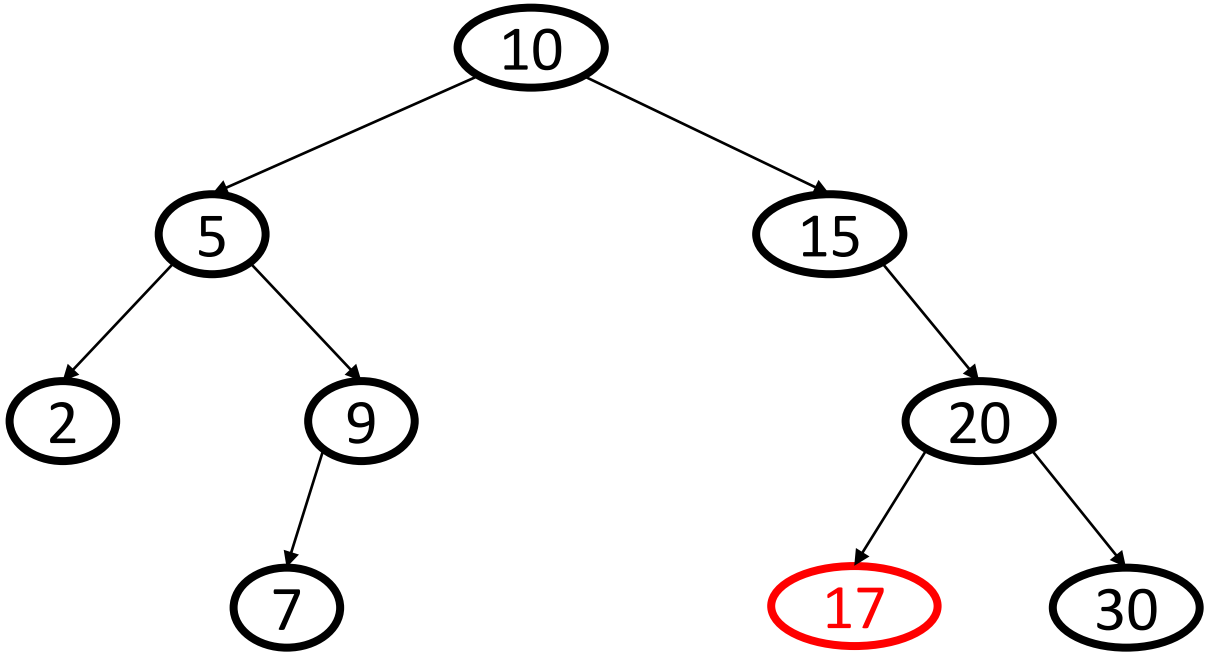
(1) Deleting a leaf



Deletion - Leaf Case

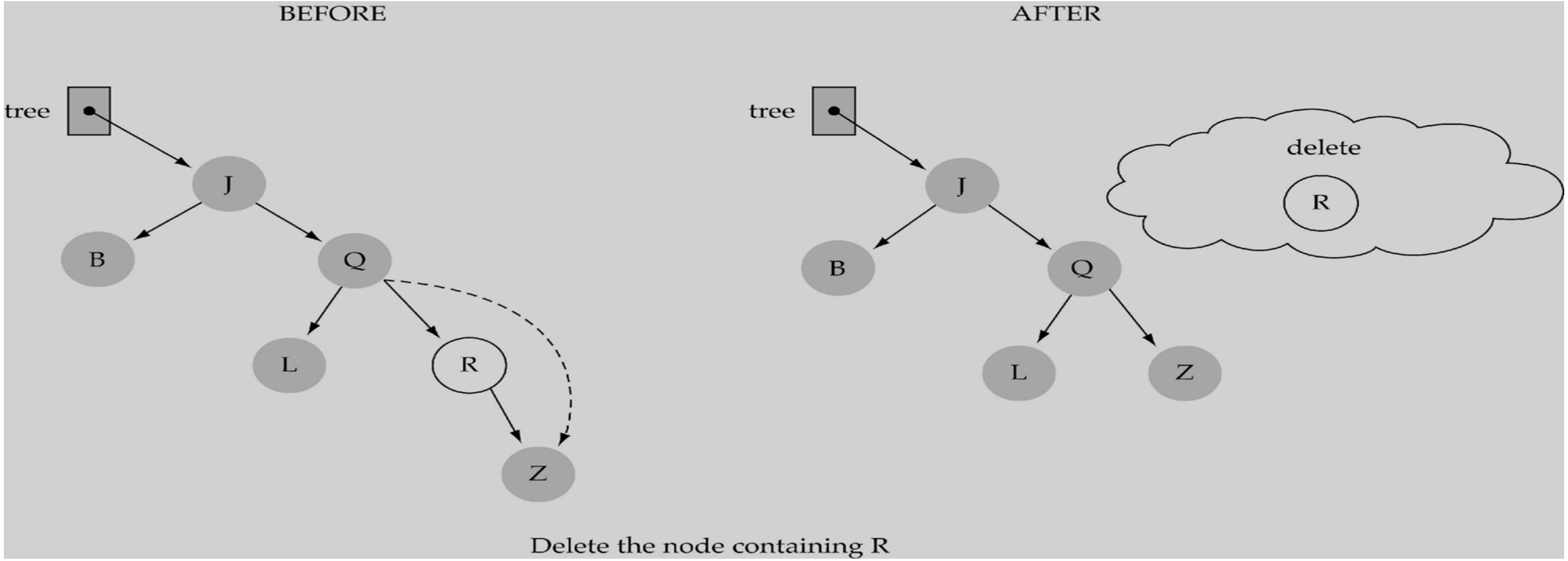
Algorithm sets corresponding link of the parent to NULL and disposes the node

Delete(17)



