MIT-WPU

।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Dr. Vishwanath Karad

MIT WORLD PEACE UNIVERSITY | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

CET2001B Advanced Data Structures

S. Y. B. Tech CSE

Semester – IV

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

CET2001B: Advanced Data Structures

Pre-requisites: Fundamentals of Data Structures

Course Objectives:

1. Knowledge

i. Learn the nonlinear data structure and its fundamental concept.

2. Skills

i. Understand the different nonlinear data structures such as Trees and Graph.

ii. Study the concept of symbol table, heap, search tree and multiway search tree.

iii. Study the different ways of file organization and hashing concepts.

3. Attitude

i. Learn to apply advanced concepts of nonlinear data structure to solve real world problems.

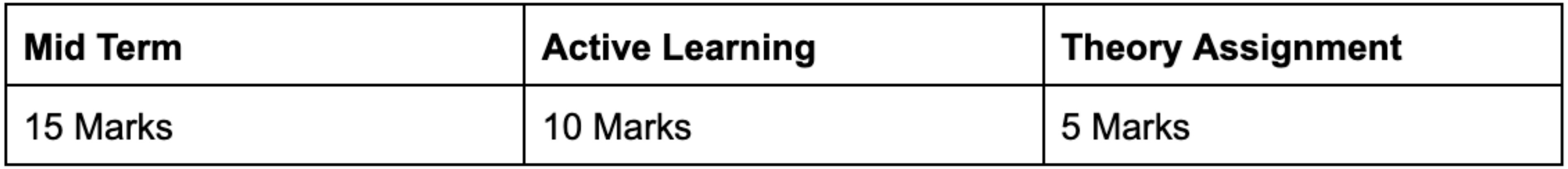
Course Outcomes:

After completion of the course the students will be able to :-

1. To choose appropriate non-linear data structures to solve a given problem.
2. To apply advanced data structures for solving complex problems of various domains.
3. To apply various algorithmic strategies to approach the problem solution.
4. To compare and select different file organization and to apply hashing for implementing direct access organization.

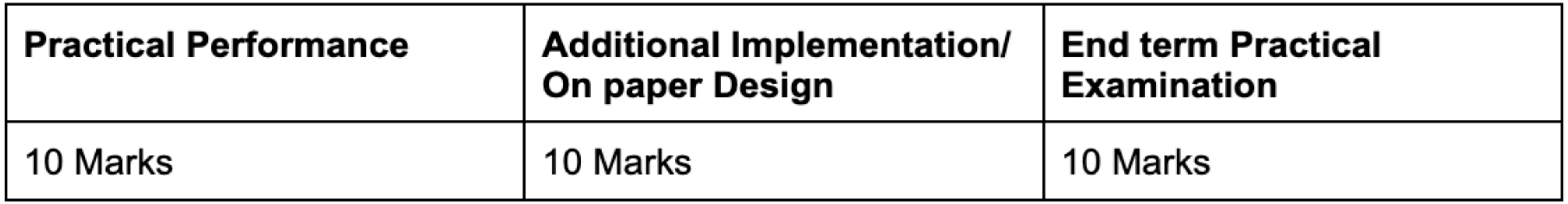
CET2001B Advanced Data Structures: Assessment Scheme:

Class Continuous Assessment (CCA) - 30 Marks



Mid Term	Active Learning	Theory Assignment
15 Marks	10 Marks	5 Marks

Laboratory Continuous Assessment (LCA) - 30 Marks



Practical Performance	Additional Implementation/ On paper Design	End term Practical Examination
10 Marks	10 Marks	10 Marks

Term End Examination: 40 Marks

Syllabus

1. Hashing - Concepts-hash table, hash function, basic operations, bucket, collision, probe, synonym, overflow, open hashing, closed hashing, perfect hash function, load density, full table, load factor, rehashing, issues in hashing, hash functions- properties of good hash function, division, multiplication, extraction, mid-square, folding and universal, Collision resolution strategies- open addressing and chaining, Hash table overflow- open addressing and chaining.

2. 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 In-order Threaded Binary tree.
Case Study- Expression tree

3. Graph - Basic Terminology, Graphs (Directed, Undirected), Various Representations, Traversals & Applications of graph- Prim’s and Kruskal’s Algorithms, Dijsktra's Single source shortest path, Analysis complexity of algorithm, topological sorting.

Continued…

4. Heap - Heap as a priority queue, Heap sort. Symbol Table-Representation of Symbol Tables- Static tree table and Dynamic tree table, Weight balanced tree - Optimal Binary Search Tree (OBST), OBST as an example of Dynamic Programming, Height Balanced Tree- AVL tree.

Search trees: Red-Black Tree, AA tree, K-dimensional tree, Splay Tree.

5. Multiway search trees, B-Tree - insertion, deletion, B+Tree - insertion, deletion, use of B+ tree in Indexing, Trie Tree.

Files: concept, need, primitive operations. Sequential file organization - concept and primitive operations, Direct Access File- Concepts and Primitive operations, Indexed sequential file Organization-concept, types of indices, structure of index sequential file, Linked Organization - multi list files.

List of Assignments

1. Implement following polynomial Operations using Circular Linked List :
   1) Create 2) Display 3) Addition

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

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

- Insert a keyword
- Delete a keyword
- Create mirror image and display level wise
- Copy
- Create mirror image and display level wise

4. Implement threaded binary tree. Perform inorder traversal on the threaded binary tree.

List of Assignments contd…

5. Consider a friend’s network on Facebook social web site. Model it as a graph to represent each node as a user and a link to represent the friend relationship between them using adjacency list representation and perform DFS traversal. Perform BFS traversal for the above graph.

6. A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. (Create & display of Graph). Solve the problem using Prim’s algorithm.

7. Read the marks obtained by students of second year in an online examination of a particular subject. Find the maximum and minimum marks obtained in that subject. Use heap data structure and heap sort.

8. Implement direct access file using hashing (linear probing with and without replacement) perform following operations on it a) Create Database b) Display Database c) Add a record d) Search a record e) Modify a record

9.Design a Project to implement a Smart text editor.

Learning Resources

Text Books:

1. Fundamentals of Data Structures, E. Horowitz, S. Sahni, S. A-Freed, Universities Press.
2. Data Structures and Algorithms, A. V. Aho, J. E. Hopperoft, J. D. UIlman, Pearson.

Reference Books:

1. The Art of Computer Programming: Volume 1: Fundamental Algorithms, Donald E. Knuth.
2. Introduction to Algorithms, Thomas, H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press.
3. Open Data Structures: An Introduction (Open Paths to Enriched Learning), (Thirty First Edition), Pat Morin, UBC Press.

Supplementary Readings:

1. Aaron Tanenbaum, “Data Structures using C”, Pearson Education.
2. R. Gilberg, B. Forouzan, "Data Structures: A pseudo code approach with C", Cenage Learning, ISBN 9788131503140
3. R.G.Dromy, “How to Solve it by Computers”, Prentice Hall.

Learning Resources contd…

Web Resources:

Web links:

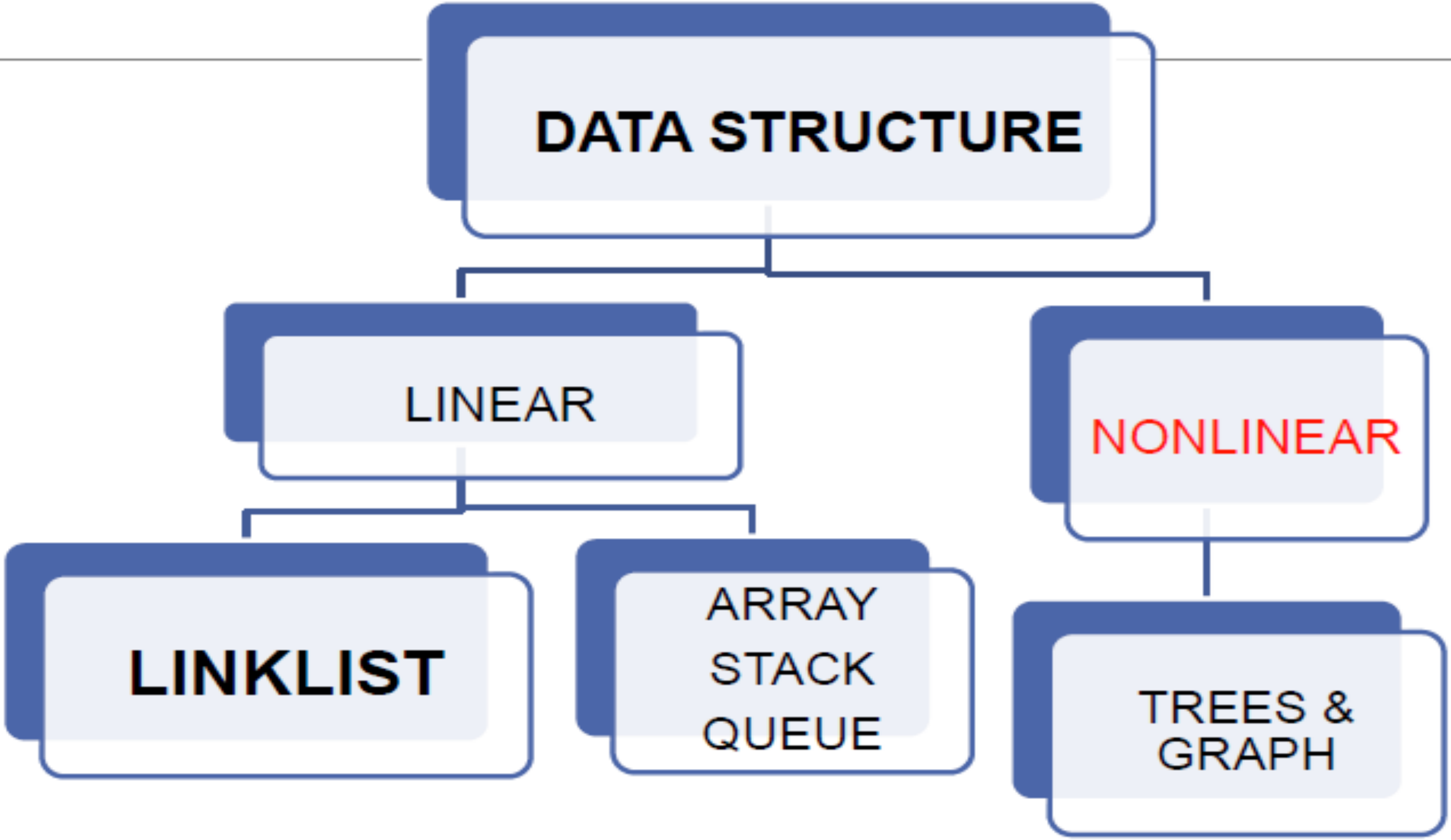
1. https://www.tutorialspoint.com/data_structures_algorithms/

MOOCs:

1. http://nptel.ac.in/courses/106102064/1

2. https://nptel.ac.in/courses/106103069/

Types of Data Structures


DATA STRUCTURE

LINEAR

NONLINEAR

LINKLIST

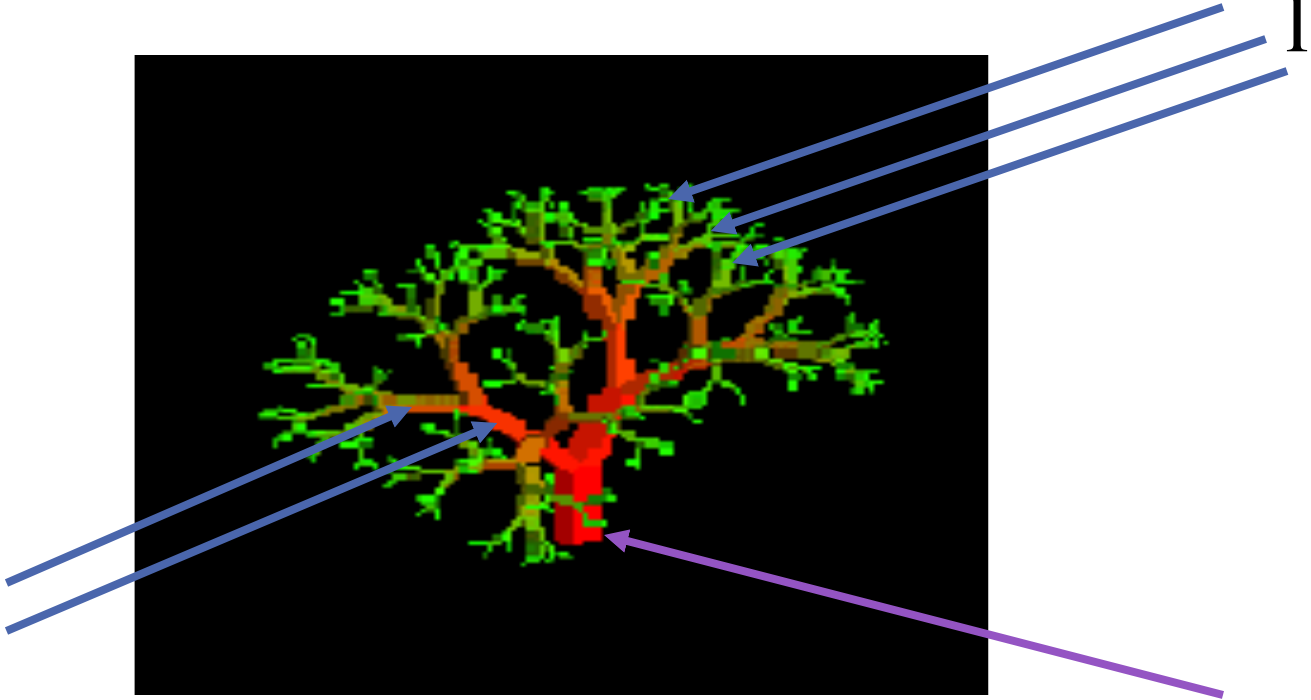
ARRAY
STACK
QUEUE

TREES &
GRAPH

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

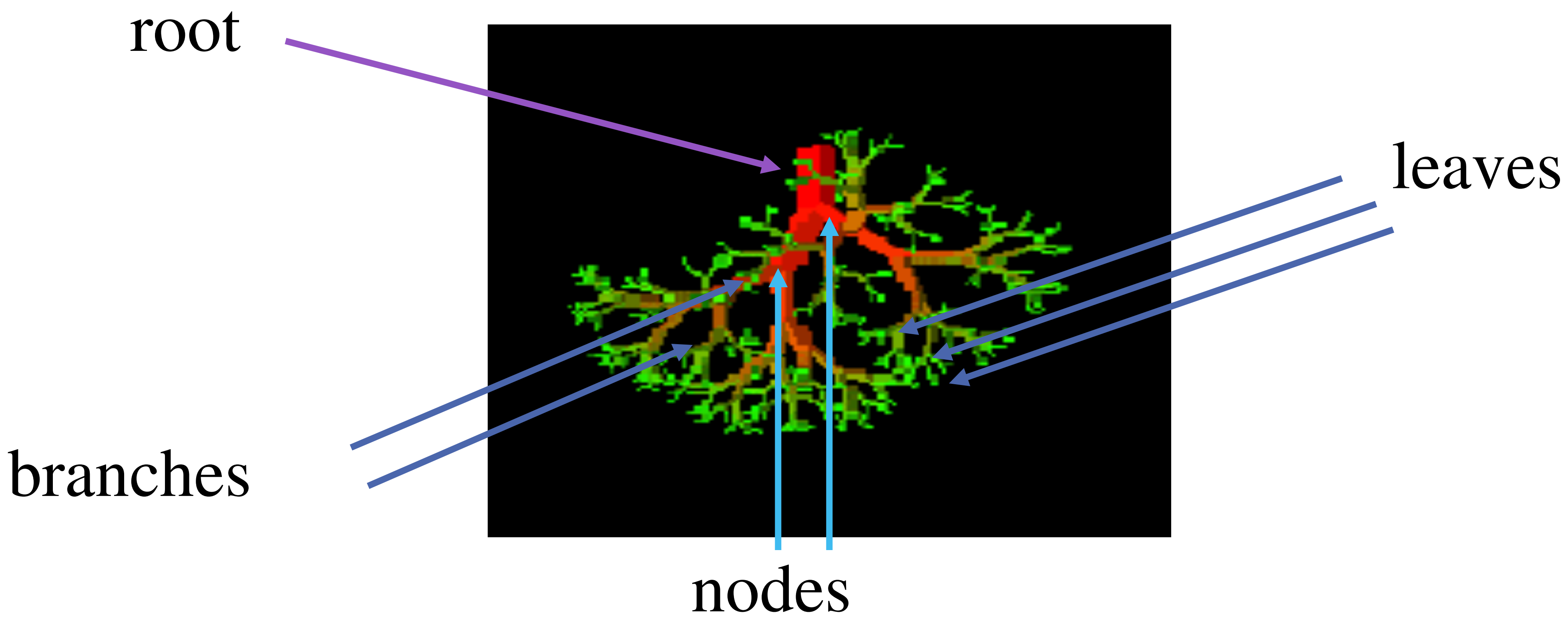


leaves

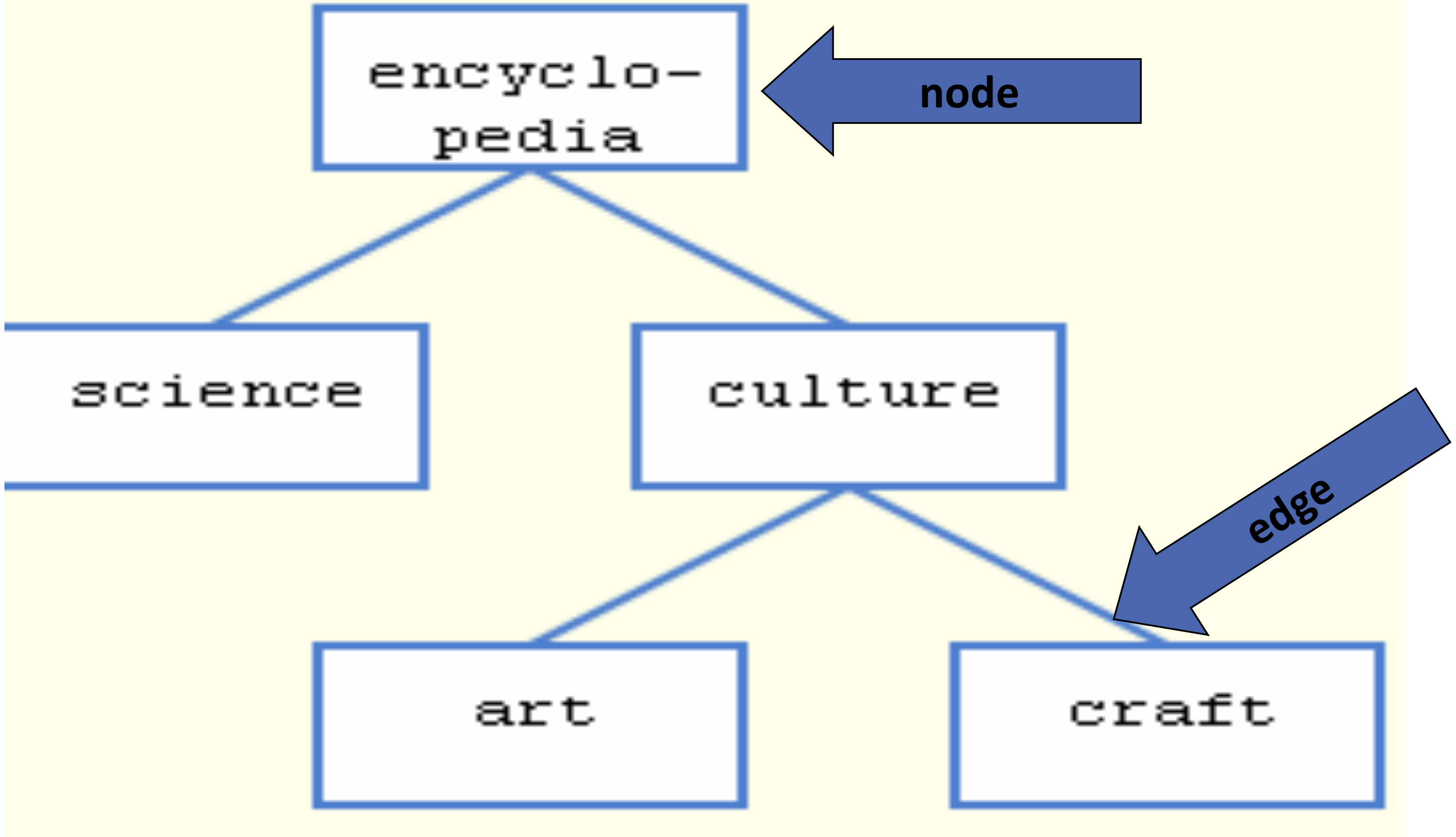
branches

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

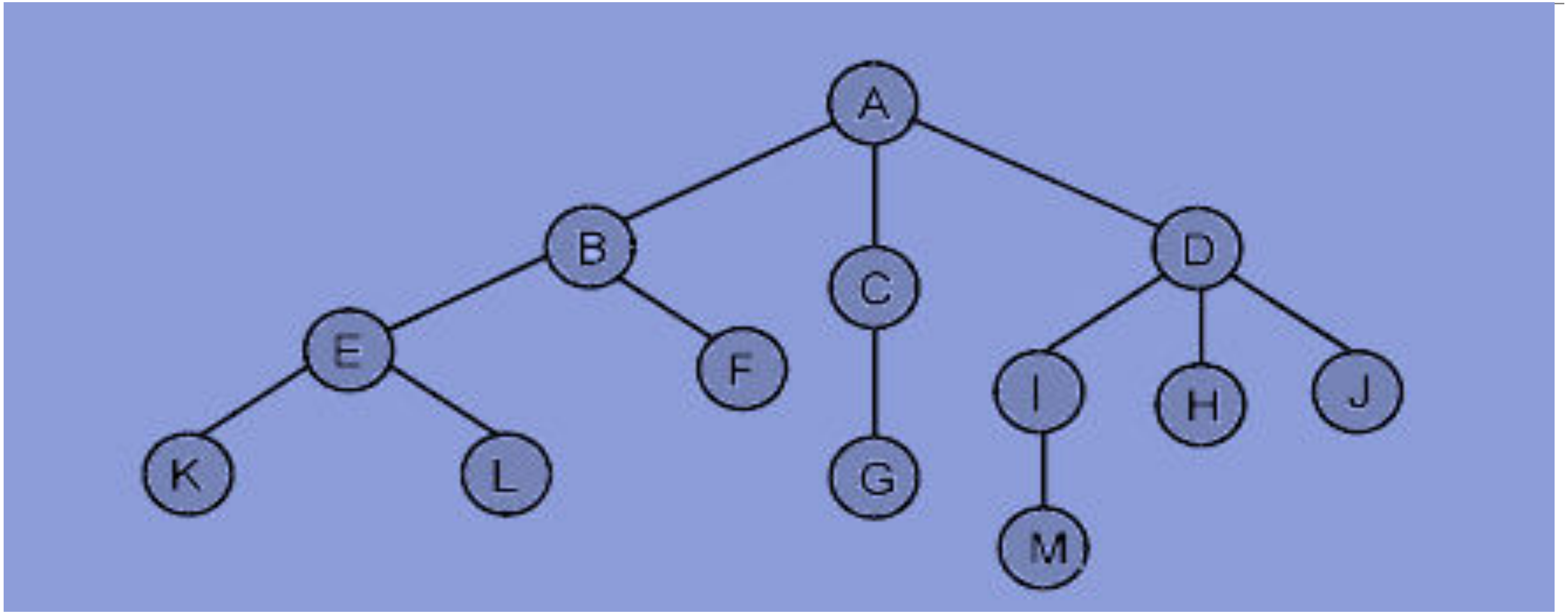


Figure 8: Sample Tree

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

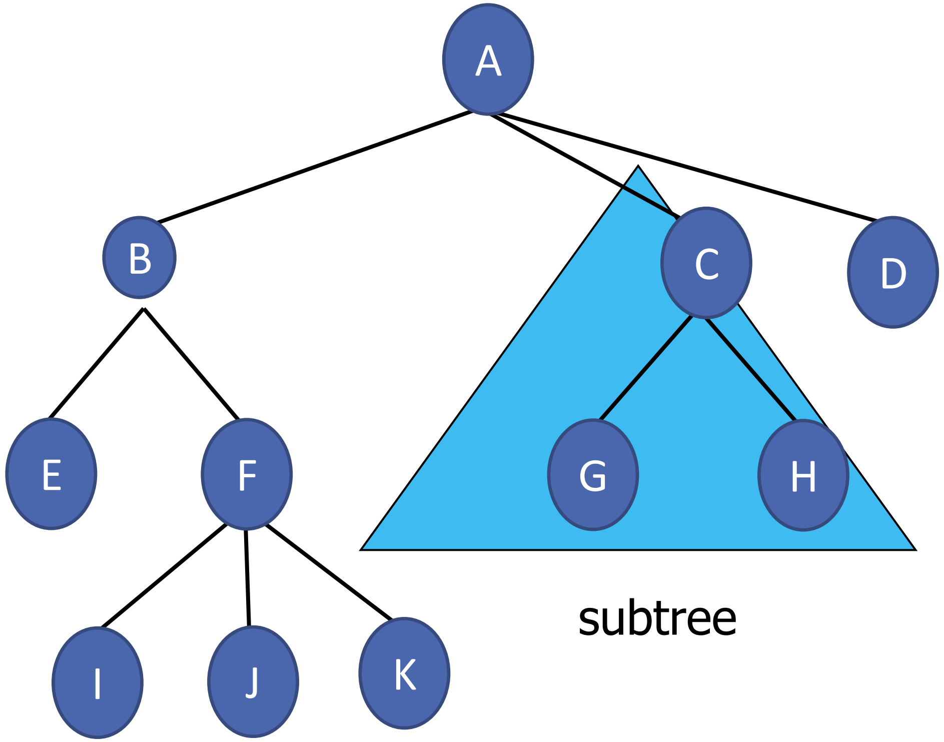
Degree of a tree: the maximum of the degree of the nodes in the tree.

Nonterminal nodes: other nodes

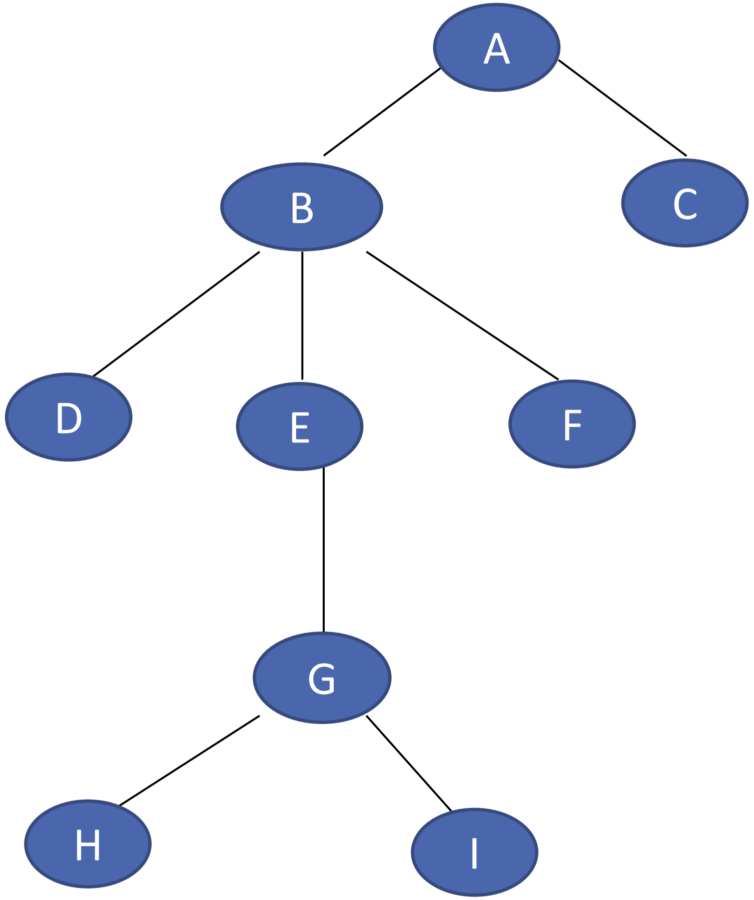
leaf or terminal node: Node that have degree zero (E, I, J, K, G, H, D)

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



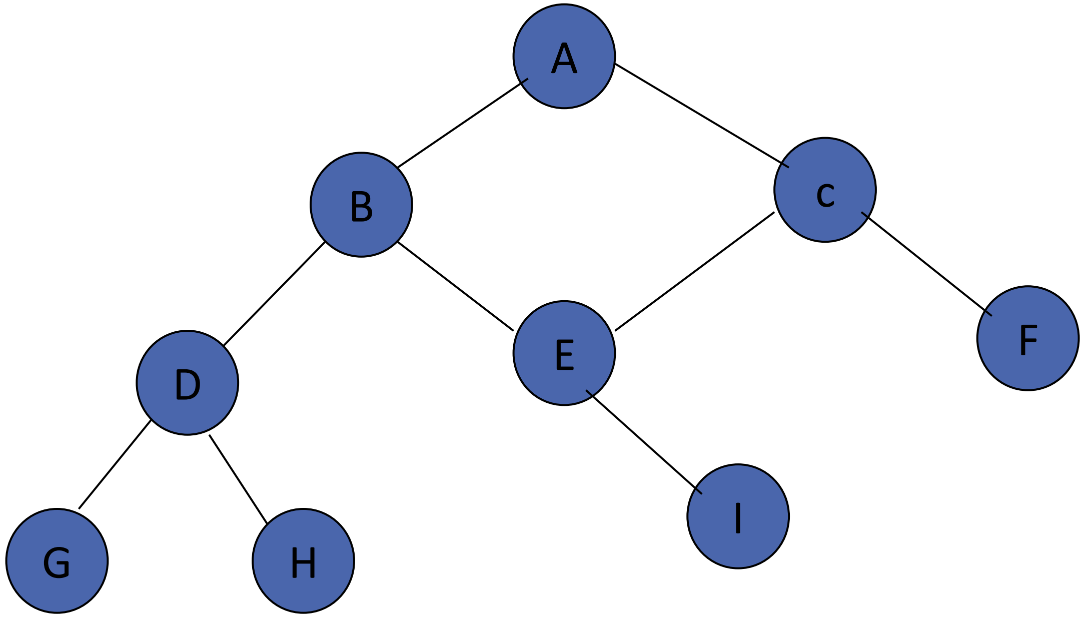
Property	**Value**
Number of nodes	9
Height	5
Root Node	A
Leaves	C,D,F,H,I
Interior nodes	B,E,G
Ancestors of H	A,B,E,G
Descendants of B	D,E,G,H,I,F
Siblings of E	D,F
Right subtree of A	A,C
Degree of this tree	3

Binary Tree

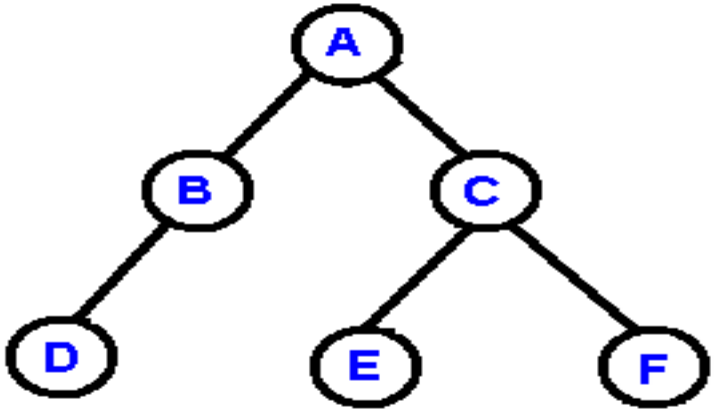
- Every node in a binary tree can have at most two children.

- 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.

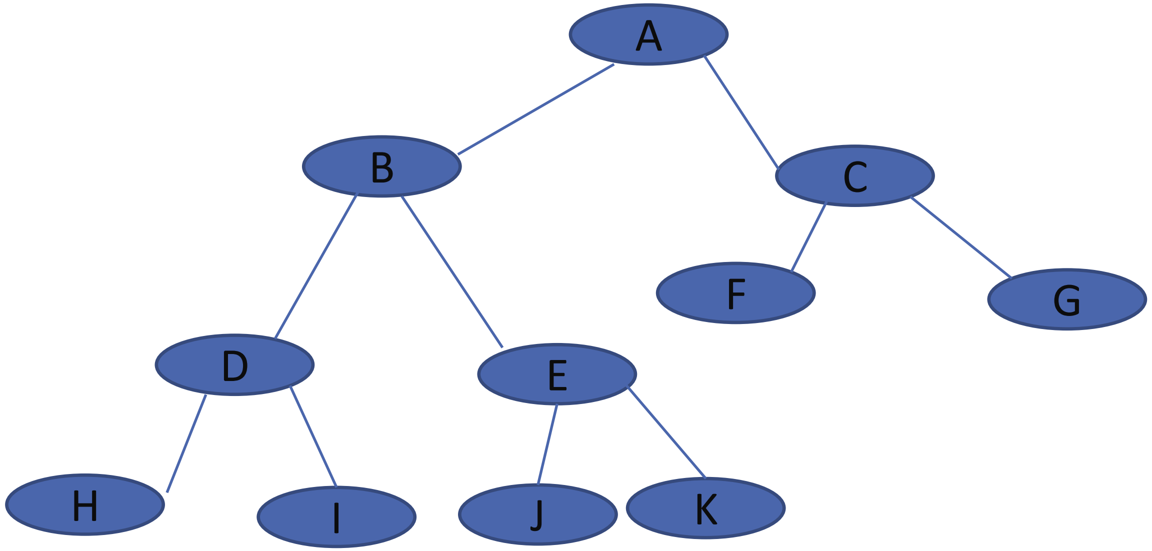
Binary Trees

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

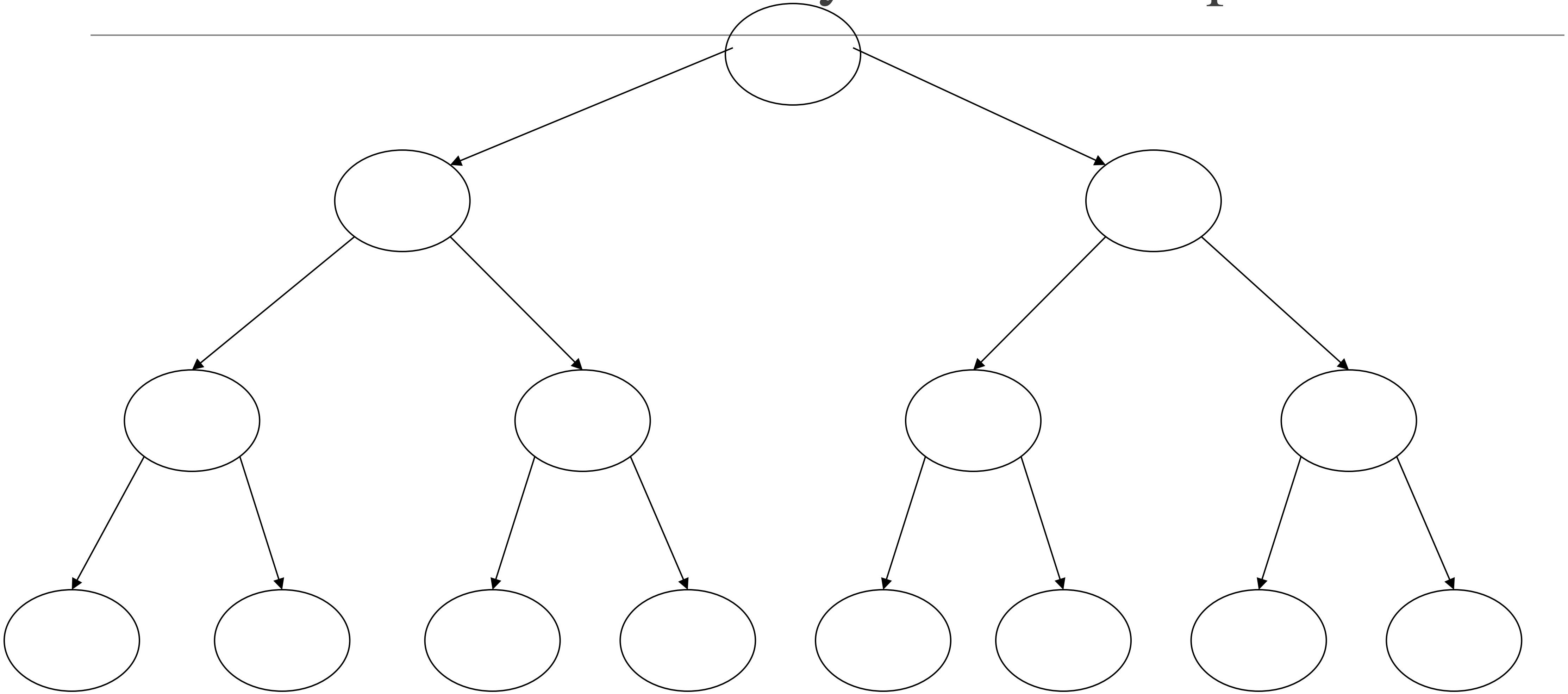
- **Complete Binary Tree**

  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



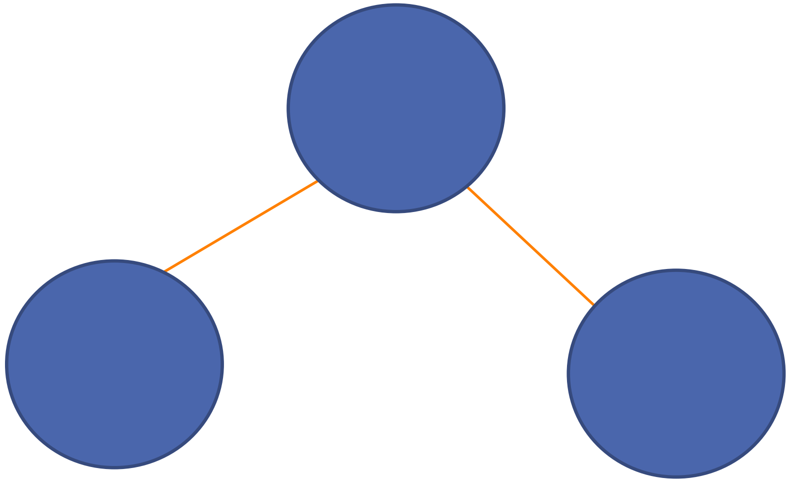
A Full Binary Tree - Example



Complete Binary Trees

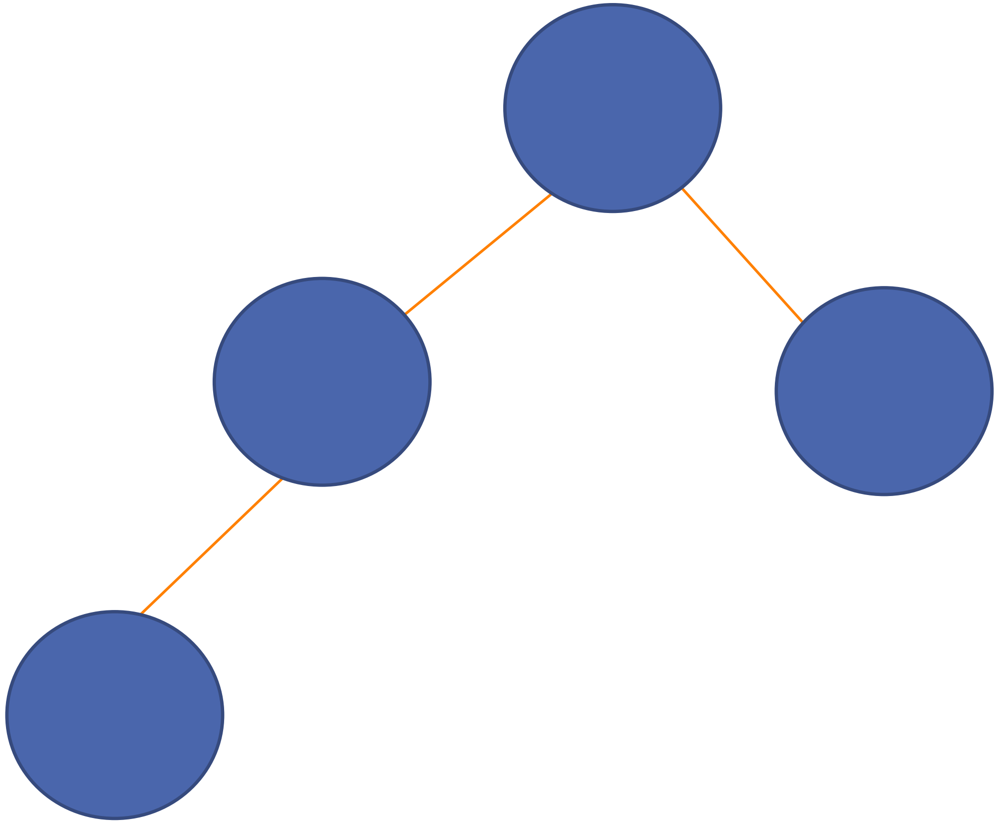
The second node of a complete binary tree is always the left child of the root...