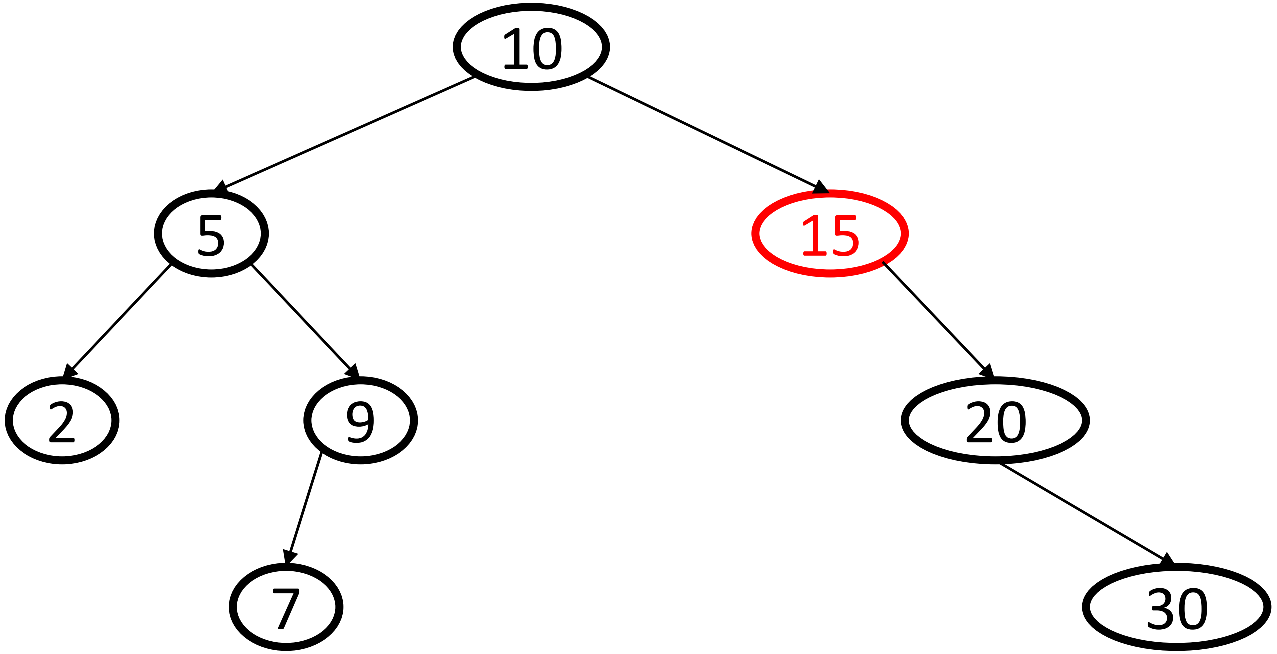
(2) Deleting a node with only one child

It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

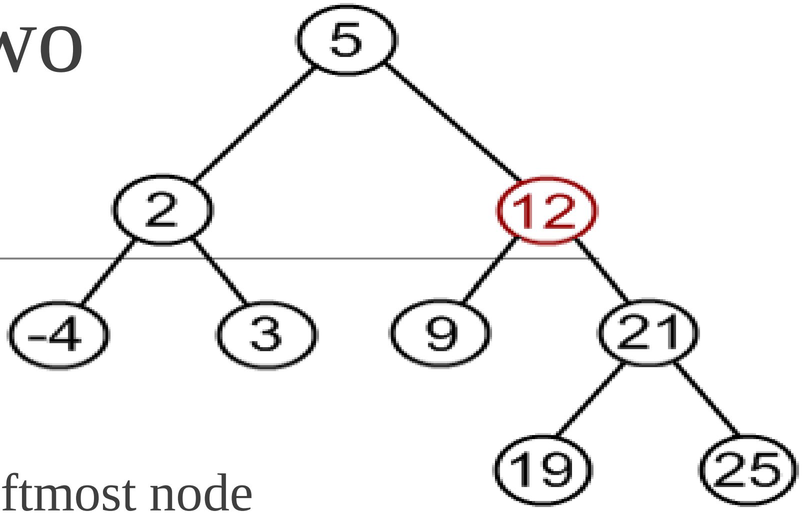


Deletion - One Child Case

Delete(15)



(3) Deleting a node with two children (contd…)