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



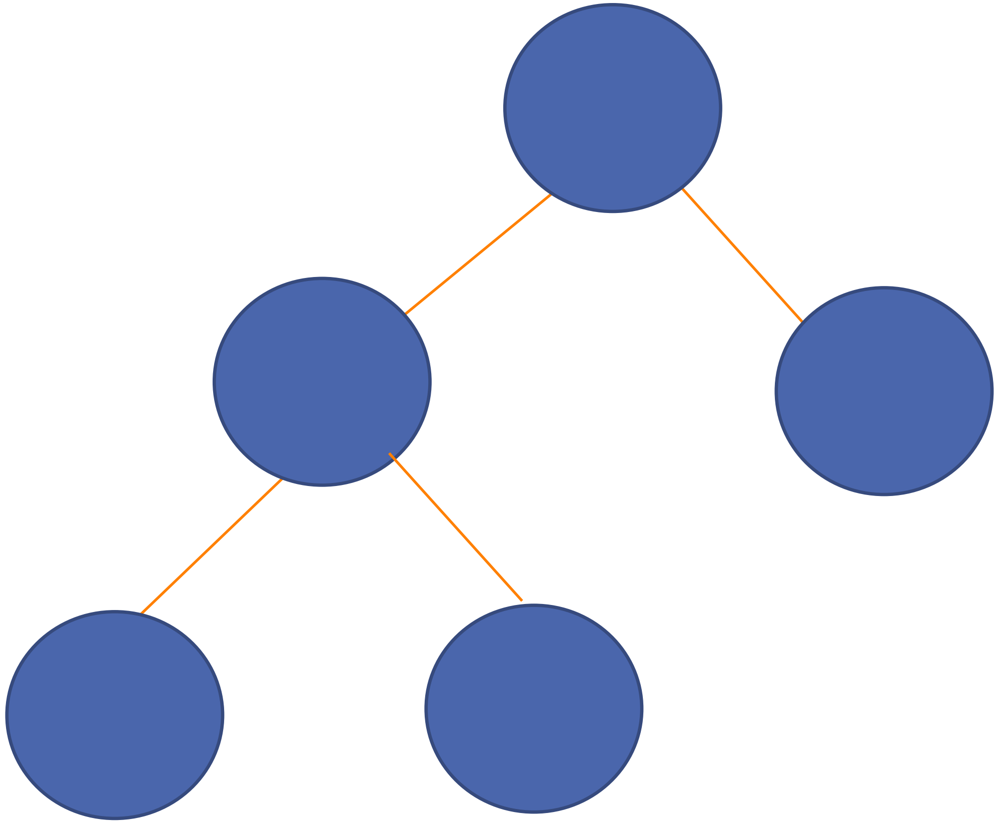
Complete Binary Trees

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



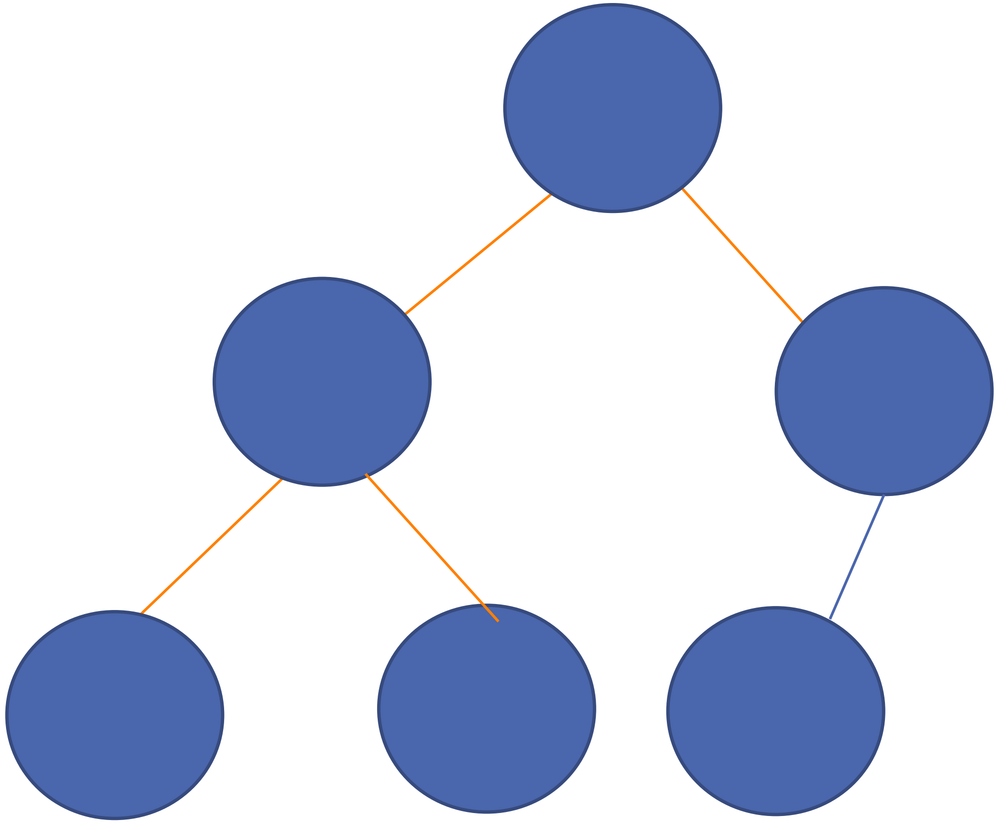
Complete Binary Trees

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



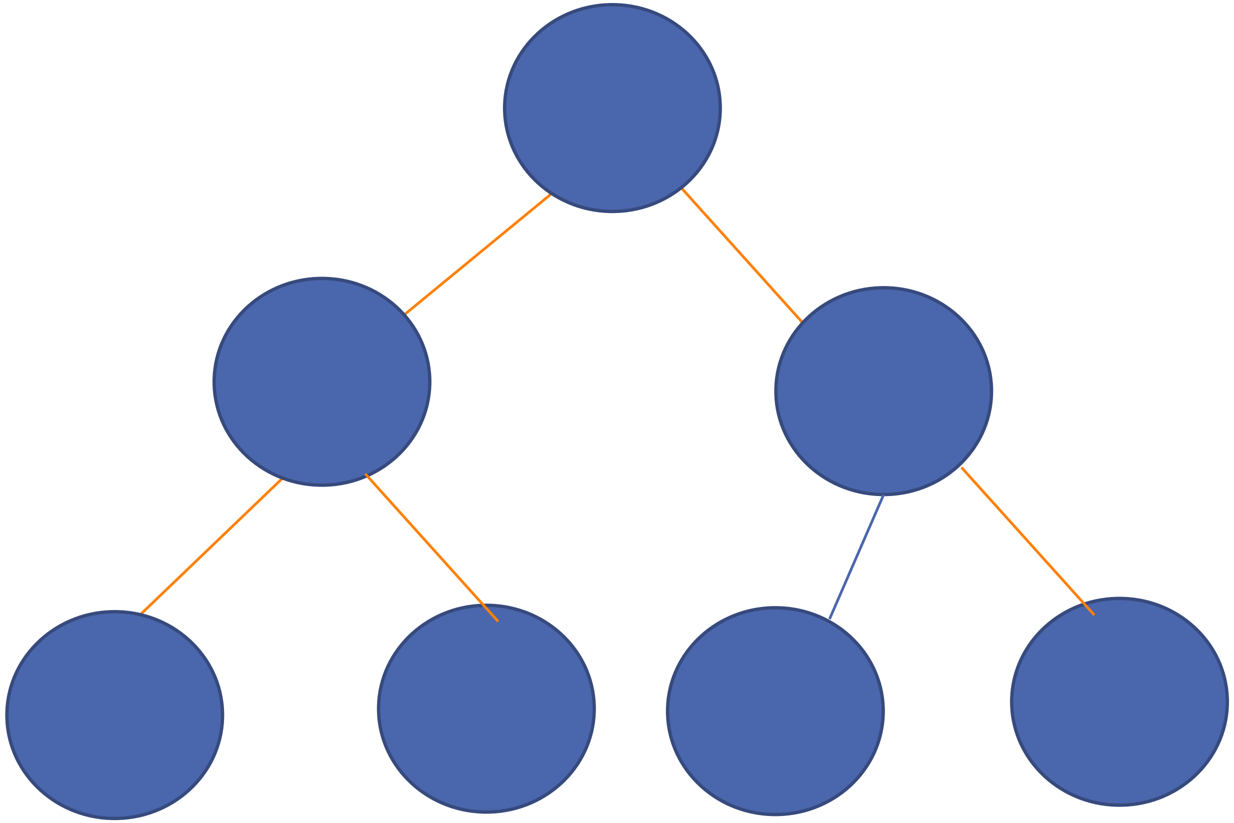
Complete Binary Trees

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



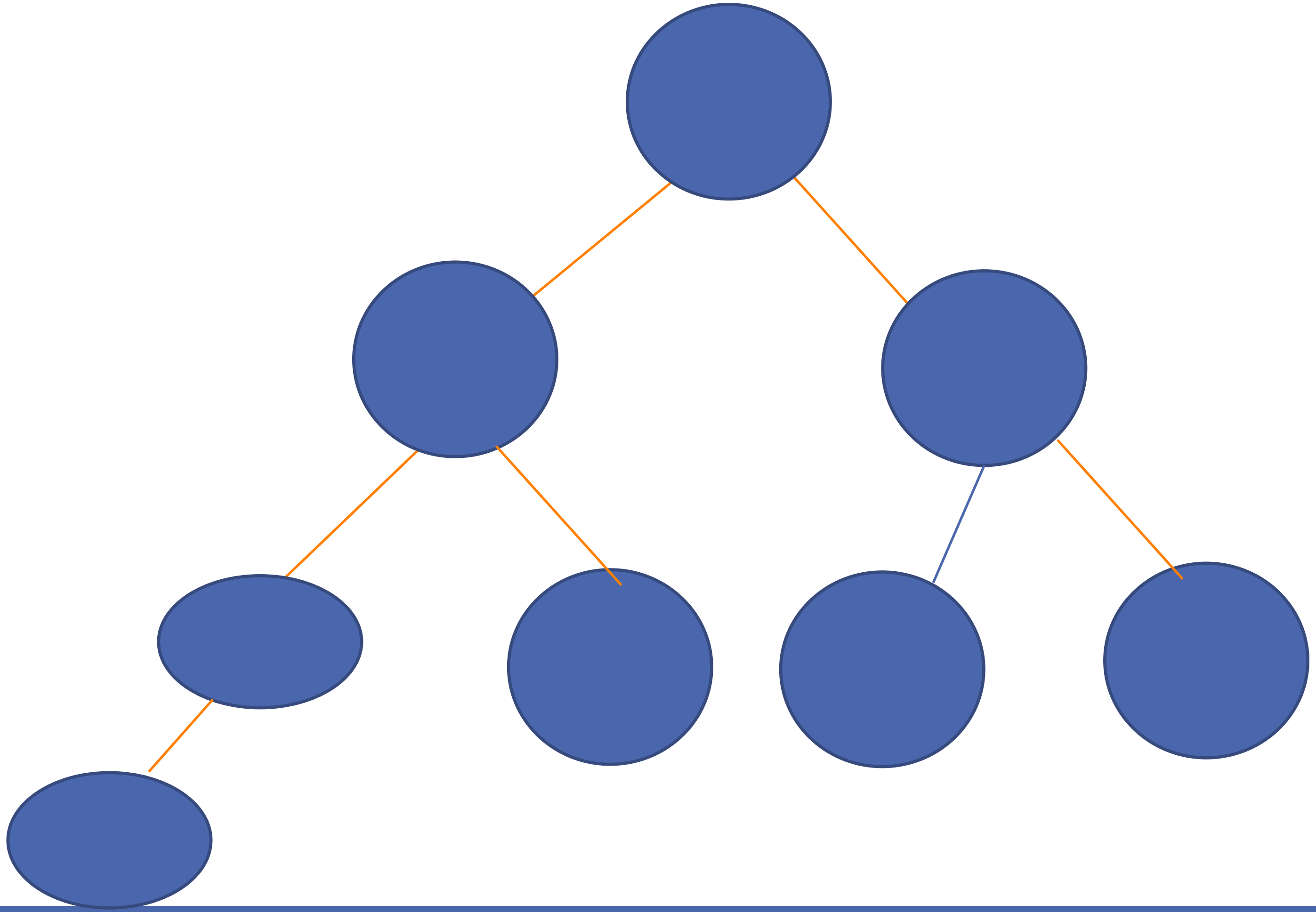
Complete Binary Trees

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



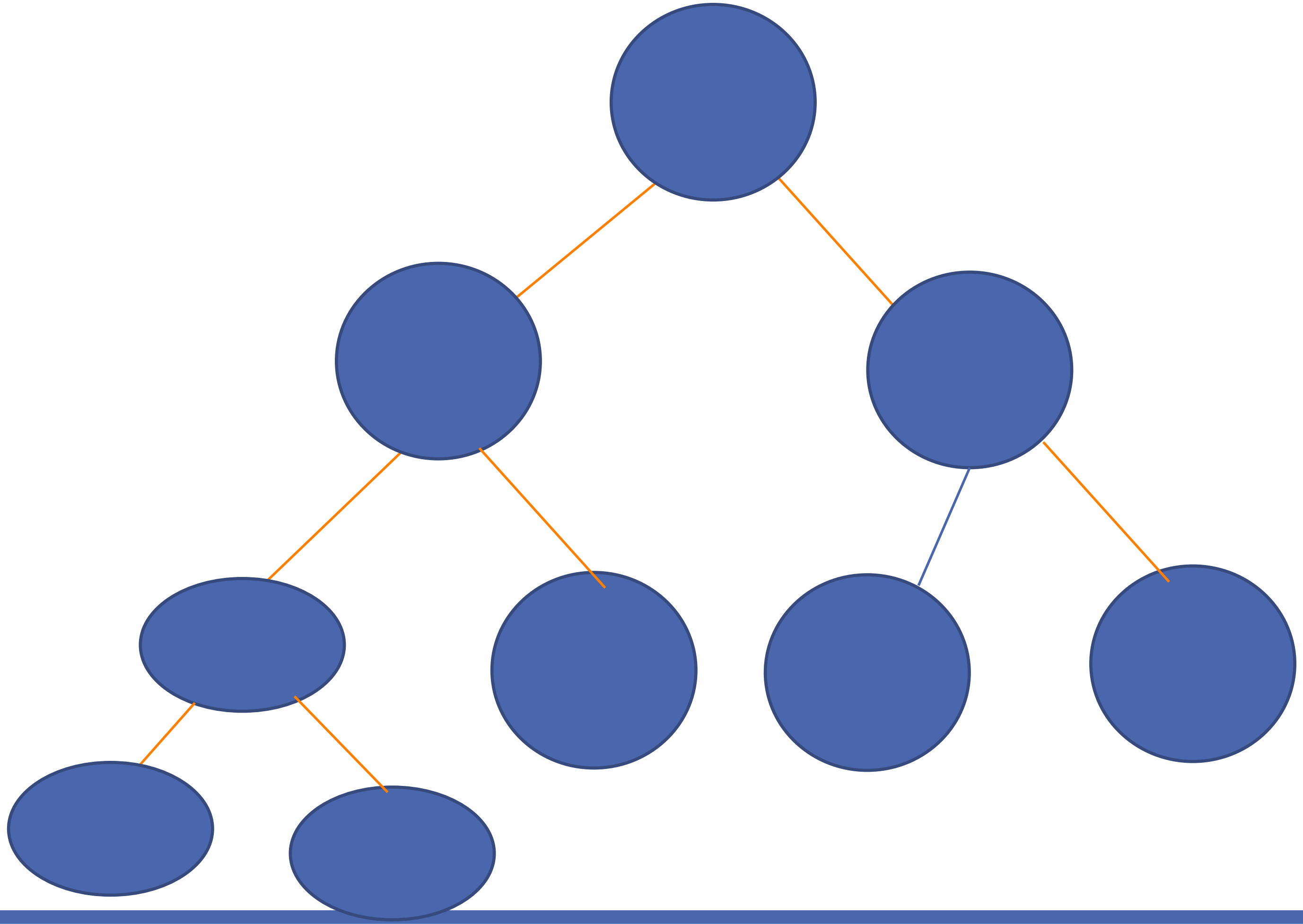
Complete Binary Trees

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

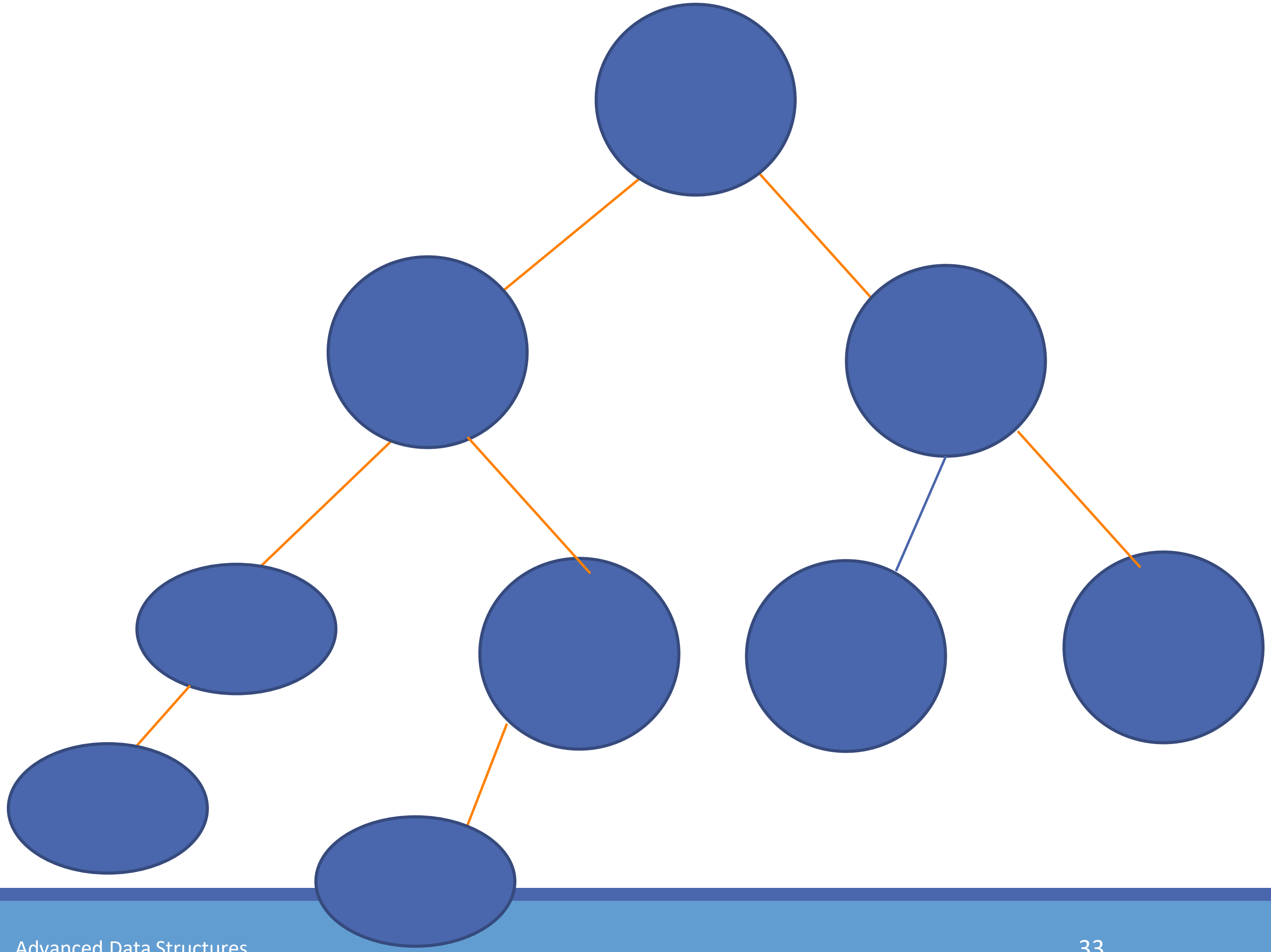


Complete Binary Trees

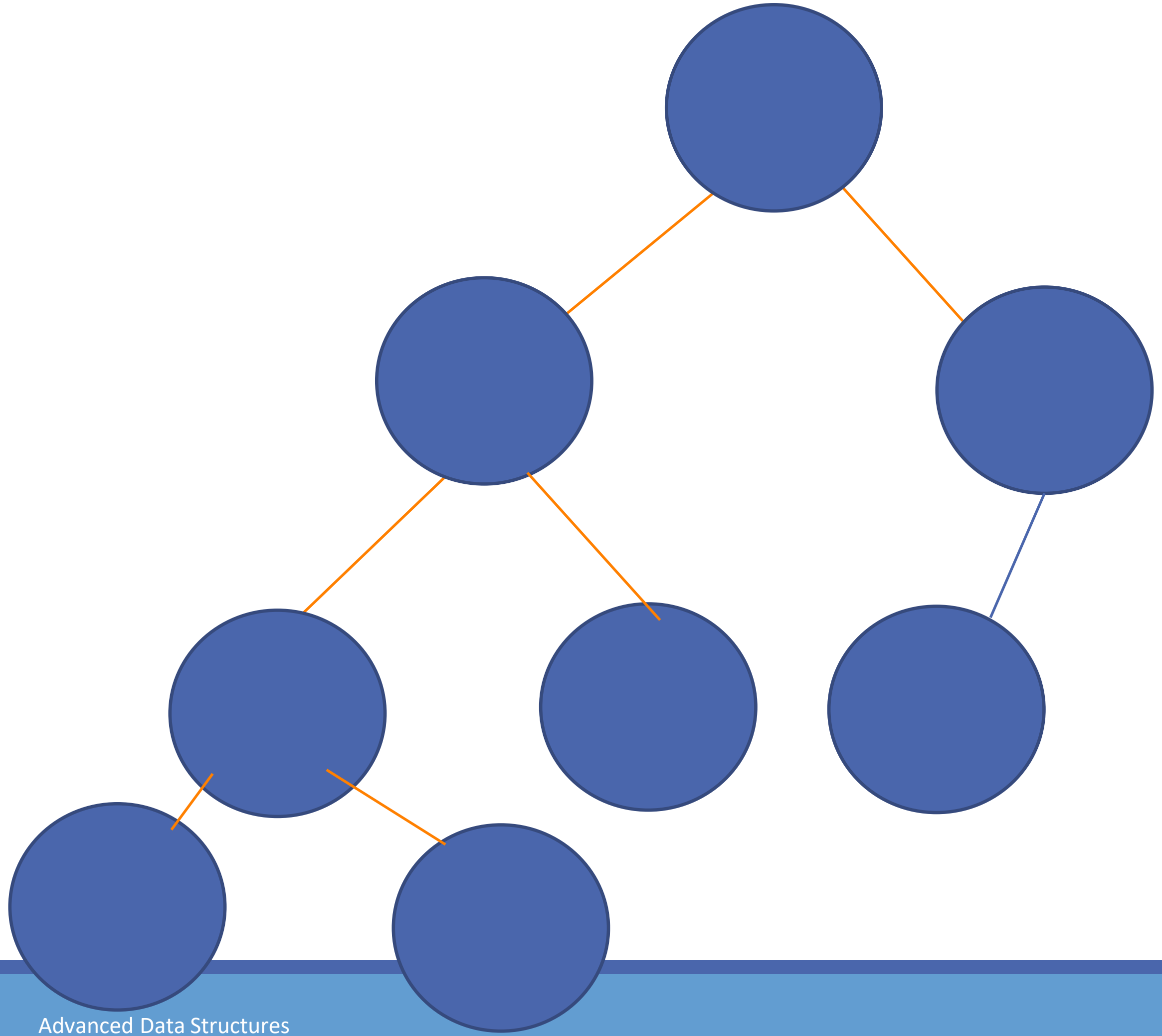
The next nodes must always fill the next level from left to right.



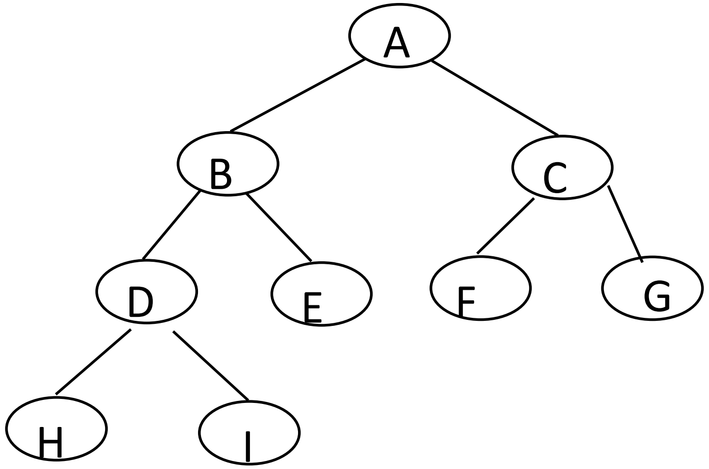
Is This Complete?



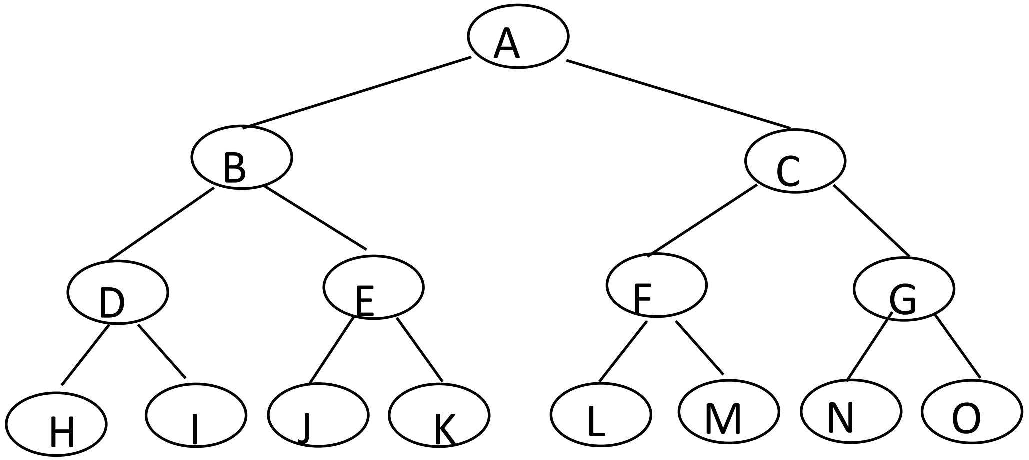
Is This Complete?



Full BT VS Complete BT

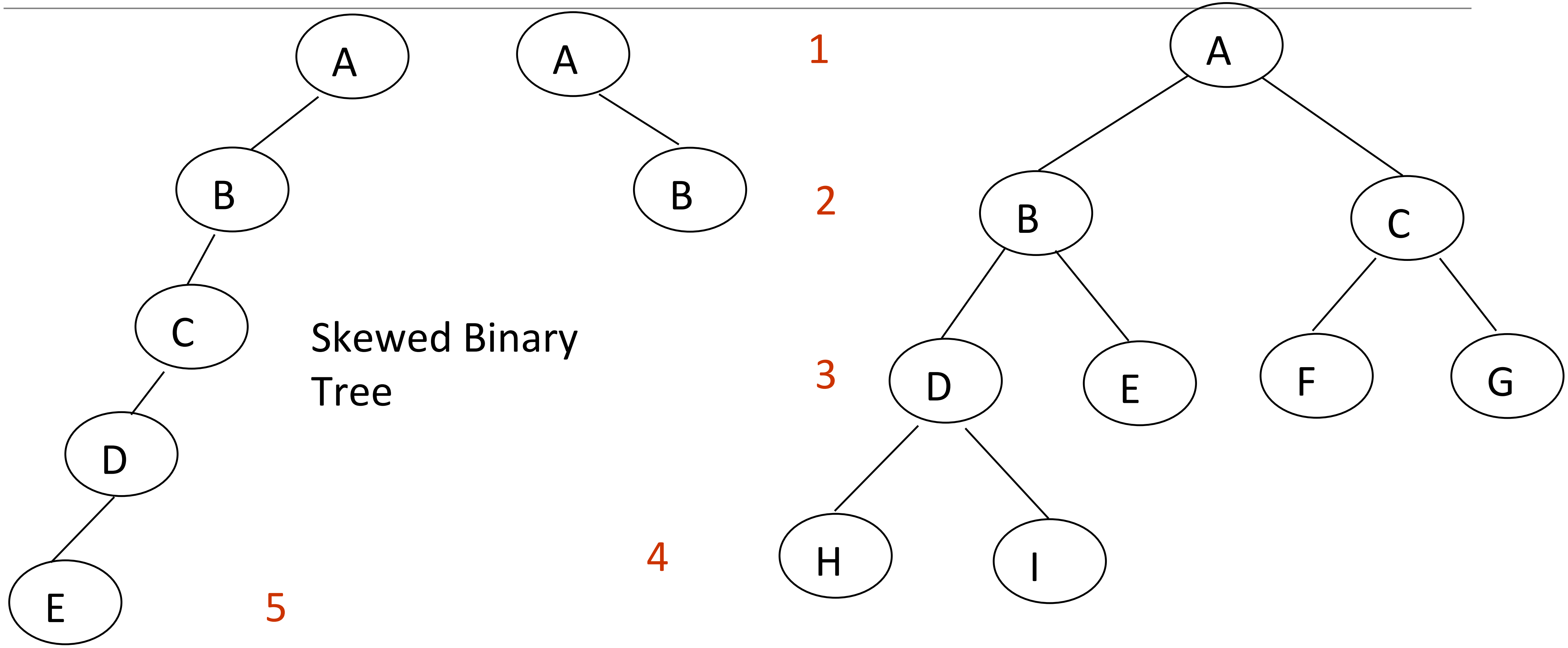


Complete binary tree



Full binary tree of depth 4

Samples of Trees



Skewed Binary Tree

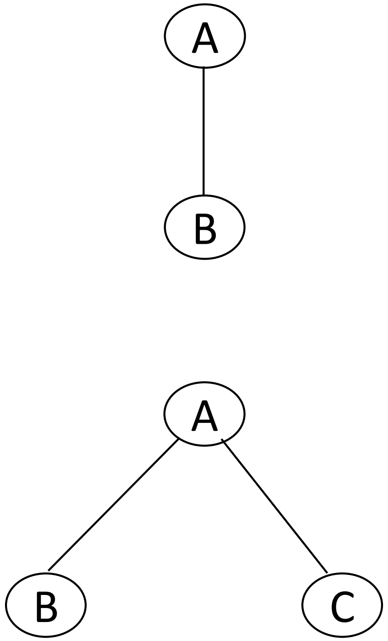
Complete Binary Tree

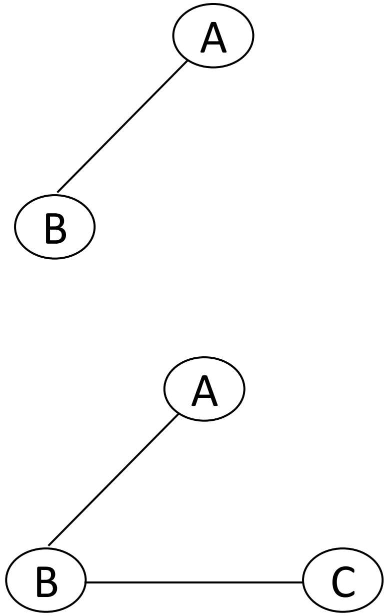
Converting tree to binary tree

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

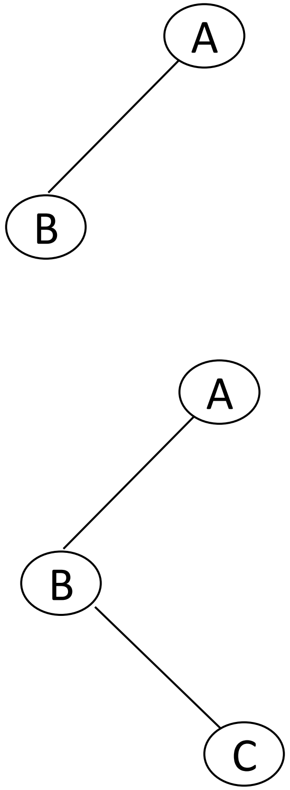
- The left subtree and the right subtree are distinguished.

Tree Representations





Left child-right sibling

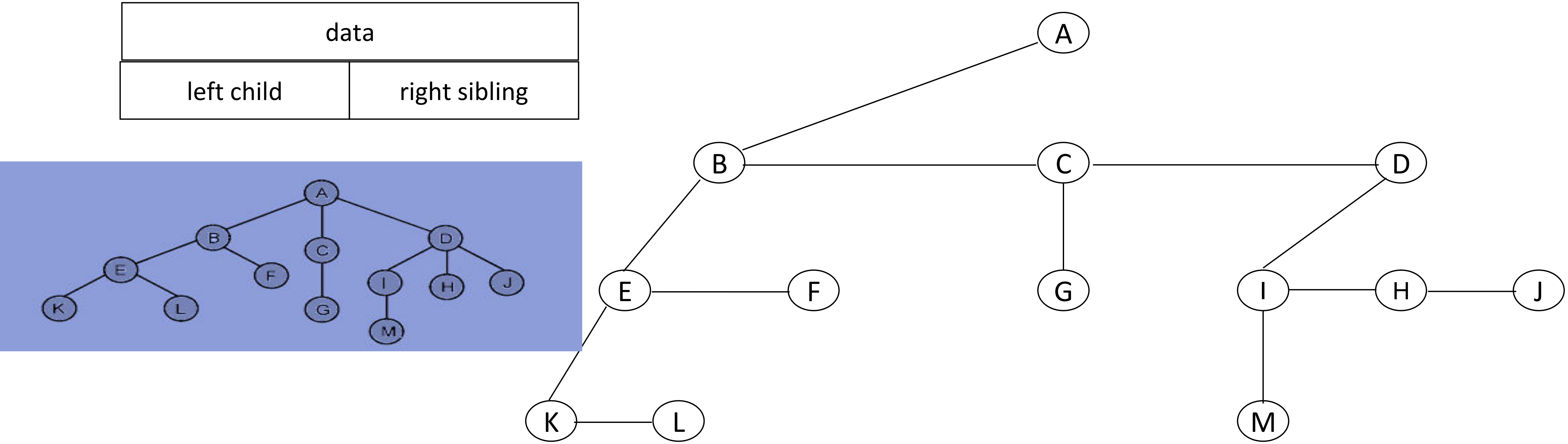


Binary tree

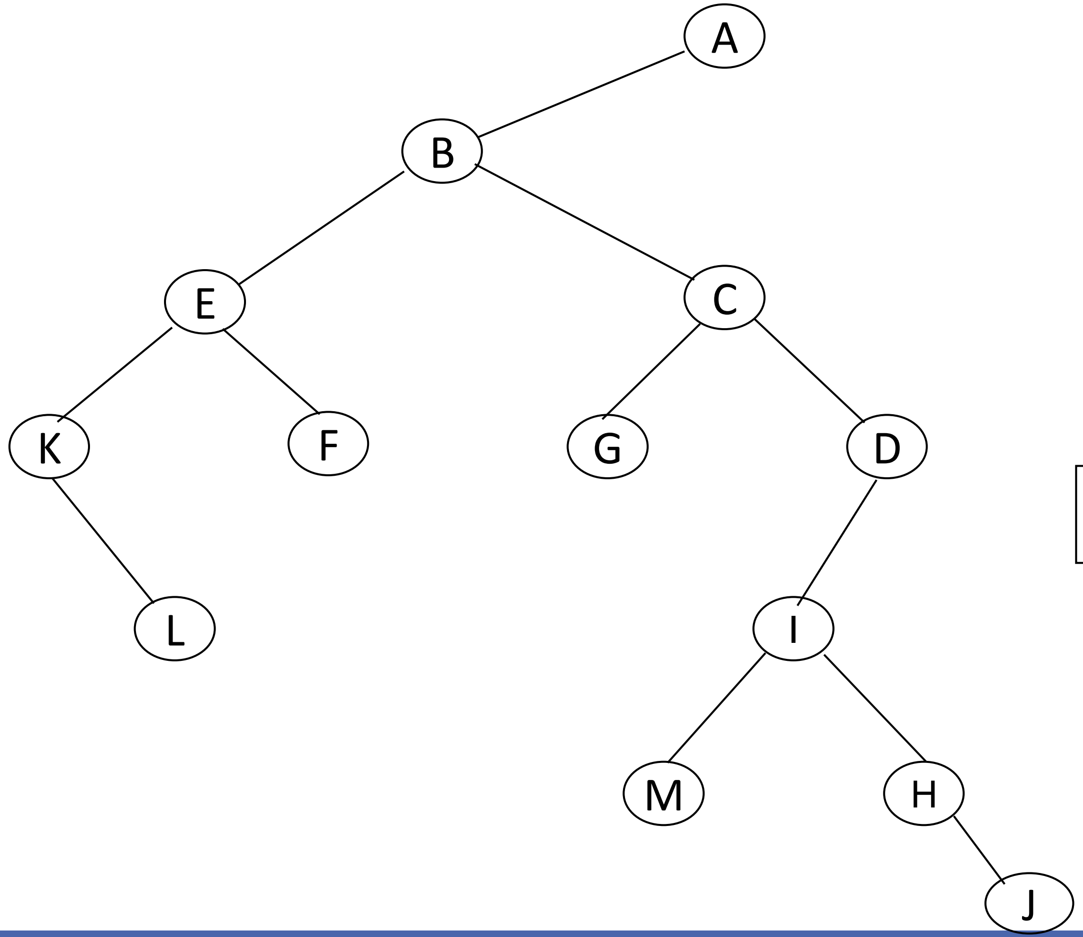
Representation of Trees

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

data	
left child	right sibling



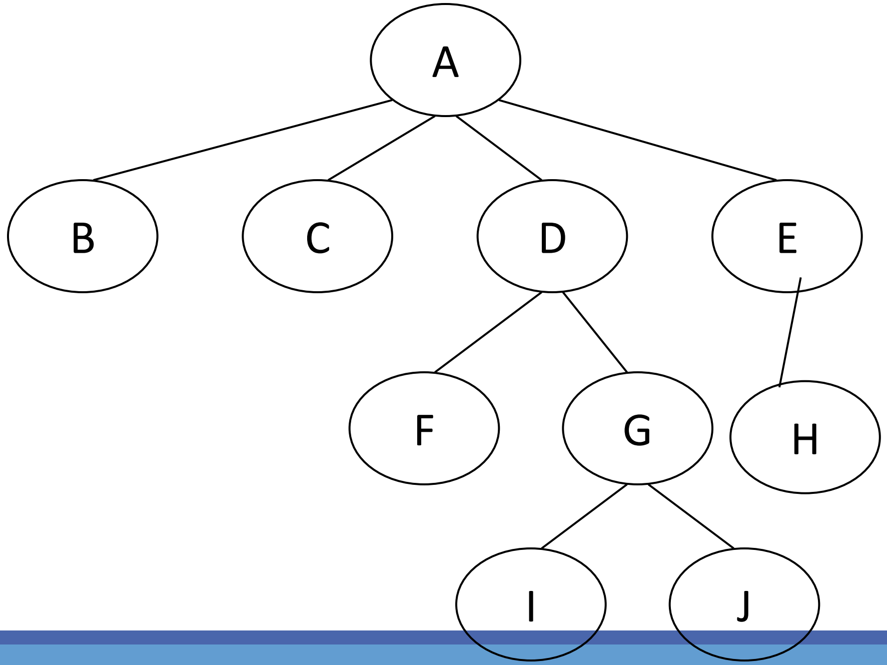
Degree Two Tree Representation

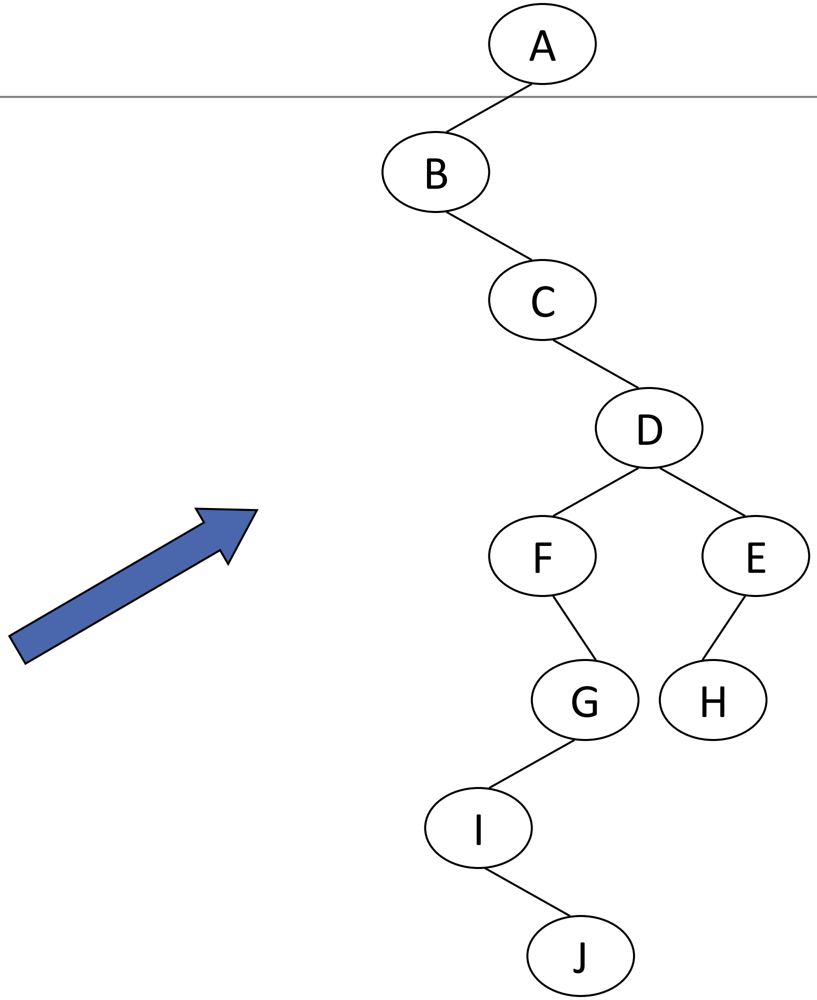


Binary Tree!