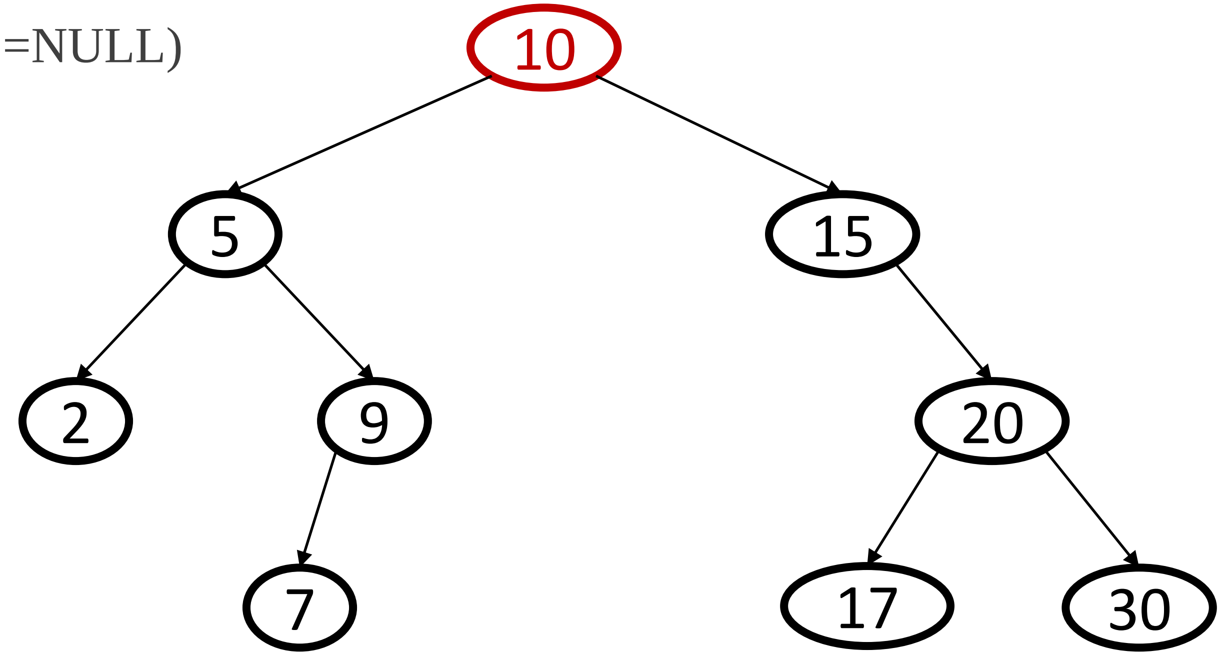


Find inorder successor

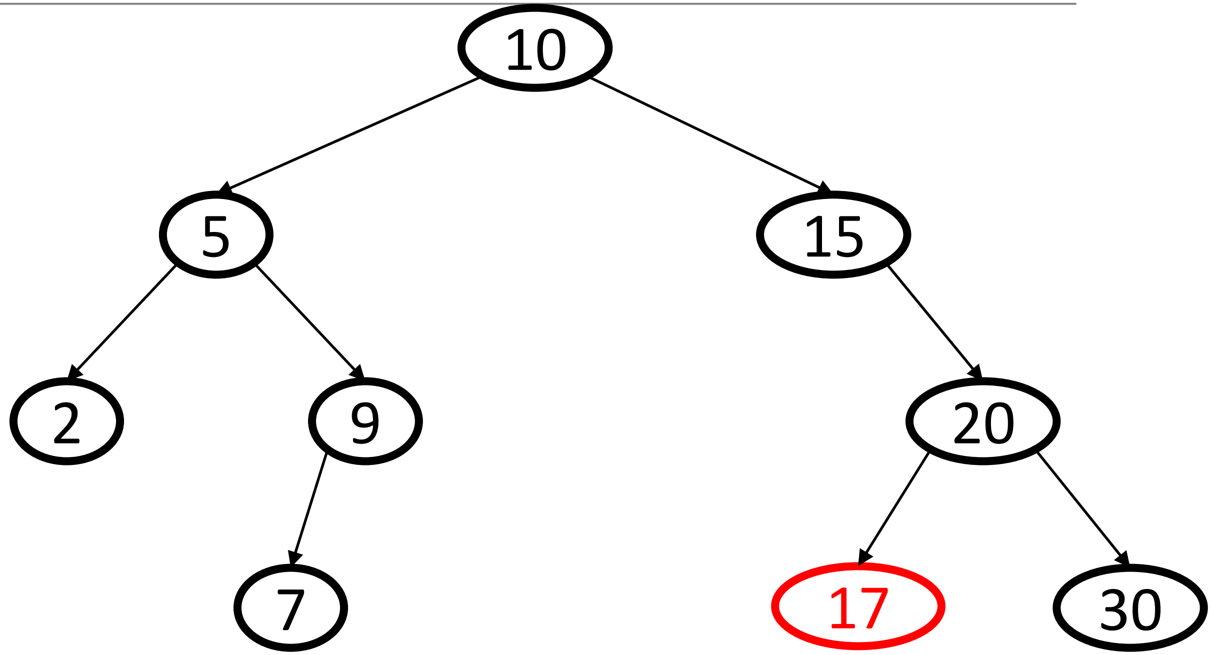
- Go to the right child and then move to the left till we get NULL for the leftmost node

- To the inorder's successor ,attach the left of the node which we want to delete

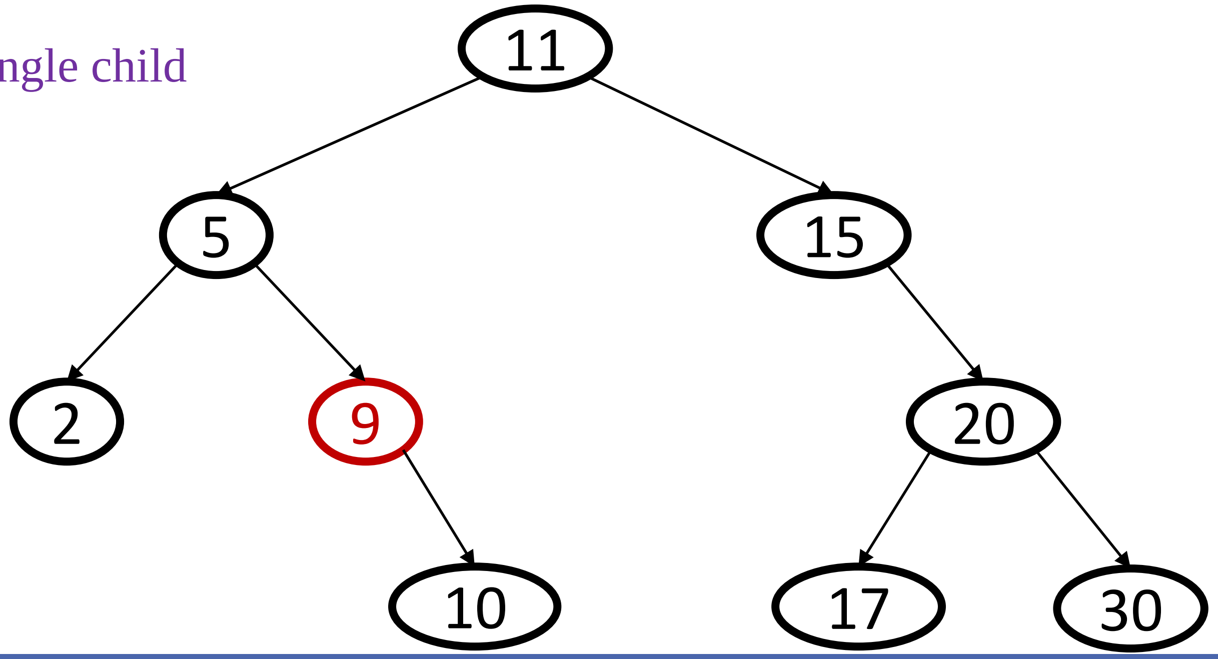
```
if(curr==root)                                  //deletion of root
{
 if(curr->rightc==NULL)
              root=root->leftc;
 else if(curr->leftc==NULL)
              root=root->rightc;
 else if(curr->rightc!=NULL && curr->leftc!=NULL)
 {
              temp=curr->leftc;
              root=curr->rightc;
              s=curr->rightc;
              while(s->leftc!=NULL)
              {
                     s=s->leftc;
              }
              s->leftc=temp;
 }
}
```



```
else if(curr!=root)            //deletion of node which is not root
{
 if(curr left and right is NULL )       //deletion of a leaf
 {
        if(parent->leftc==curr)
        parent->leftc=NULL;
        else
        parent->rightc=NULL;
 }
```

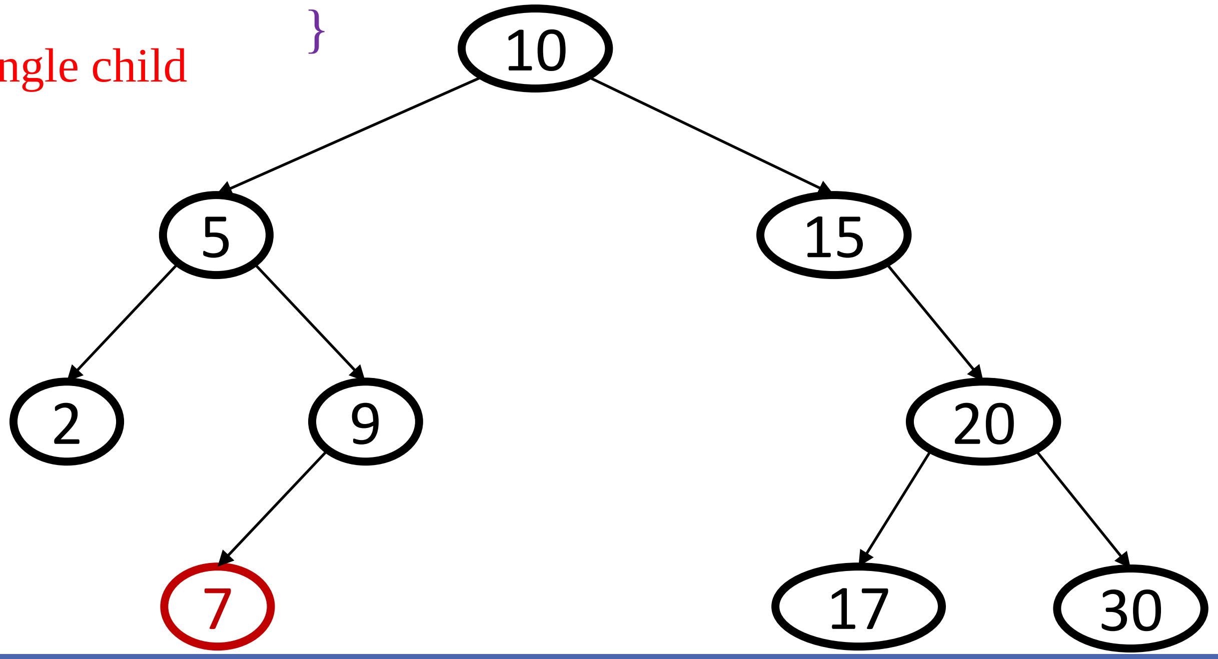


```
else if(curr!=root)          //deletion of node which is not root
{
 if(curr left and right is NULL )        //deletion of a leaf
 {
        if(parent->leftc==curr)
        parent->leftc=NULL;
        else
        parent->rightc=NULL;
 }
 else if(curr->leftc is NULL)      //deletion of a single child
 {
        if(parent->leftc==curr)
        parent->leftc=curr->rightc;
        else
        parent->rightc=curr->rightc;
 }
```