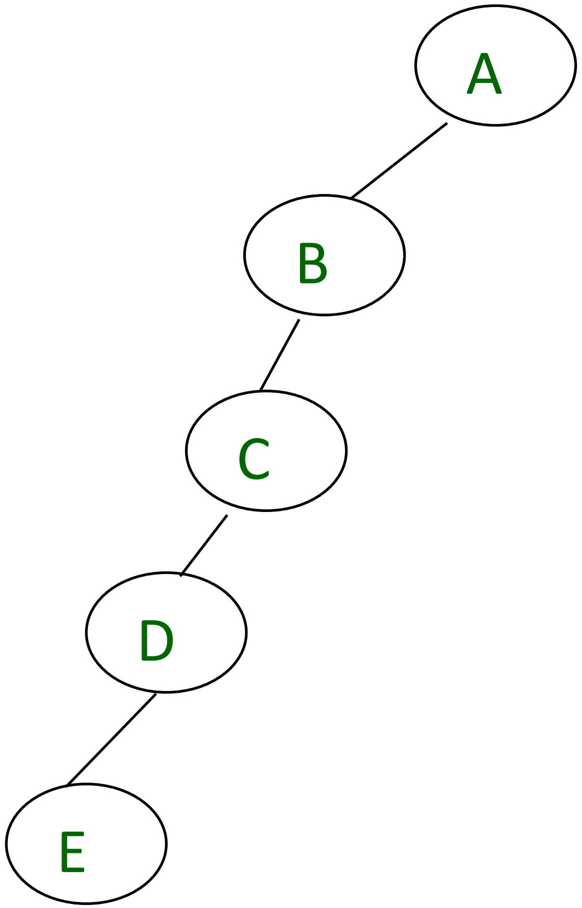
Converting to a Binary Tree

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





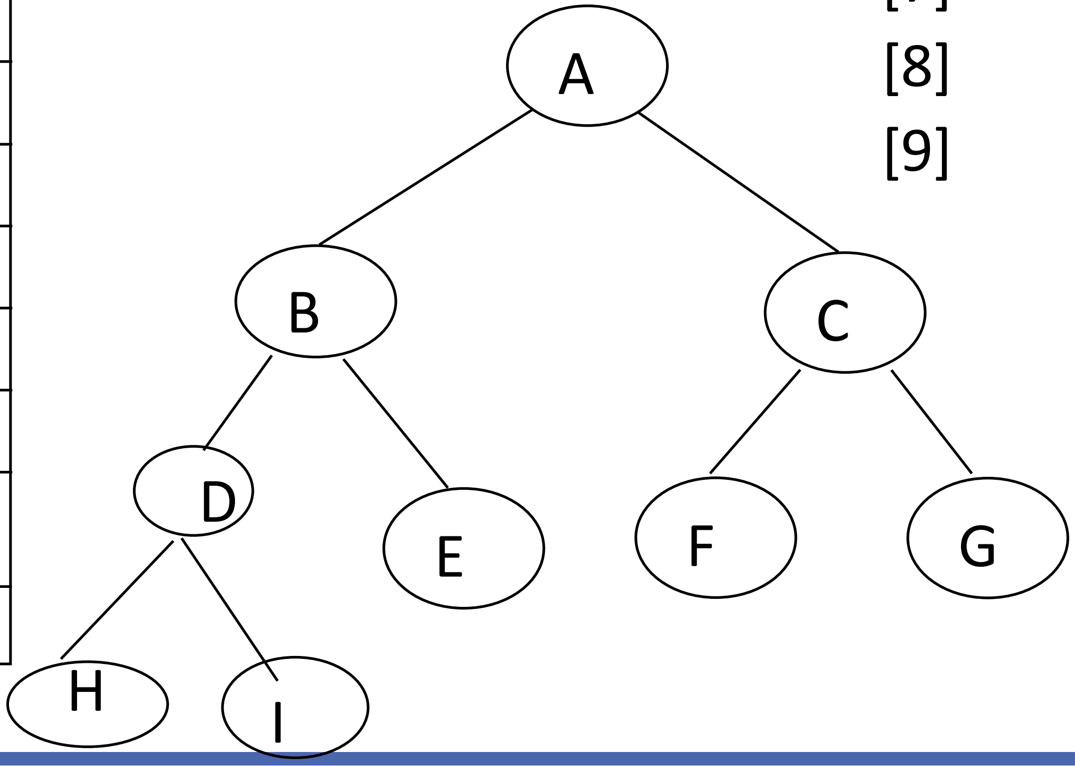
Sequential Representation



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

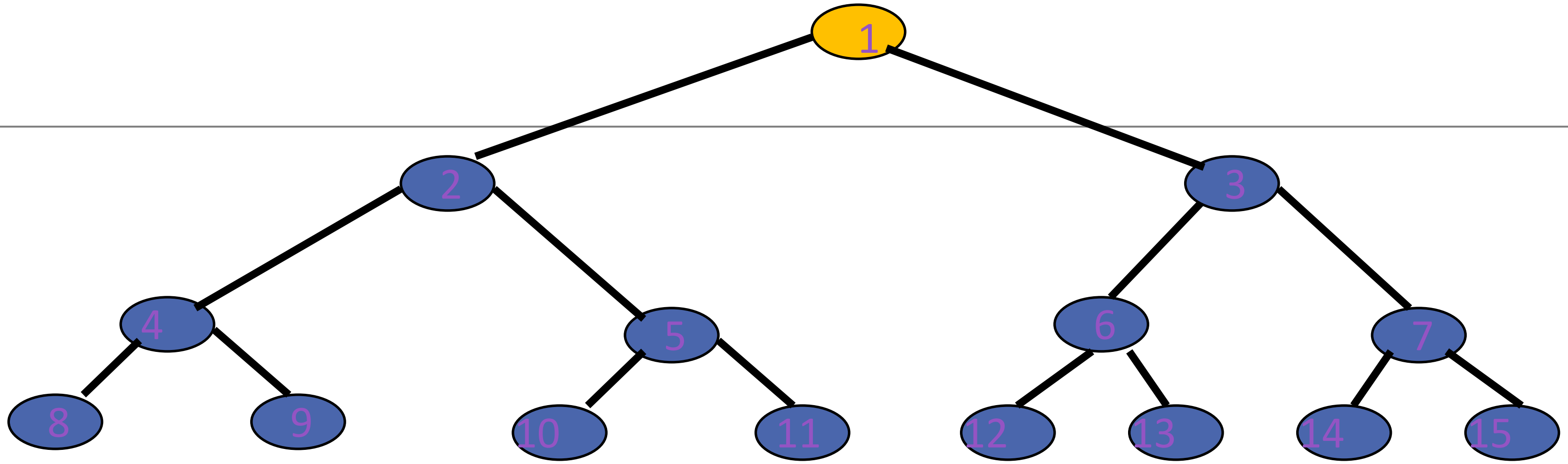
(1) waste space

(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

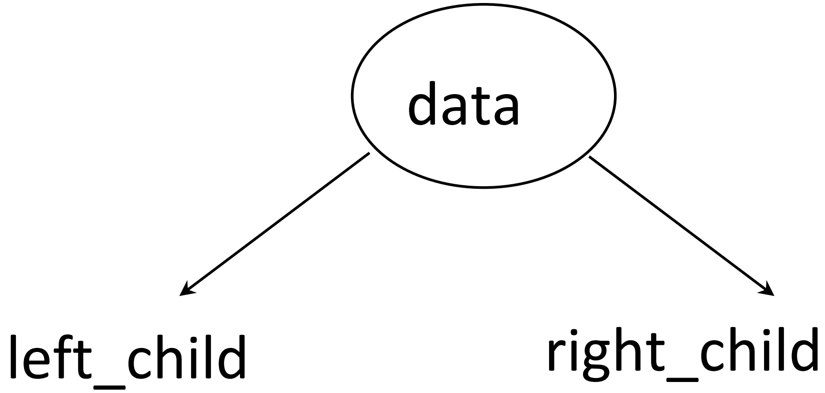
Right child of node i is node 2i+1 if 2i+1 is <=n

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

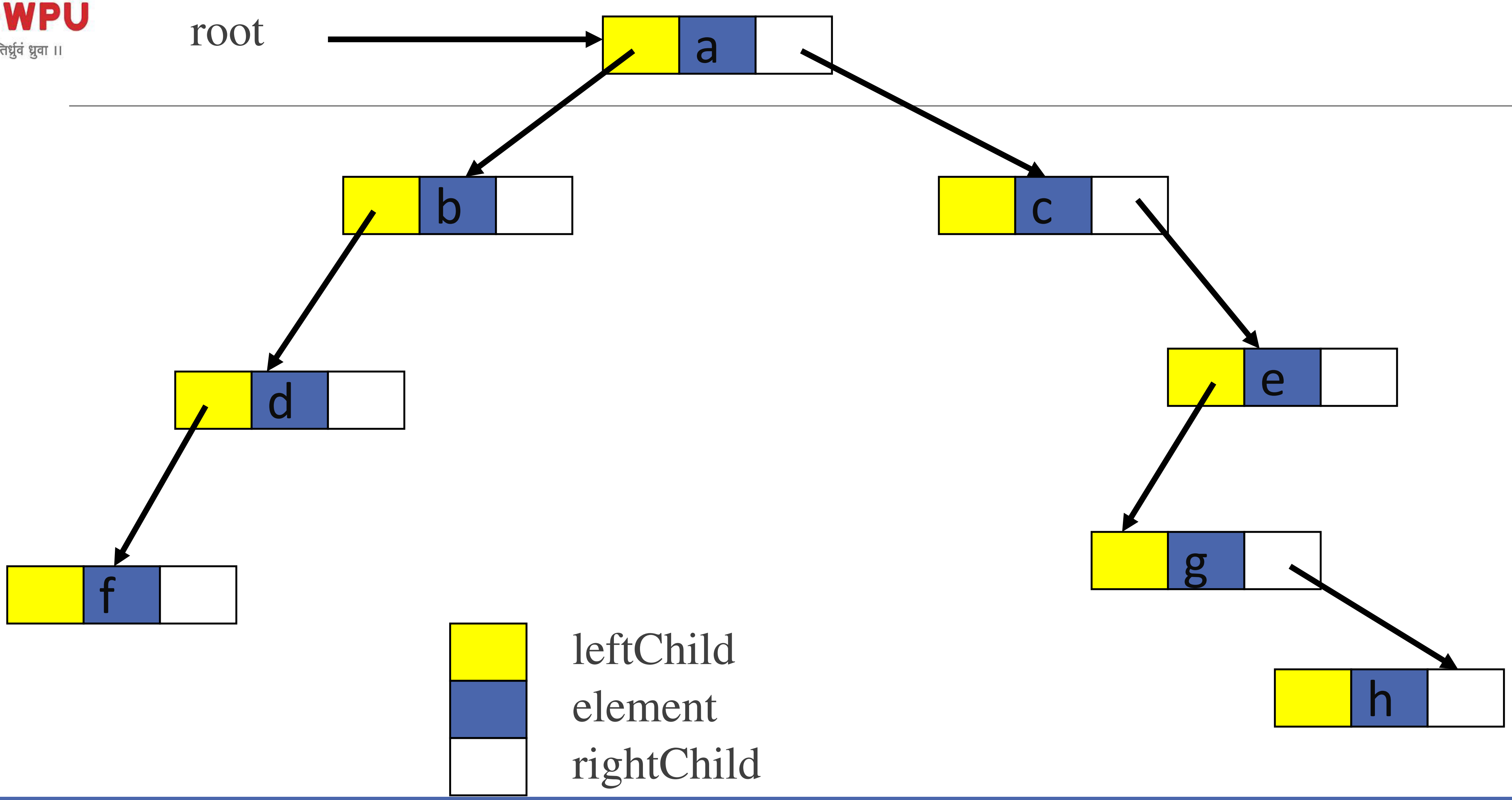
Linked Representation

```
class node{
   int data;
   node *lchild;
   node *rchild;
};
```

left_child	data	right_child



Linked Representation Example



Binary Tree Creation

```
class treenode
{
    char data[10];
    treenode *left;
    treenode *right;
    friend class tree;
}
class tree
{
    treenode *root;
    public:
      tree();
       void create_r();
       void create_r(treenode *);
  }
```

```
Algorithm create_r()     //Driver for creation
{
    Allocate memory for root and accept data;
     create_r(root);
 }
```

```
int main()
{
  tree bt;
  bt.create_r();
 }
```

```
tree::tree()      //constructor
{
   root=NULL;
 }
```

Algorithm create_r(treenode * temp)   //workhorse for creation

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

```
Algorithm create_nr()
{
   if root=NULL
   {
     Allocate memory for root and accept the data;      }
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.

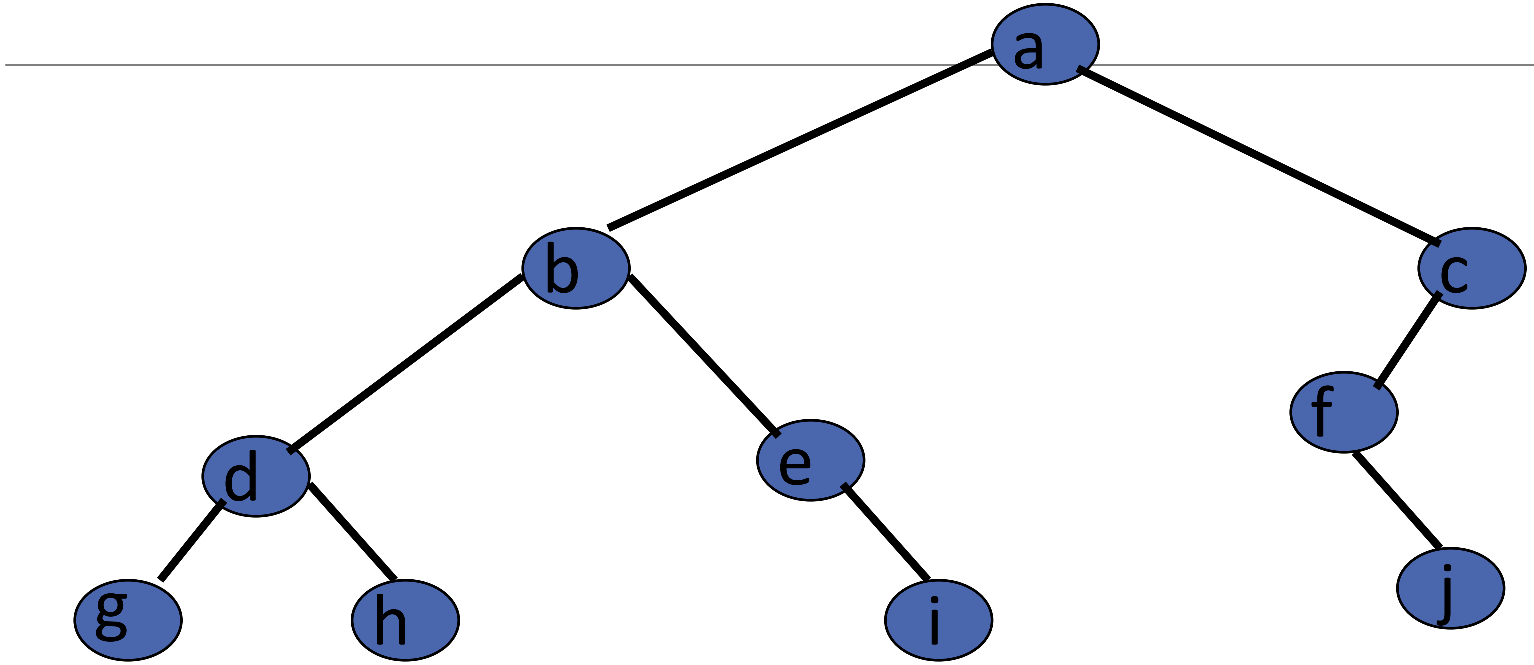
- Usually in a left to right fashion

- This is implemented with a queue

Depth First

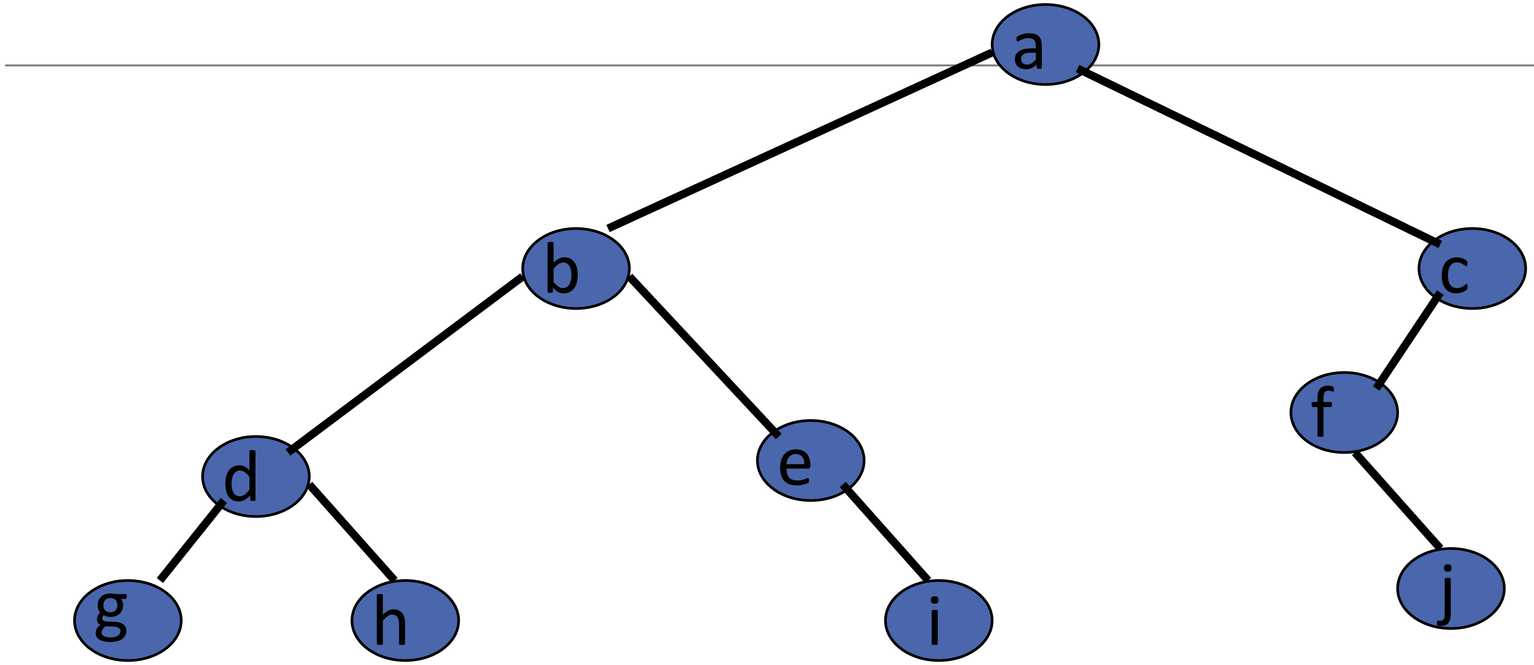
- In a depth first traversal all the nodes on a branch are visited before any others are visited
- There are three common depth first traversals
  - Inorder
  - Preorder
  - Postorder
- Each type has its use and specific application

Inorder Example (Visit = print)



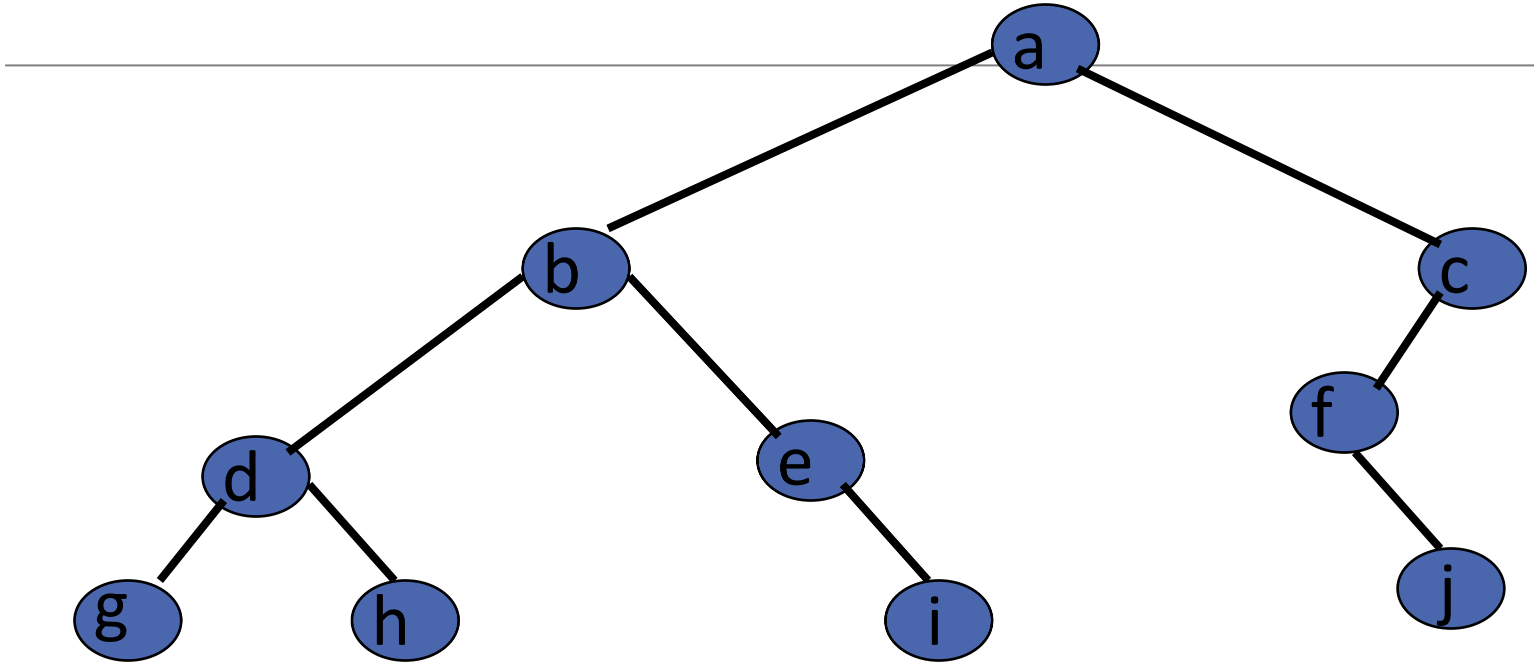
g d h b e i a f j c

Preorder Example (Visit = print)



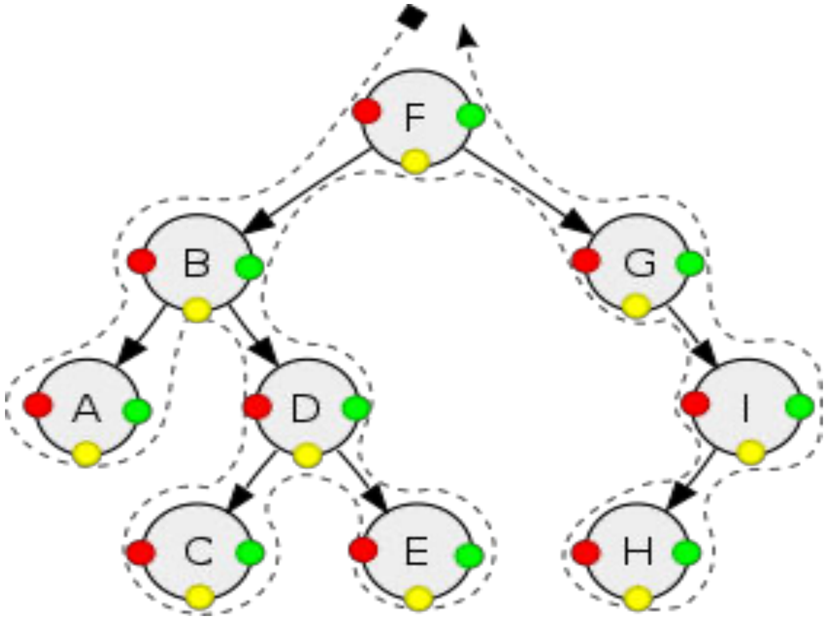
a b d g h e i c f j

Postorder Example (Visit = print)



g h d i e b j f c a

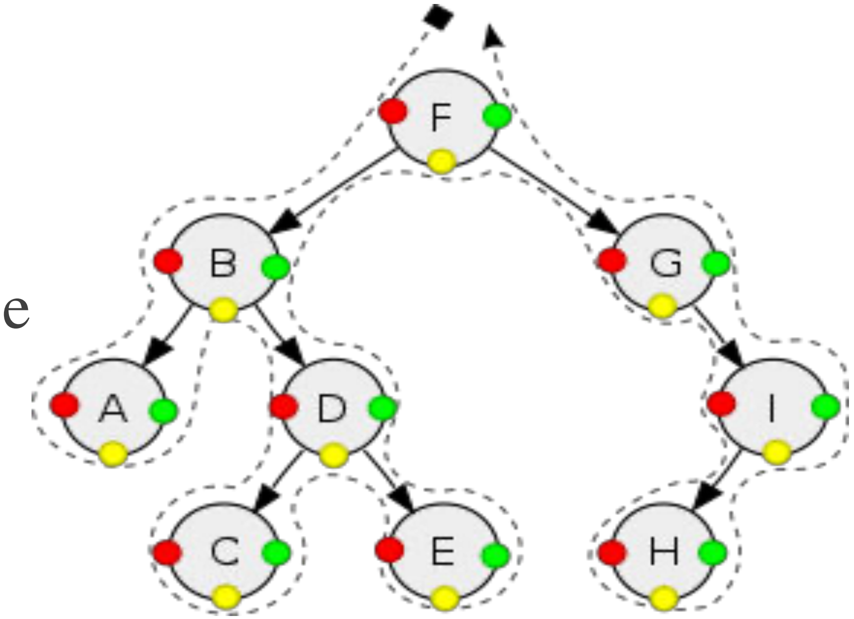
Depth first traversal



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.

Inorder Traversal (recursive version)

```
Algorithm inorder_r()    //Driver
{
  inorder_r(root);
}
 Algorithm inorder_r(treenode *temp)    // Workhorse
{
  if temp!=NULL
  {
    inorder_r(temp->left);
    Print temp->data;
    inorder_r(temp->right);
  }
}
```



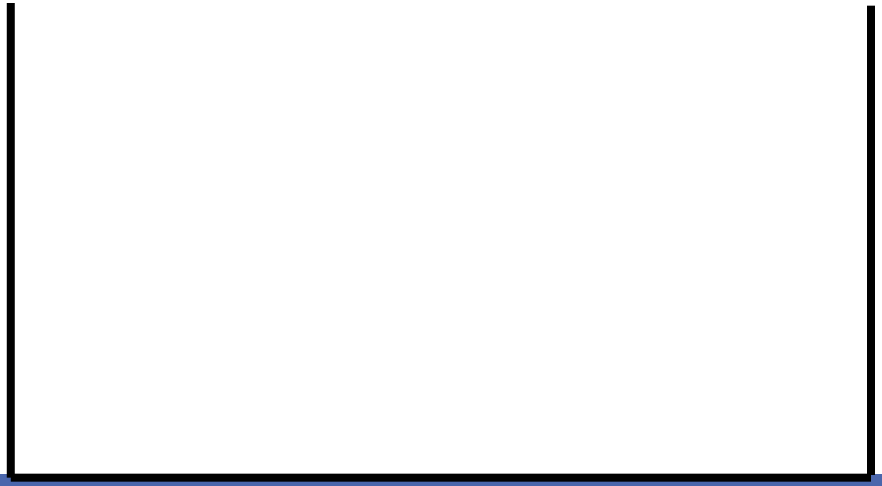
pre-order (red): F, B, A, D, C, E, G, I, H;

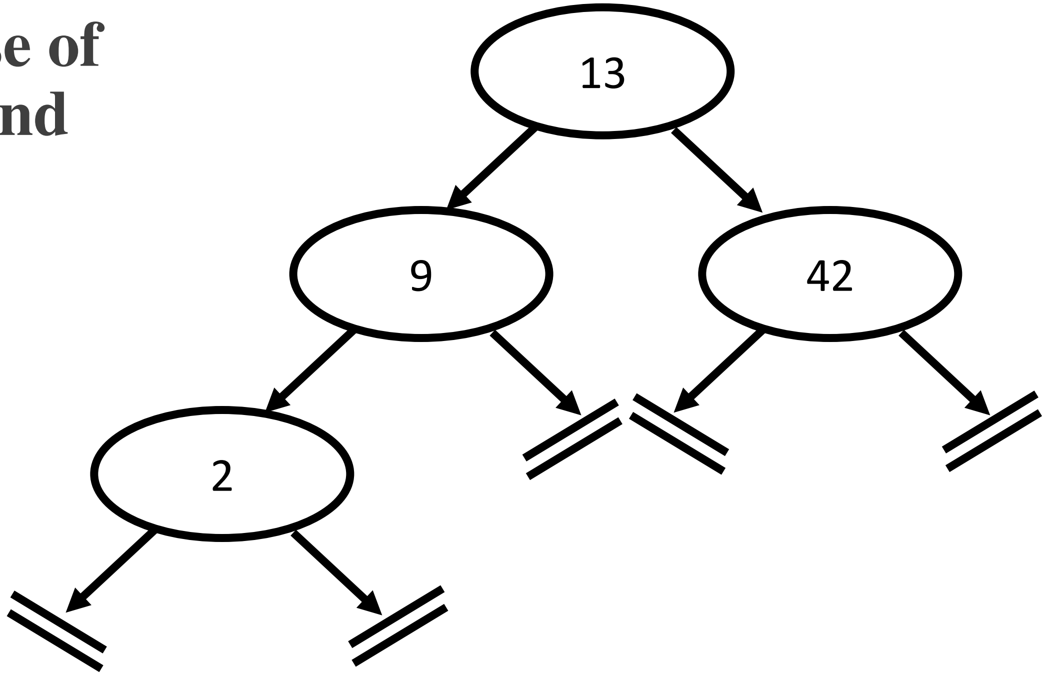
in-order (yellow): A, B, C, D, E, F, G, H, I;

post-order (green): A, C, E, D, B, H, I, G, F.

Use of the Activation Stack

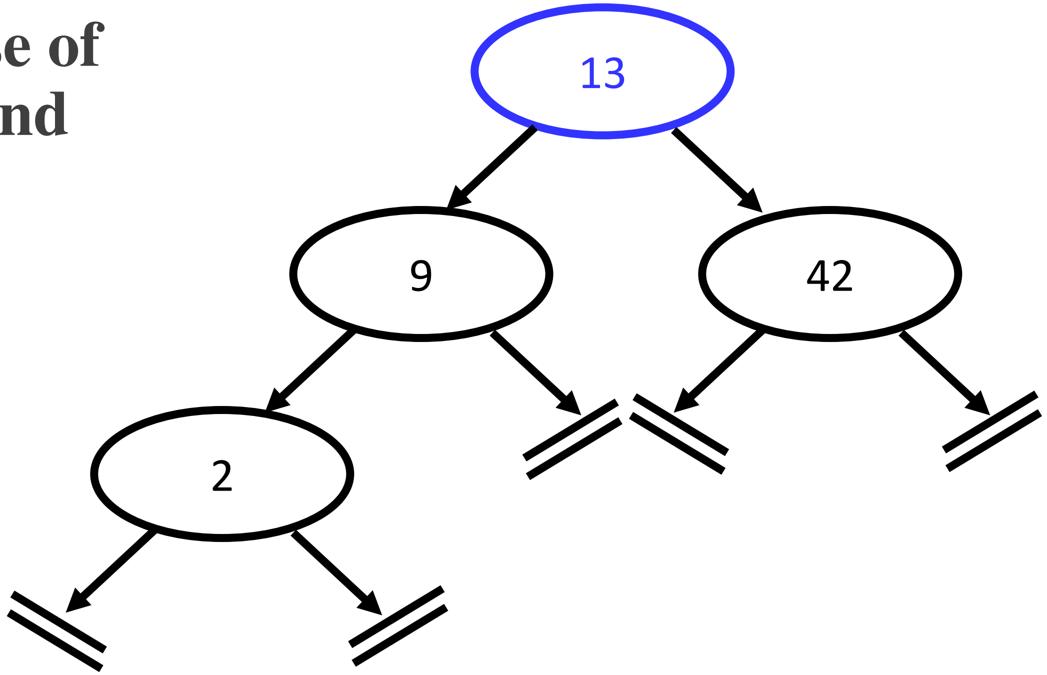
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

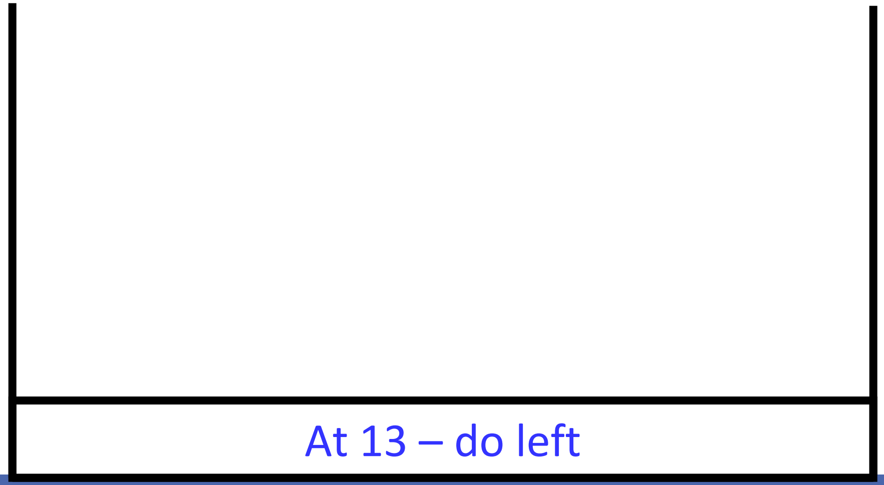




Use of the Activation Stack

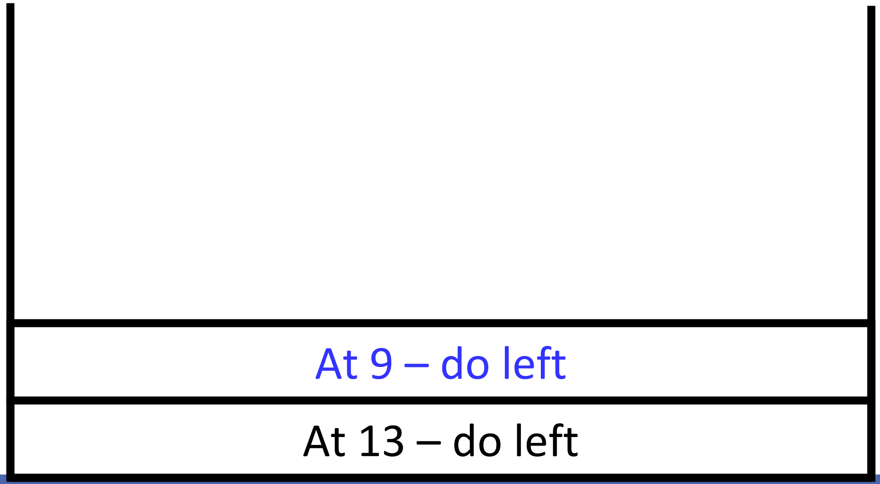
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.




At 13 – do left

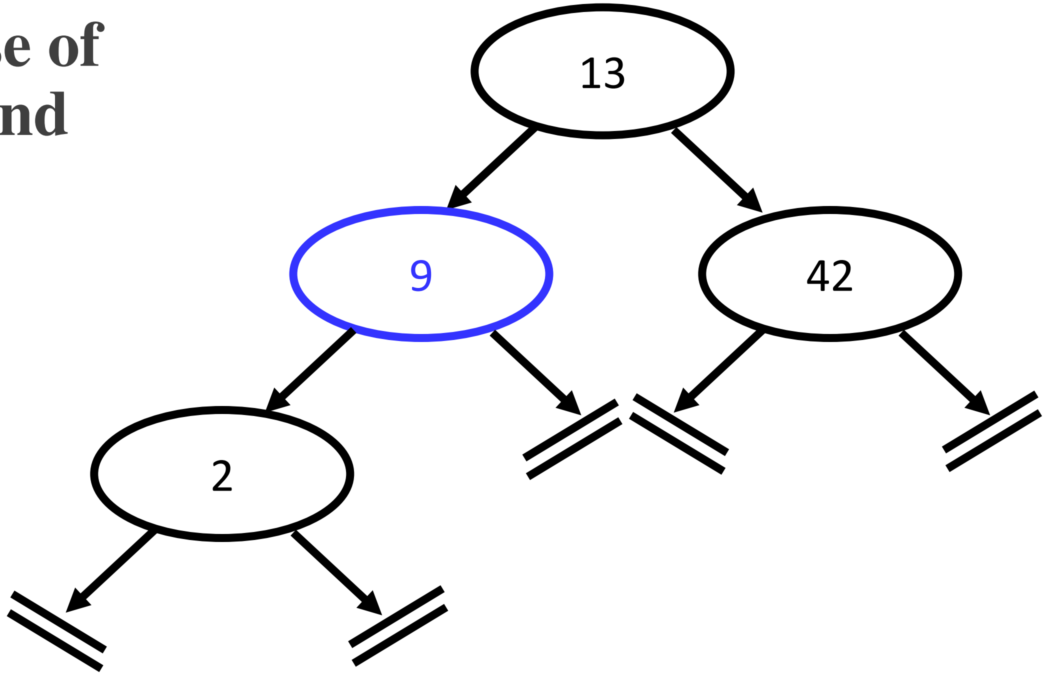
Use of the Activation Stack

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



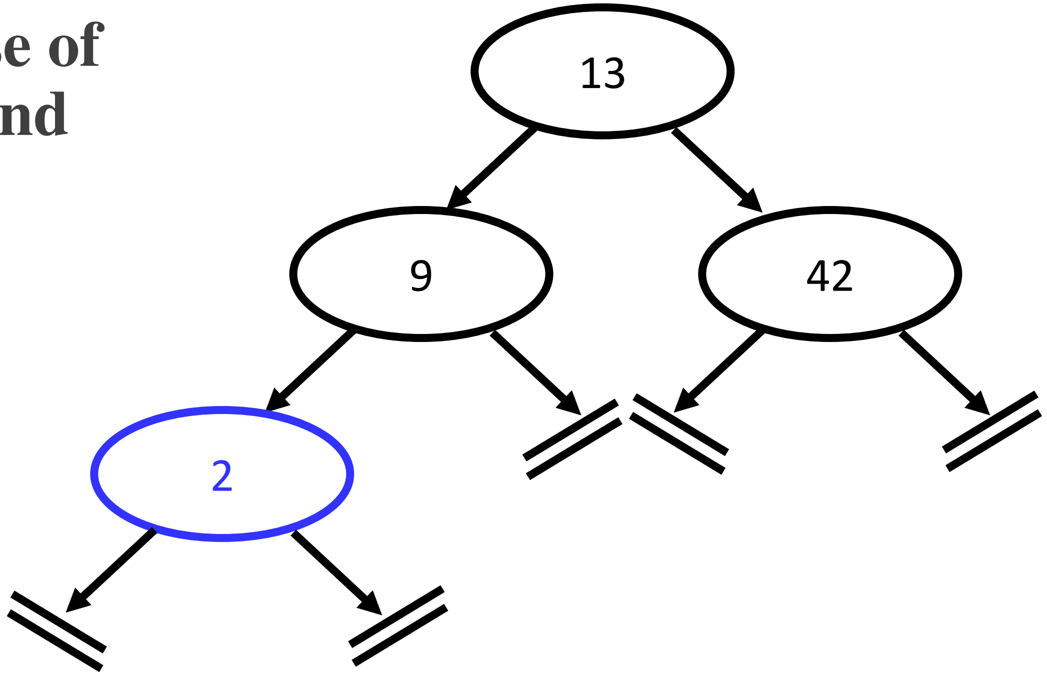
At 9 – do left

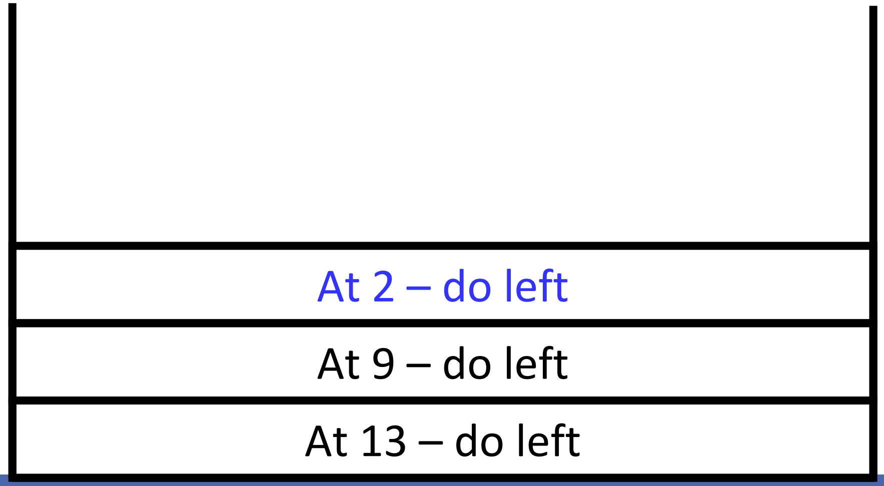
At 13 – do left



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.





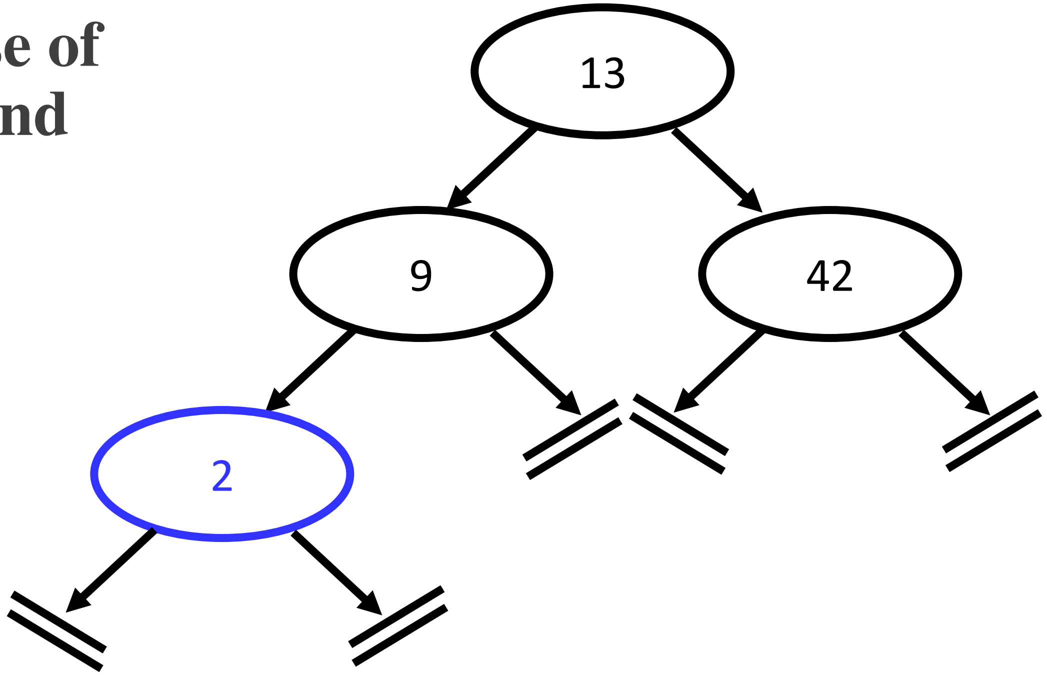
At 2 – do left

At 9 – do left

At 13 – do left

Use of the Activation Stack

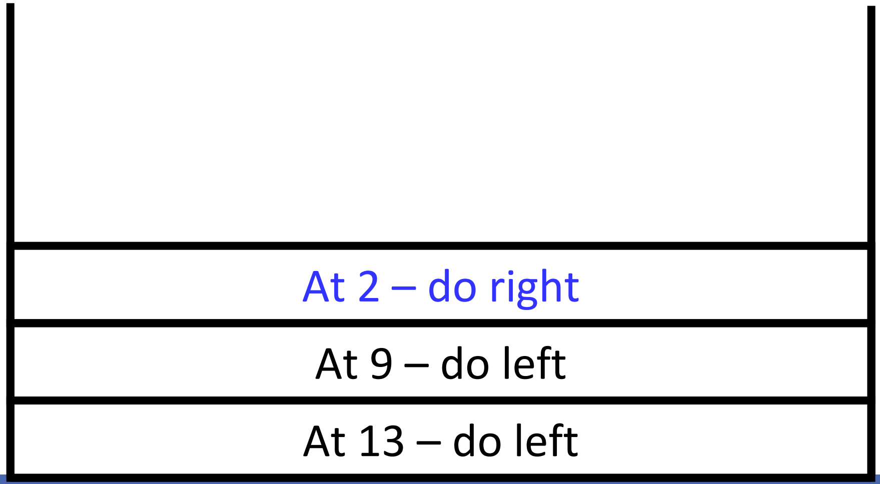