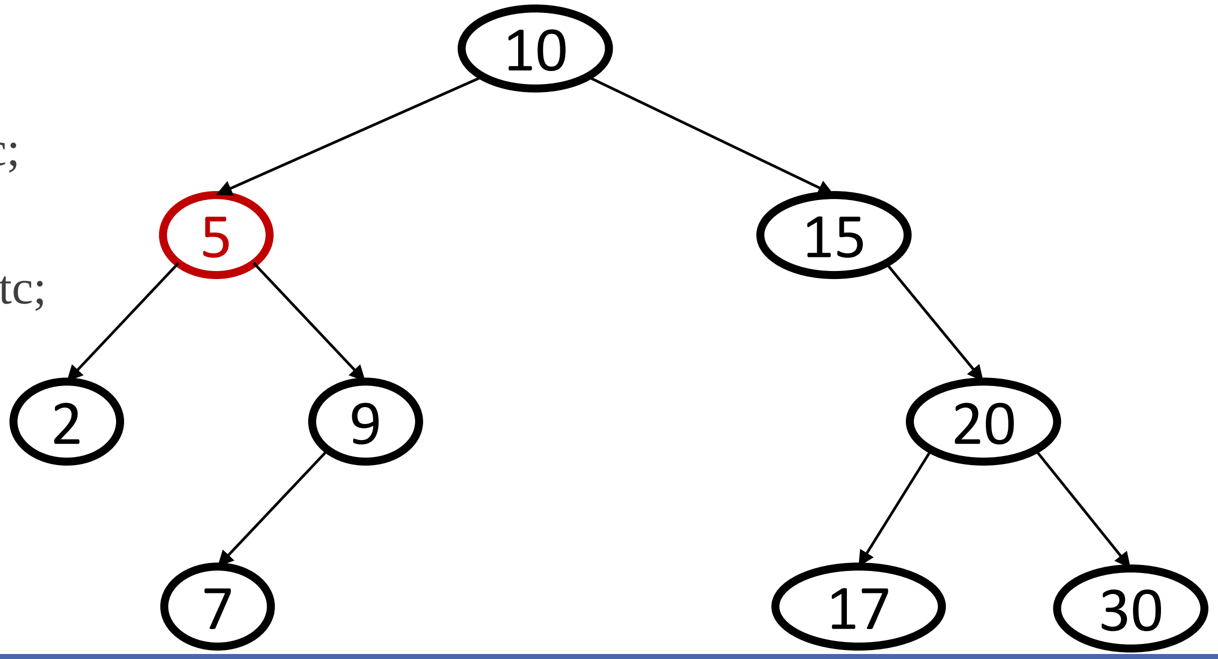


```
else if(curr!=root)            //deletion of node which is not root
{
 if(curr left and right is NULL )        //deletion of a leaf
 {
          if(parent->leftc==curr)
          parent->leftc=NULL;
          else
          parent->rightc=NULL;
 }
 else if(curr->leftc is NULL)      //deletion of a single child
 {
          if(parent->leftc==curr)
          parent->leftc=curr->rightc;
          else
          parent->rightc=curr->rightc;
 }
```

```
else if(curr->rightc is NULL)  //deletion of a
 single child
 {
          if(parent->leftc==curr)
          parent->leftc=curr->leftc;
          else
          parent->rightc=curr->leftc;
 }
```



```
else                                    //deletion of a node having two child
 {
        s=curr->rightc;
        temp=curr->leftc;
        while(s->leftc!=NULL)
        {
                s=s->leftc;
        }
        s->leftc=temp;
        if(parent->leftc==curr)
                parent->leftc=curr->rightc;
        else
                parent->rightc=curr->rightc;
 }
}
Assign curr left and right to NULL;
free(curr);
}
```



Assignment

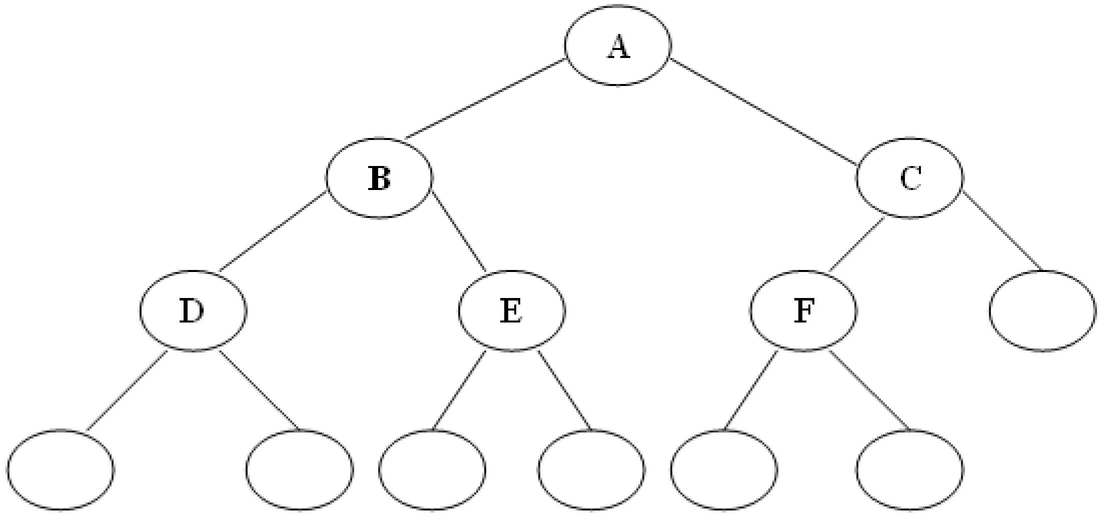
Implement dictionary using binary search tree where dictionary stores keywords & its meanings. Perform following operations:

1. Insert a keyword
2. Delete a keyword
3. Create mirror image and display level wise
4. Copy

Threaded Binary Tree

In a linked representation of a binary tree, the number of null links (null pointers) are actually more than non-null pointers.

Consider the following binary tree:



A Binary tree with the null pointers

Threaded Binary Trees

Two many null pointers in current representation of binary trees

n: number of nodes	6
number of non-null links: n-1	5
total links: 2n	12
null links: 2n-(n-1)=n+1	7

Replace these null pointers with some useful “threads”.

Threaded Binary Tree

The objective here to make effective use of these null pointers.

- According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.

- And binary tree with such pointers are called threaded tree.

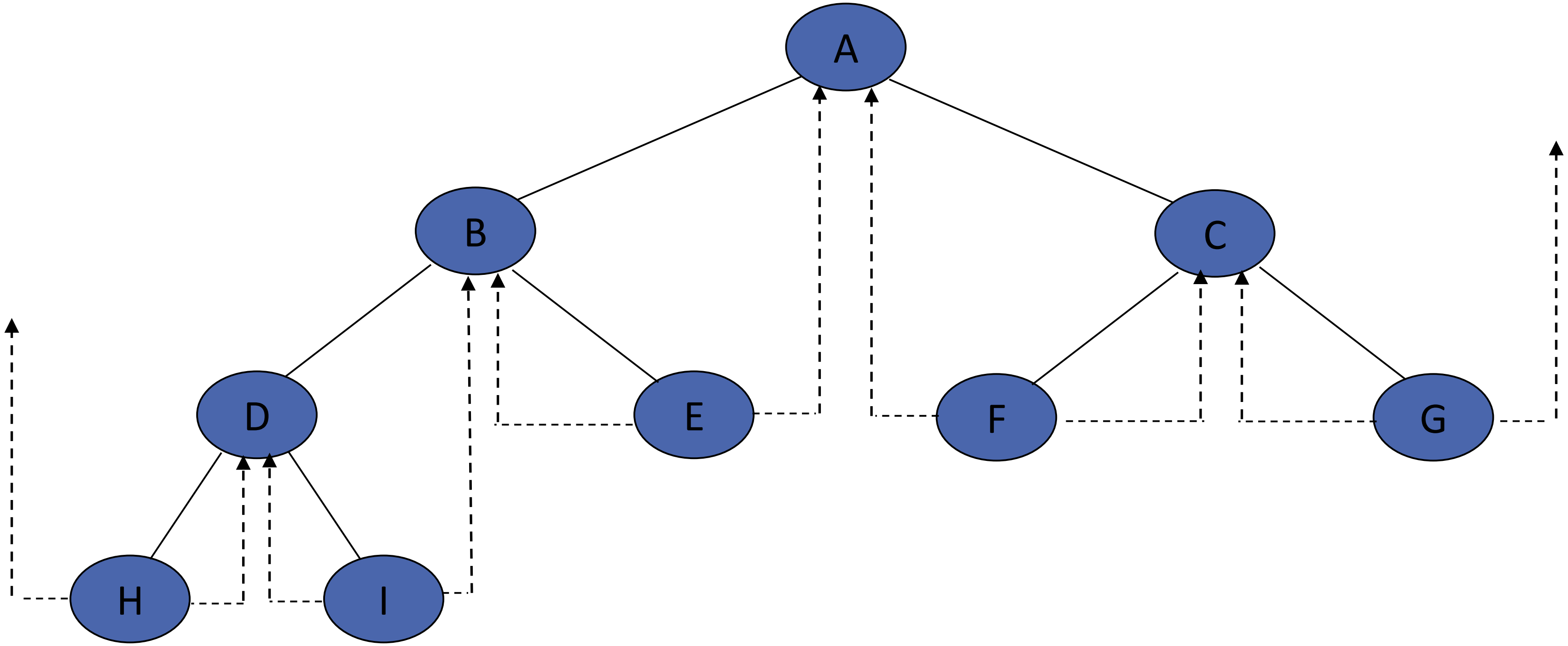
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

Threaded Binary Tree

Threading Rules

- RightChild null link at node p is replaced by the inorder successor of p.
- LeftChild null link at node p is replaced by the inorder predecessor of p.

Threaded Tree



Inorder sequence: H, D, I, B, E, A, F, C, G

Threads

To distinguish between normal pointers and threads, two Boolean fields, LeftThread and RightThread, are added to the record in memory representation.

Threaded Binary Tree

- Therefore we have an alternate node representation for a threaded binary tree which contains five fields as show bellow:

For any node p, in a threaded binary tree.

lthred(p)=0 indicates lchild (p) is a thread pointer

lthred(p)=1 indicates lchild (p) is a normal

rthred(p)=0 indicates rchild (p) is a thread

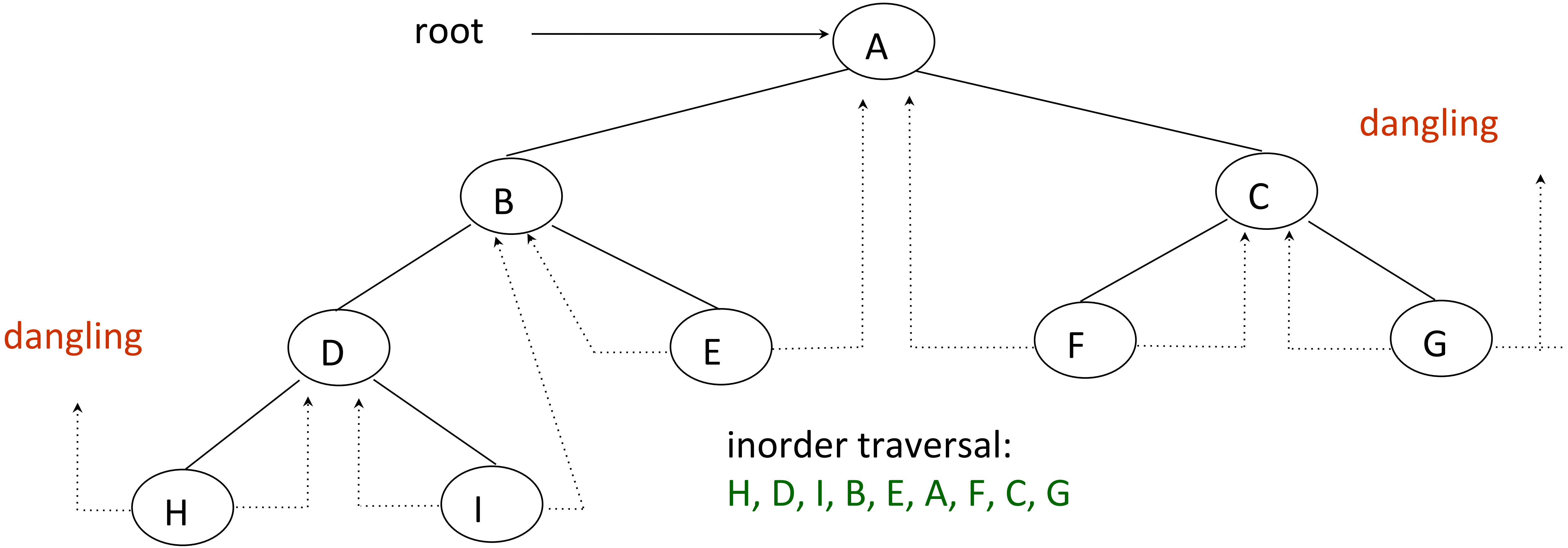
rthred(p)=1 indicates rchild (p) is a normal pointer