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.



At 2 – print
At 9 – do left
At 13 – do left

Use of the Activation Stack

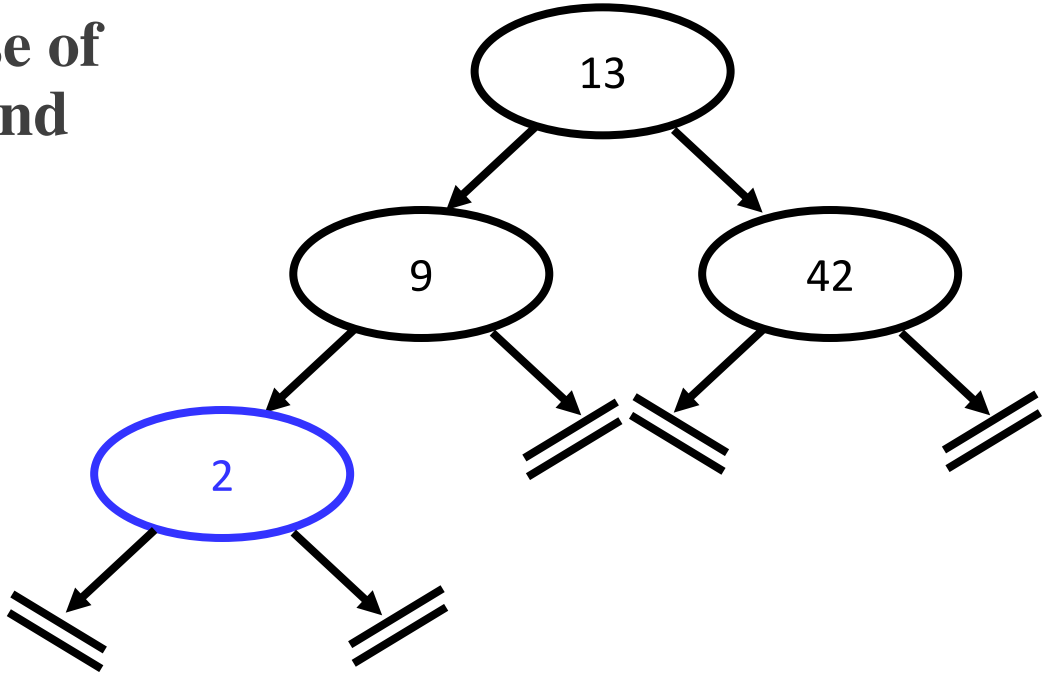
With a recursive module, we can make use of the activation stack to visit the sub-trees and “remember” where we left off.



At 2 – do right

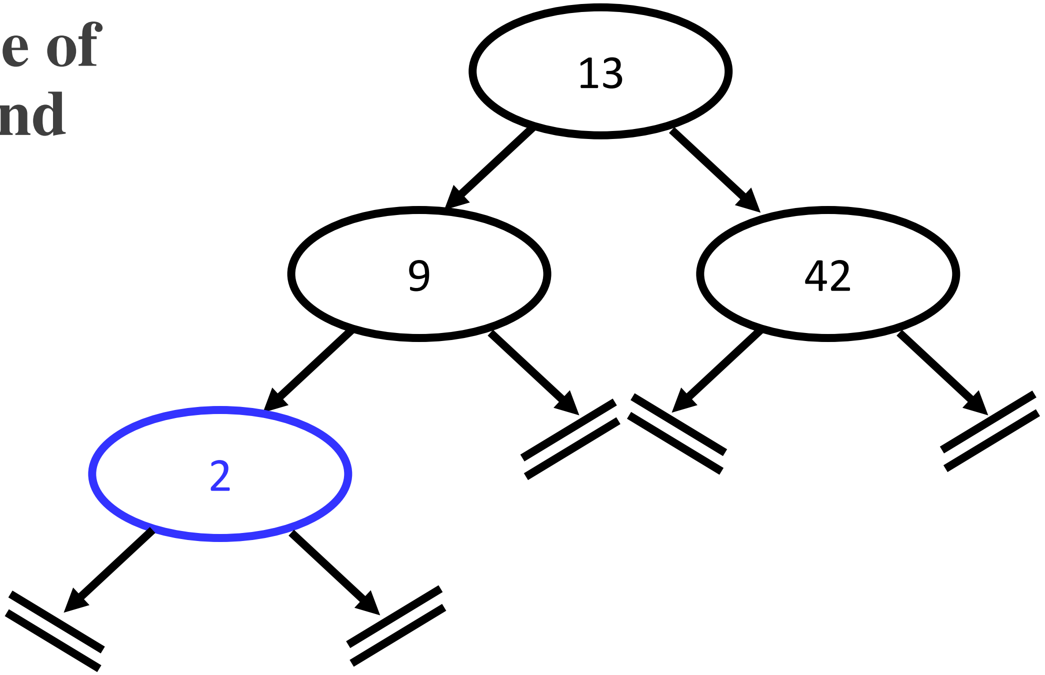
At 9 – do left

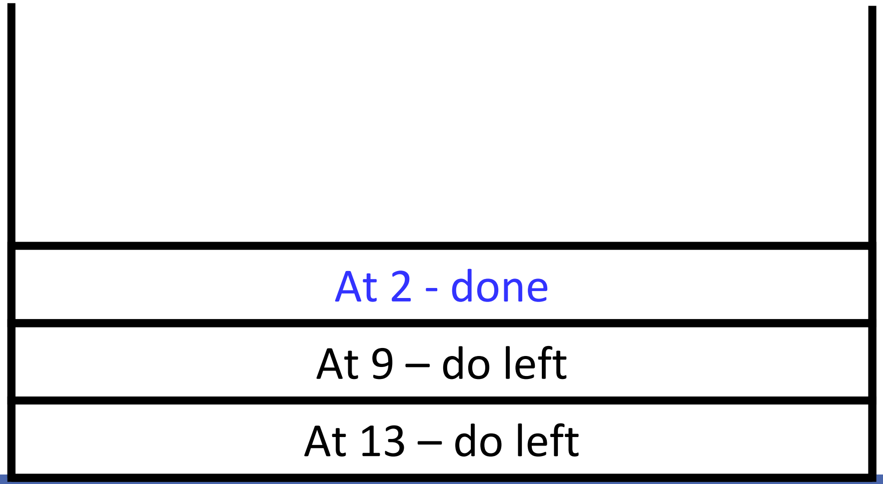
At 13 – do left



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.





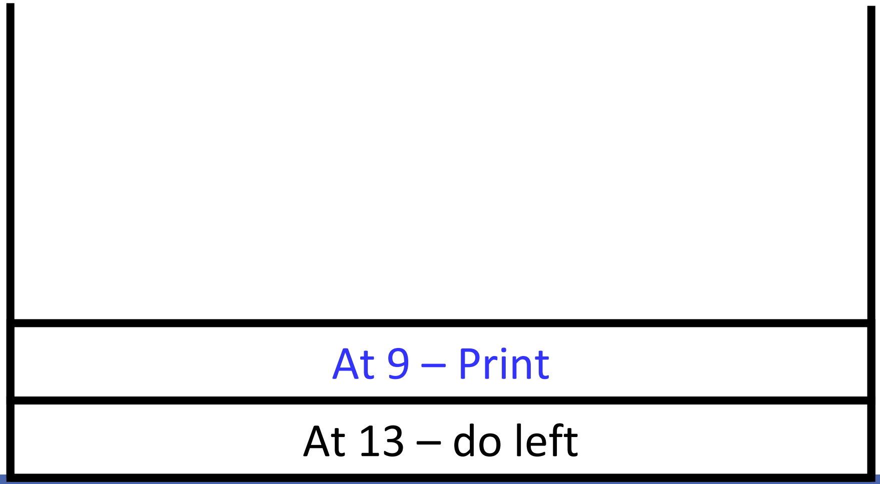
At 2 - done

At 9 – do left

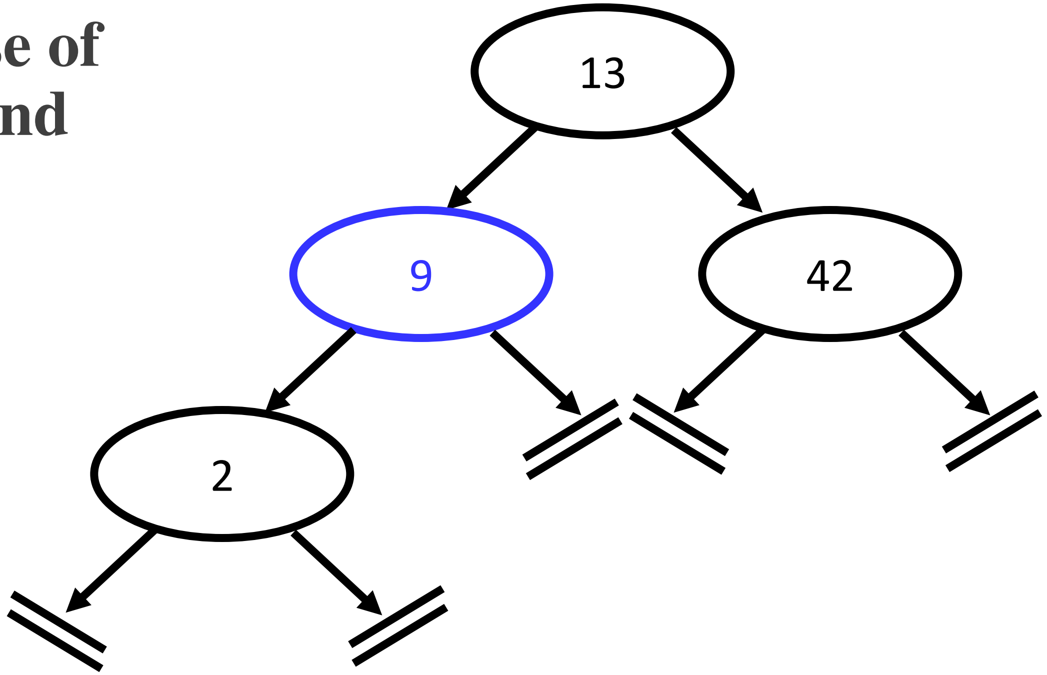
At 13 – do left

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.

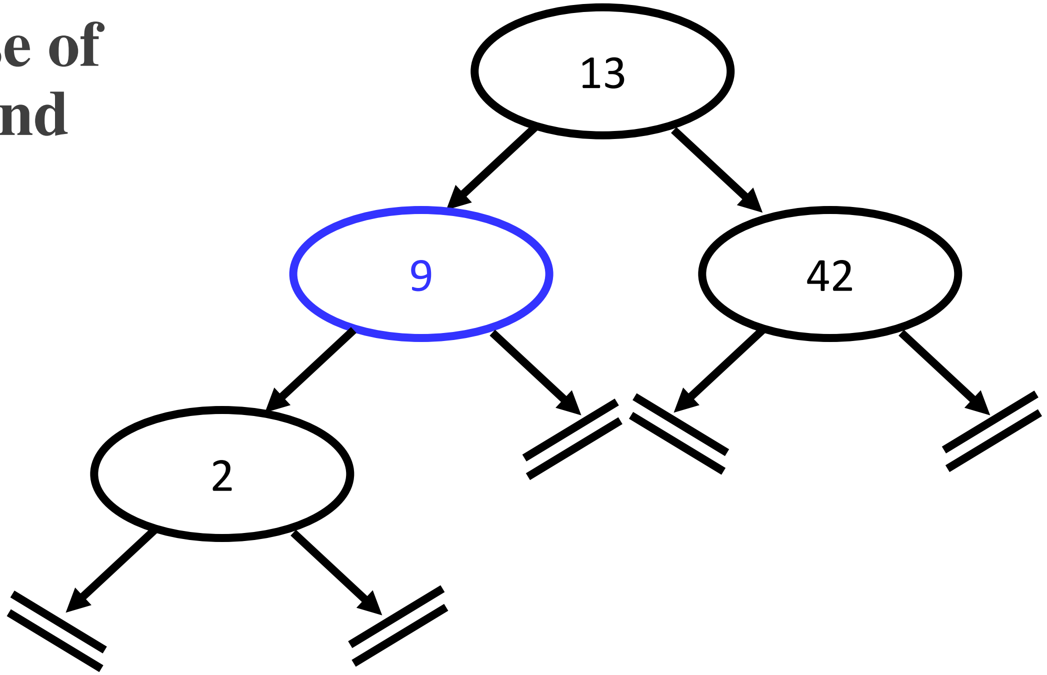

At 9 – Print

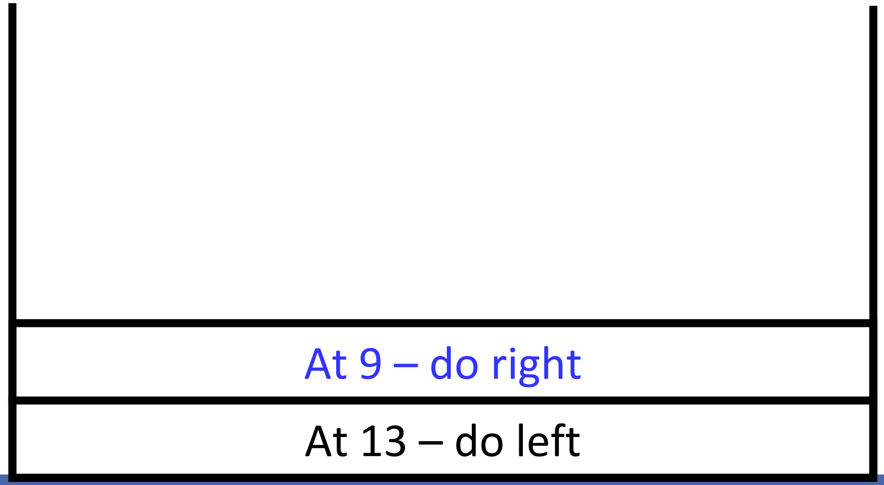
At 13 – do left



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.



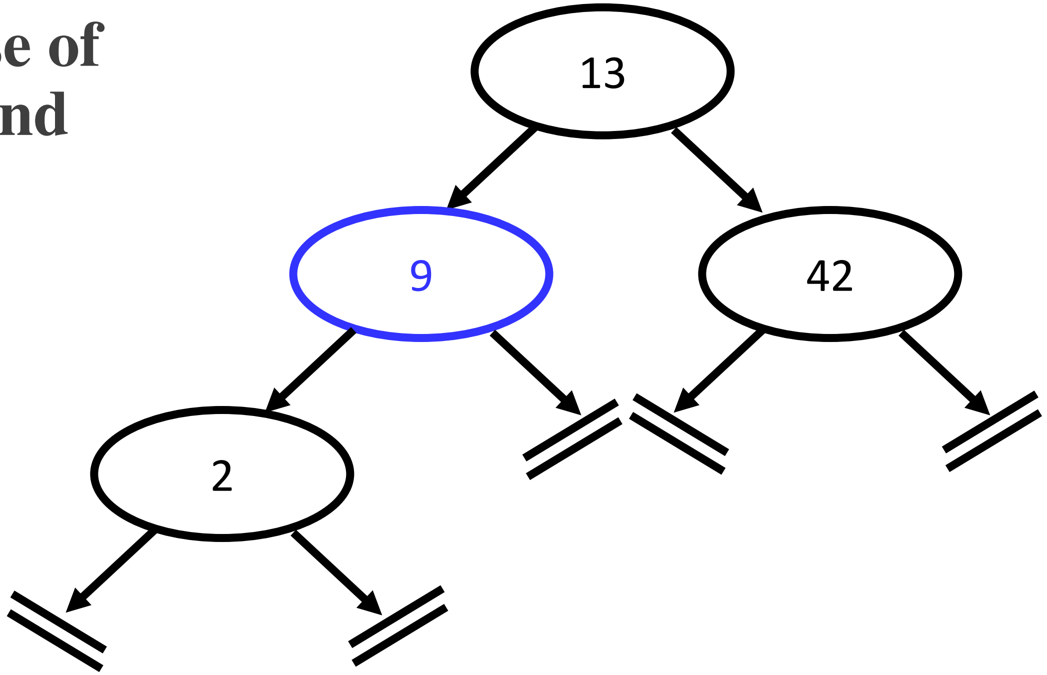


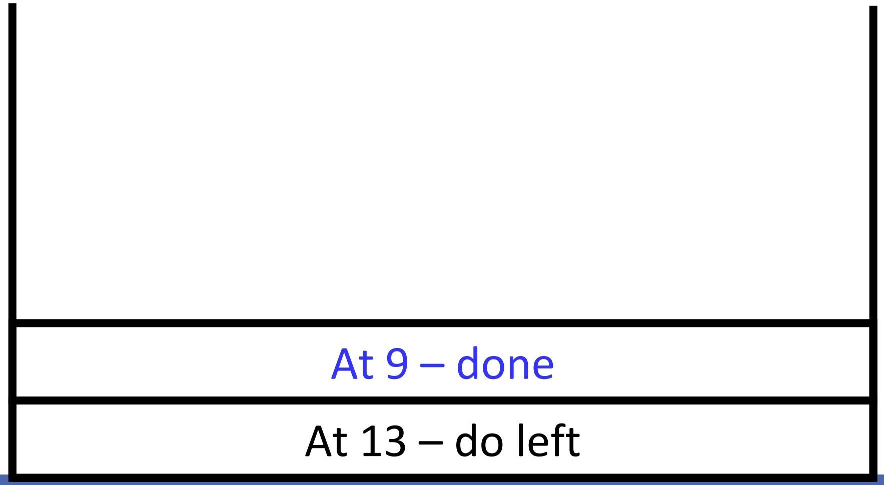
At 9 – do right

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.



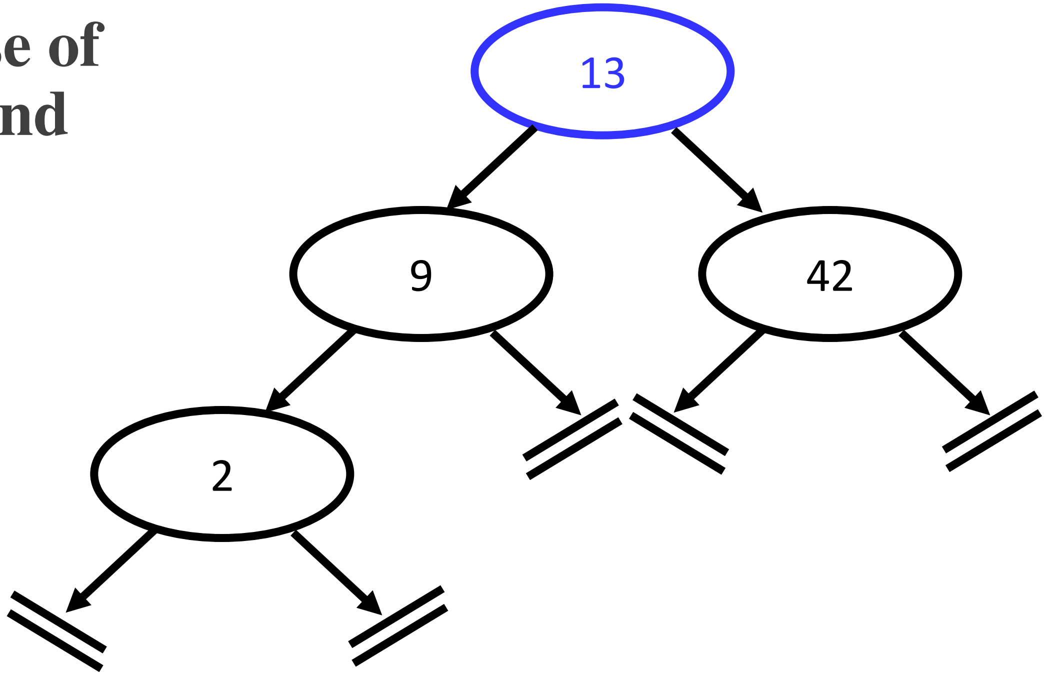


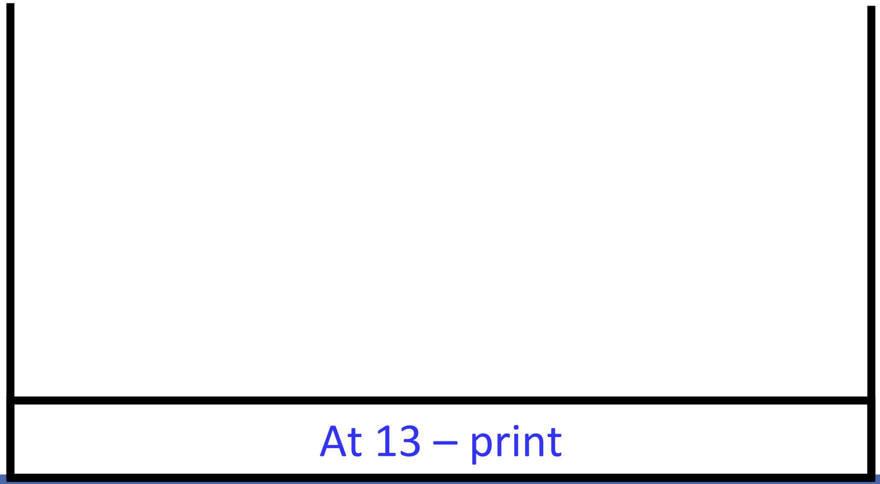
At 9 – done

At 13 – do left

Use of the Activation Stack

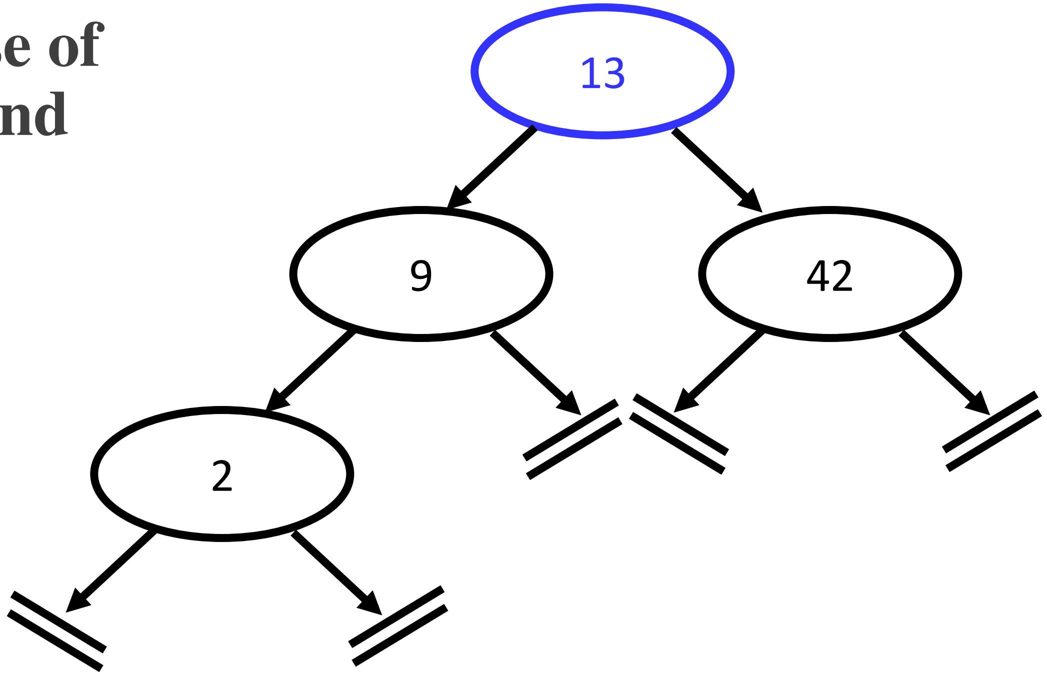