Threaded Binary Trees (*Continued*)

If ptr->left_child is null,
replace it with a pointer to the node that would be visited *before* ptr in an *inorder traversal*

If ptr->right_child is null,
replace it with a pointer to the node that would be visited *after* ptr in an *inorder traversal*

A Threaded Binary Tree



inorder traversal:

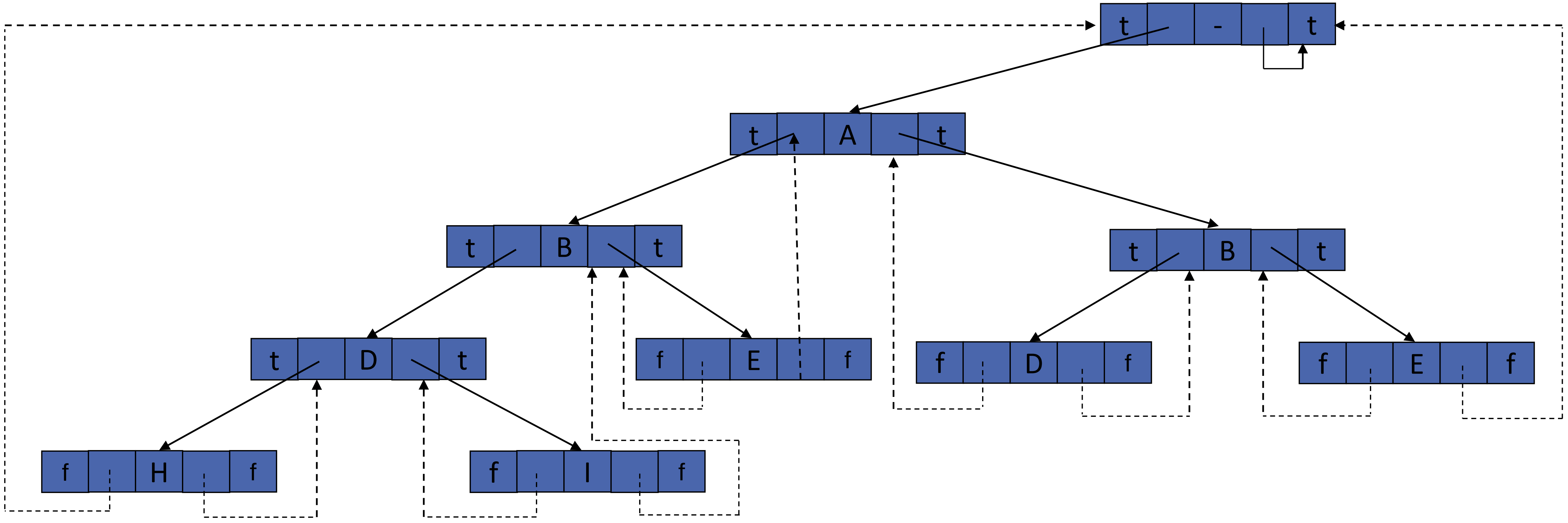
H, D, I, B, E, A, F, C, G

Threads (Contd...)

- To avoid dangling threads, a head node is used in representing a binary tree.

- The original tree becomes the left subtree of the head node.

Memory Representation of Threaded Tree



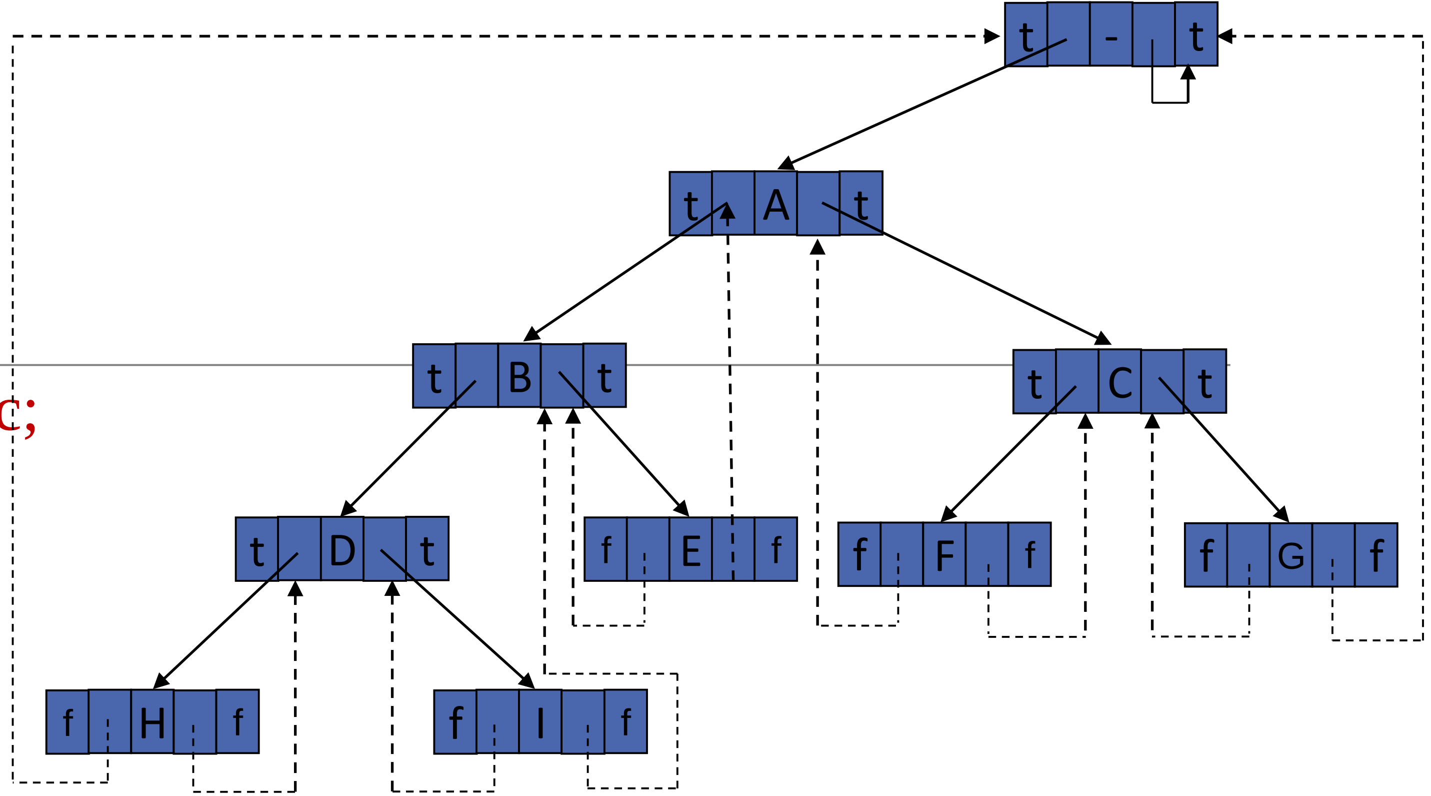
```
struct tbtnode{
        char data;
        int rbit, lbit;
        tbtnode *rightc;
        tbtnode *leftc;
};
```

```
main(){
        tbtnode *head;
        Allocate memory for head;
        rbit = 1;
        lbit = 0;
        head->left = head-> right = head;
        create(head);
        inorder(head);
        preorder(head);
}
```

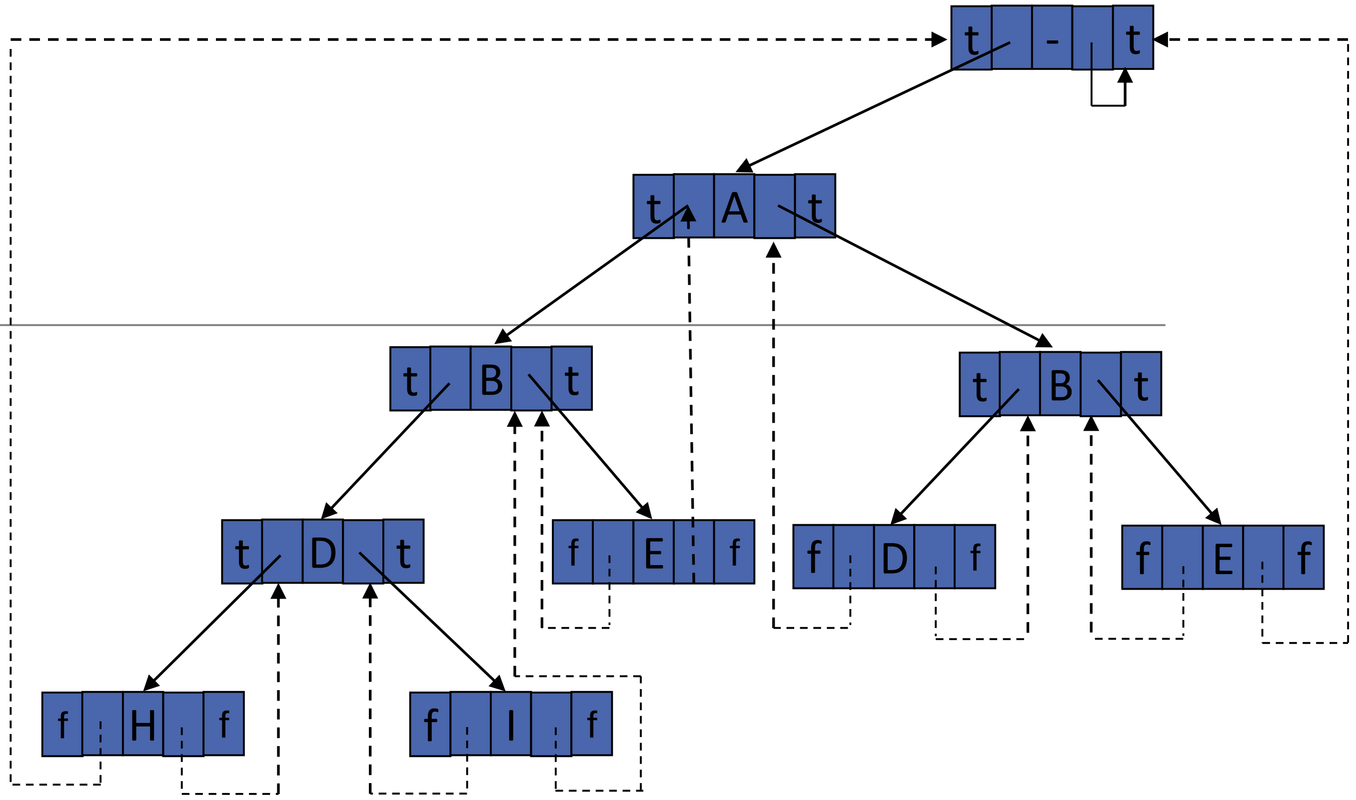
```
Algorithm create(tbtnode *head)
{
 Allocate memory for root;
 Accept root data;
 Assign head lbit to 1;
 Assign root->leftc and rightc to head;
 Assign root->lbit and rbit to 0;
 Assign head->leftc to root;

do
{
    Initialize  flag to 0;
    temp=root;
    Allocate memory to curr  and accept curr->data;
    Assign curr->lbit and rbit to 0;
```

```
while(flag==0)
 {
     Accept choice left or right;
   if  ch1='l'
   {
      if(temp->lbit==0)
     {

          curr->rightc=temp;
          curr->leftc=temp->leftc;
          temp->leftc=curr;
          temp->lbit=1;
          flag=1;
     }
   else
          temp=temp->leftc;
     }// end if for left
```

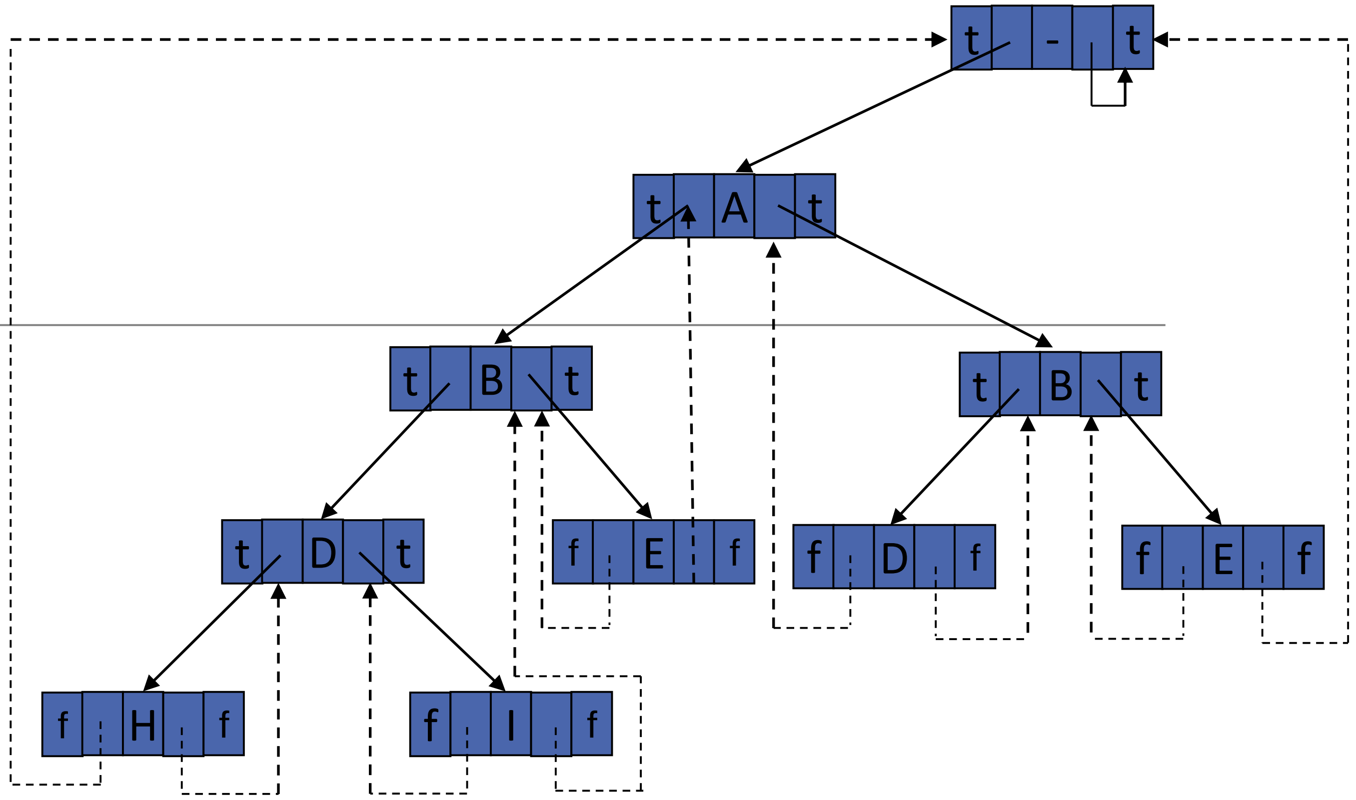
```
if(ch1=='r')
 {
        if(temp->rbit==0)
        {
                curr->leftc=temp;
                curr->rightc=temp->rightc;
                temp->rightc=curr;
                temp->rbit=1;
                flag=1;
        }
        else
                temp=temp->rightc;
 }  // end if for right
}   //end while flag
 Accept choice for continue;

}while(ch=='y');

} //end algo
```



```
Algorithm inorder(tbtnode *head)
{
  temp = head;
  while(1)
 {
          temp=inordersucc(temp);
           if(temp == head) break;
           print temp->data;
 }

}
Algorithm node * inordersucc(temp)
{
    x=temp->right;
    if(temp->rbit==1)
     {
        while(x->lbit==1)
              x=x->left;
     }
     return x;
}
```



```
Algorithm preorder(tbtnode *head)
{
 Assign temp to head->left;
 while(temp != head)
 {
          print temp->data;
          while(temp->lbit != 0)
          {
                    move temp to temp->left;
                    print temp->data;
          }
          while(temp->rbit == 0)
                     move temp to temp->right;
          move temp to temp->right;
 }

}
```



Advantages of threaded binary tree:

- The traversal operation is more faster than that of its unthreaded version, because with threaded binary tree non-recursive implementation is possible which can run faster and does not require the botheration of stack management.

- We can efficiently determine the predecessor and successor nodes starting from any node. In case of unthreaded binary tree, however, this task is more time consuming and difficult.

- Any node can be accessible from any other node. Threads are usually more to upward whereas links are downward. Thus in a threaded tree, one can move in their direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root.

Threaded Binary Tree

Disadvantages of threaded binary tree:

- Insertion and deletion from a threaded tree are very time consuming operation compare to non-threaded binary tree.
- This tree require additional bit to identify the threaded link.

Practice Problems

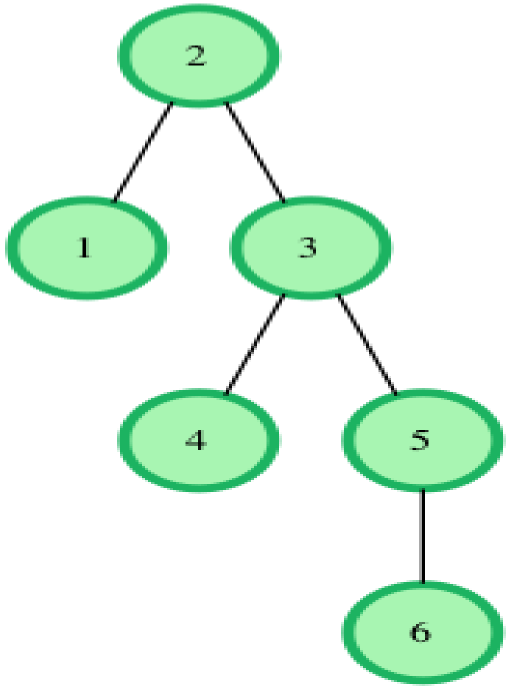
1.Given a pointer to the root of a binary tree, print the top view of the binary tree.

The tree as seen from the top the nodes, is called the top view of the tree.

For example :

Sample Input: **Sample Output- 1->2->5->6**

2. You are given pointer to the root of the binary search tree and two values v1 and v2 . You need to return the lowest common ancestor (LCA) of v1 and v2 in the binary search tree.



In the diagram above, the lowest common ancestor of the nodes 4 and 6 is the node 3. Node 3 is the lowest node which has nodes 4 and 6 as descendants.

3. Given a tree and an integer, k, in one operation, we need to swap the subtrees of all the nodes at each depth h, where h ∈ [k, 2k, 3k,...]. In other words, if h is a multiple of k, swap the left and right subtrees of that level.

You are given a tree of n nodes where nodes are indexed from [1..n] and it is rooted at 1. You have to perform t swap operations on it, and after each swap operation print the in-order traversal of the current state of the tree.

Input Format

The first line contains n, number of nodes in the tree.

Each of the next n lines contains two integers, a b, where a is the index of left child, and b is the index of right child of i^{th} node.

Note: -1 is used to represent a null node.

Sample Input 0

```
3
2 3
 -1 -1
 -1 -1
 2
1
1
```

Sample Output 0

```
3 1 2
 2 1 3
```