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

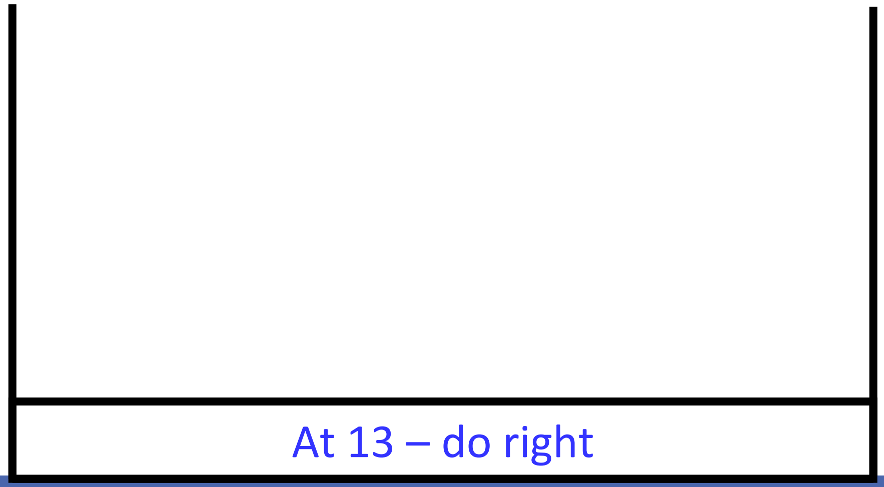



At 13 – print

Use of the Activation Stack

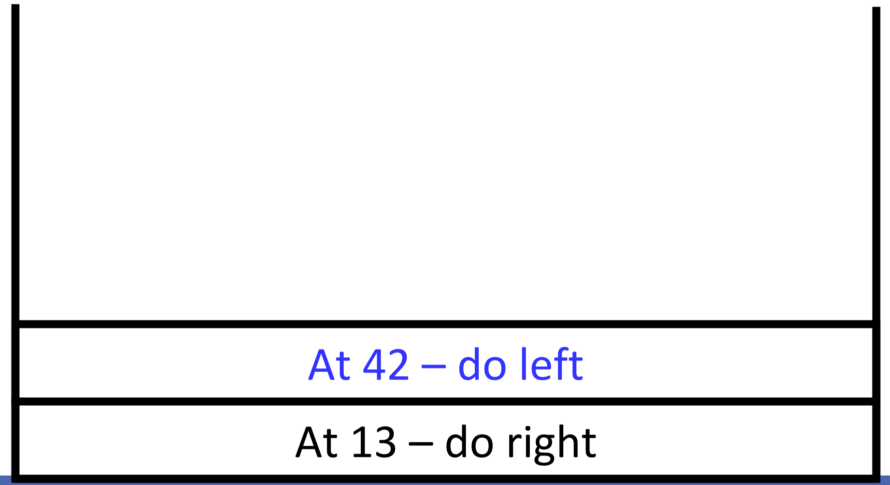
With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.





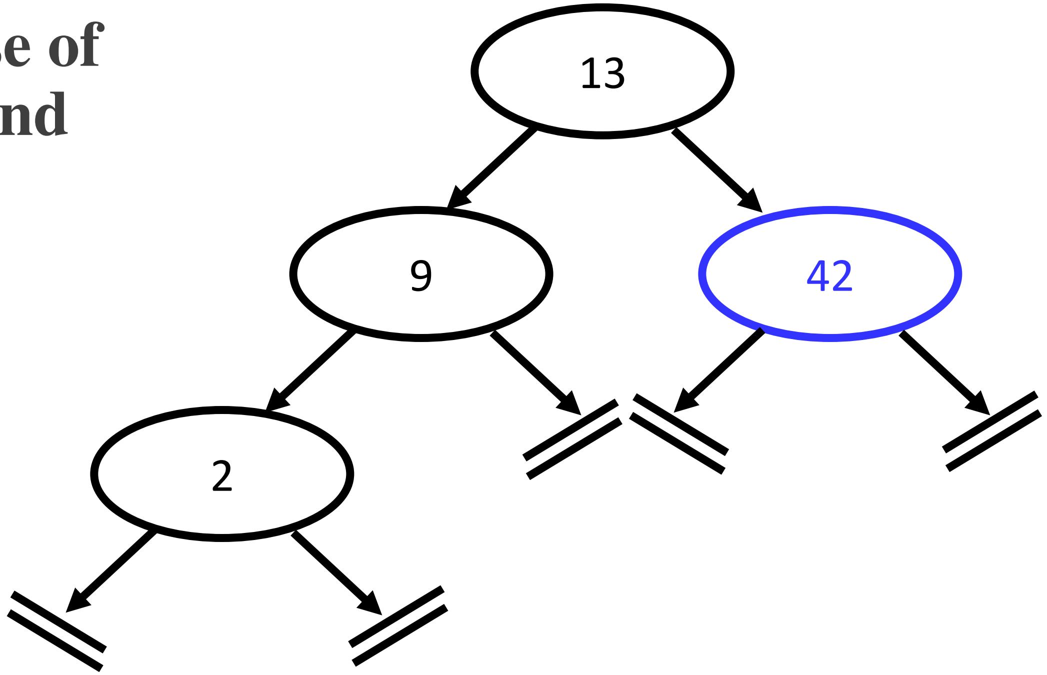
Use of the Activation Stack

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



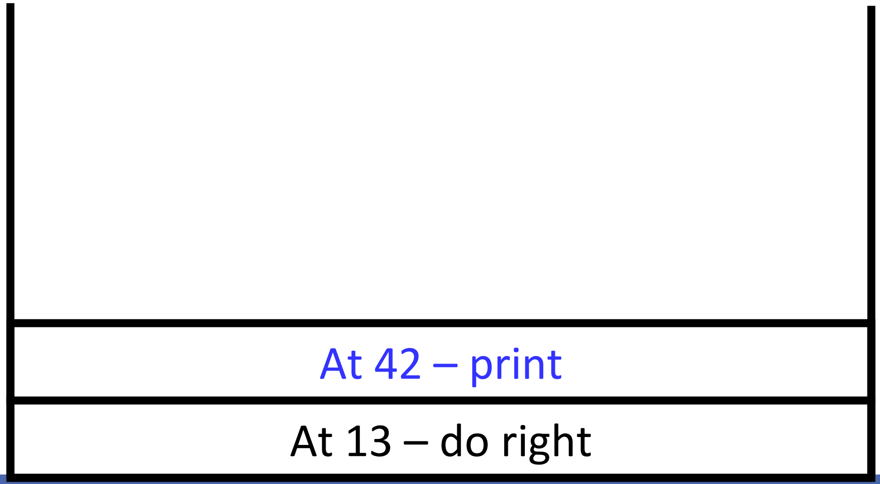
At 42 – do left

At 13 – do right



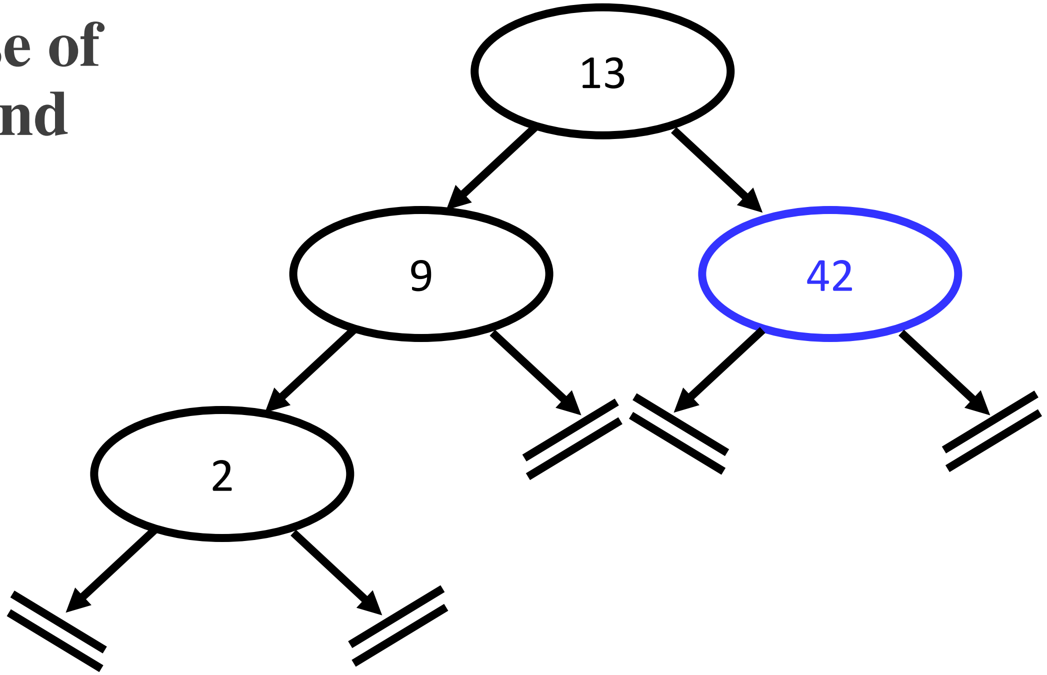
Use of the Activation Stack

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



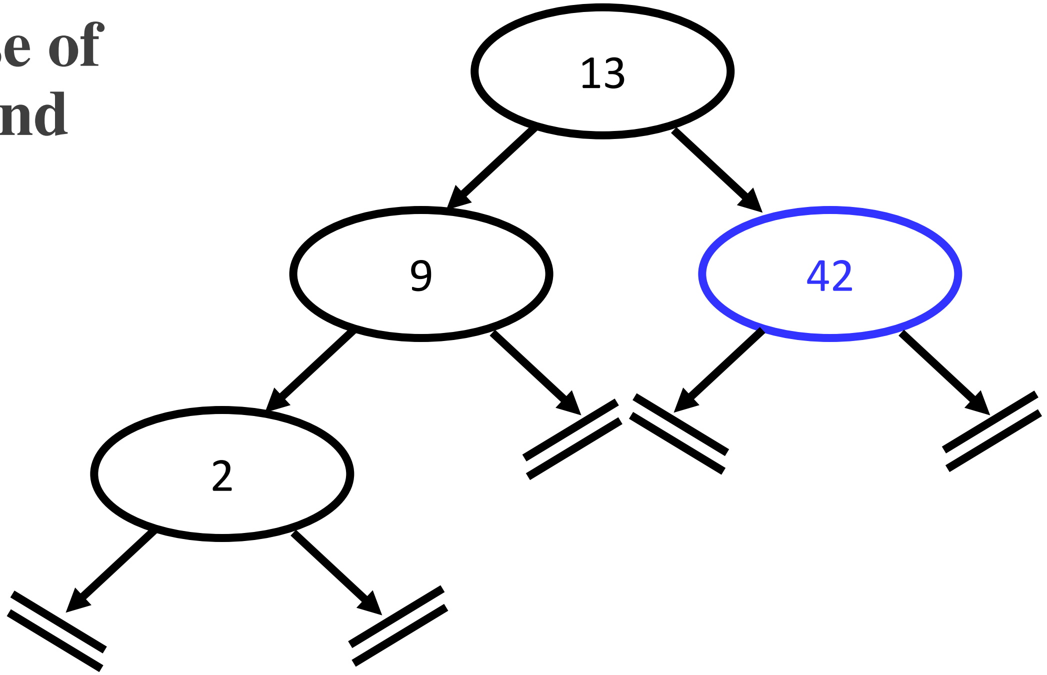
At 42 – print

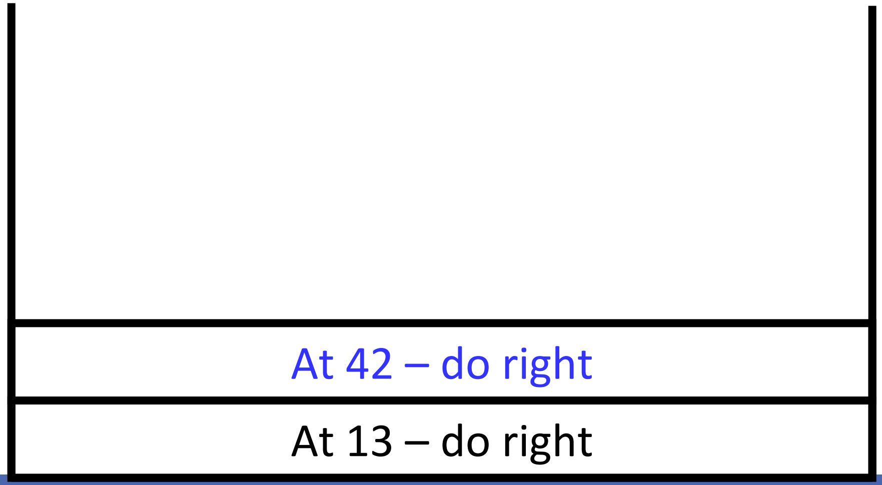
At 13 – do right



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.



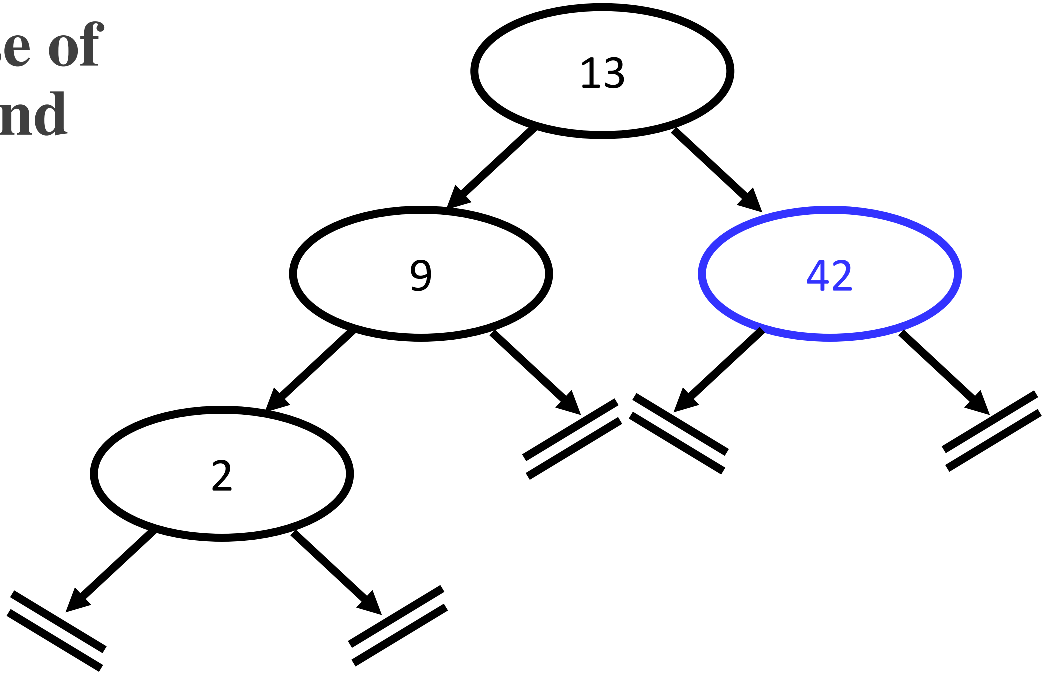


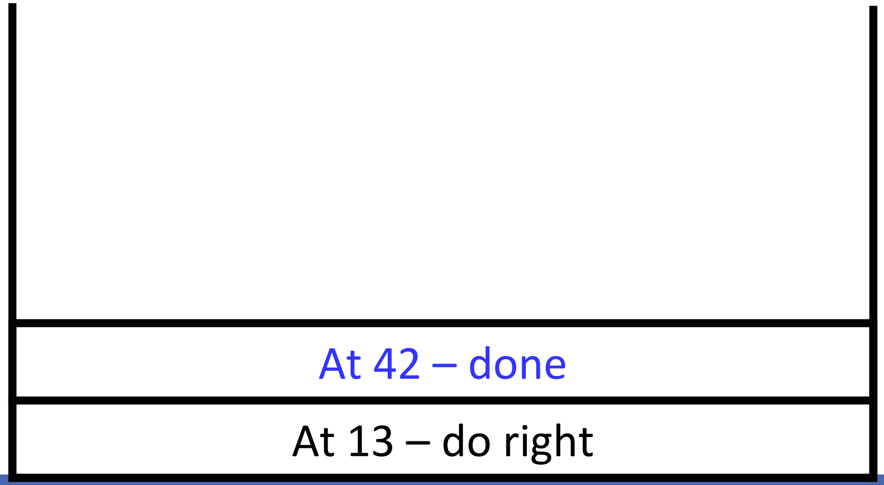
At 42 – do right

At 13 – do right

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.

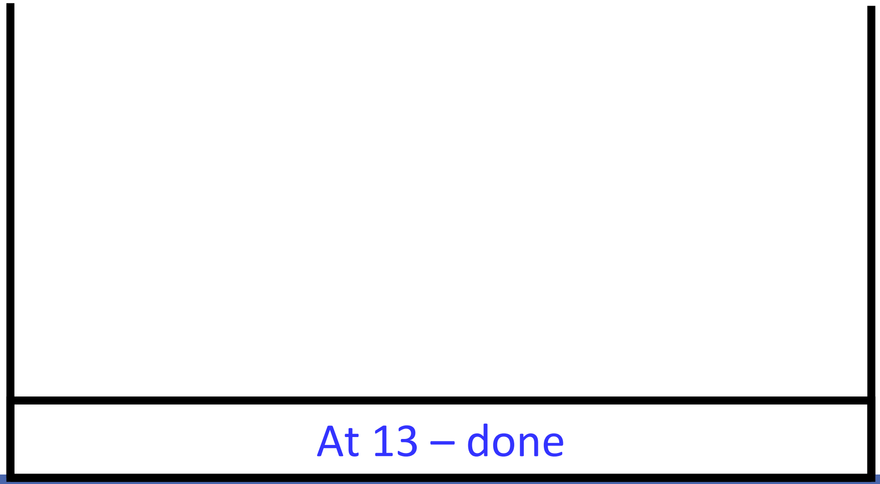



At 42 – done

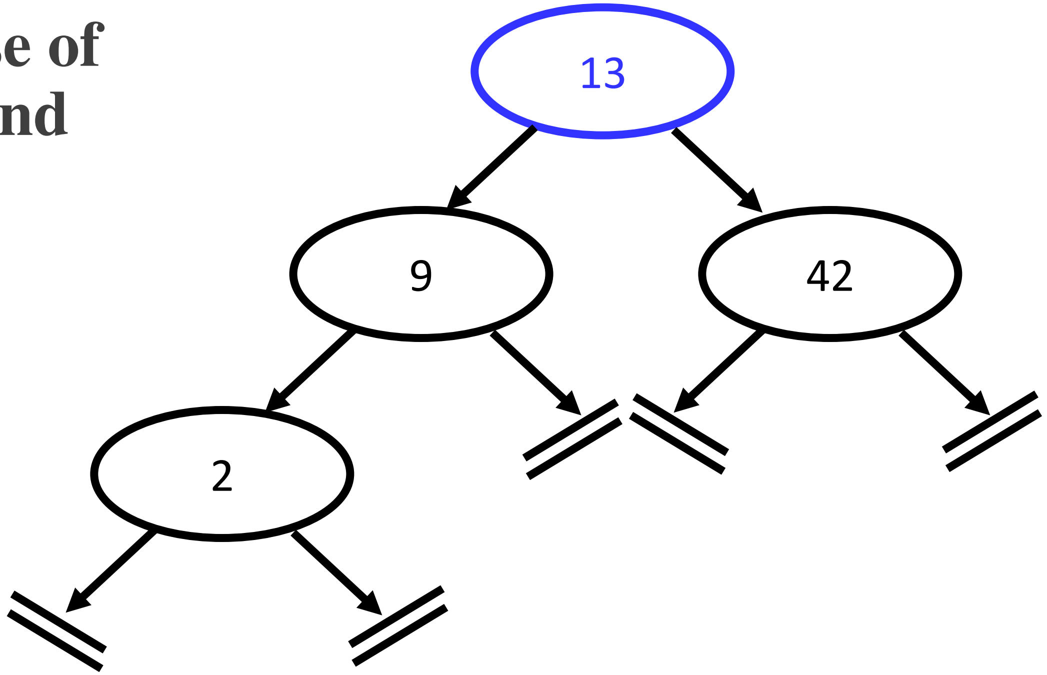
At 13 – do right

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.

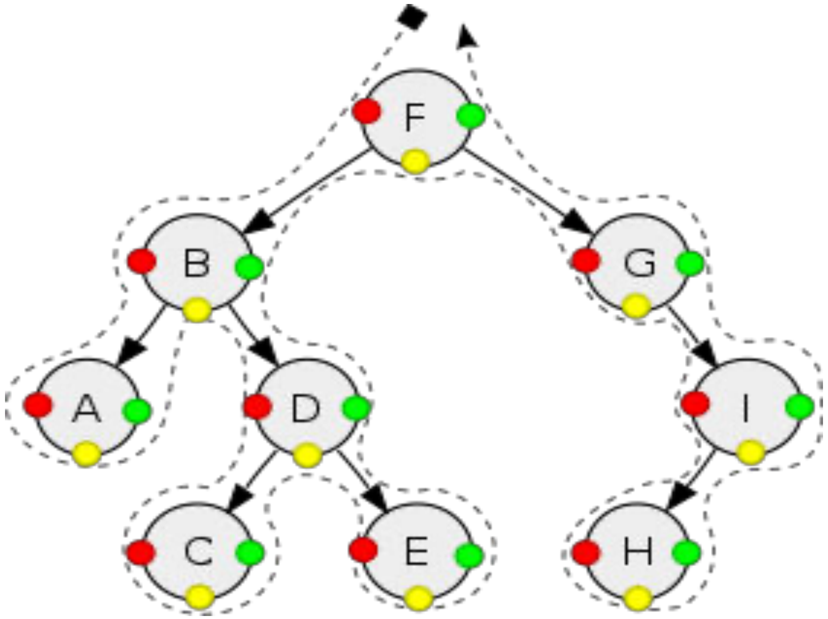


At 13 – done



Preorder Traversal (recursive version)

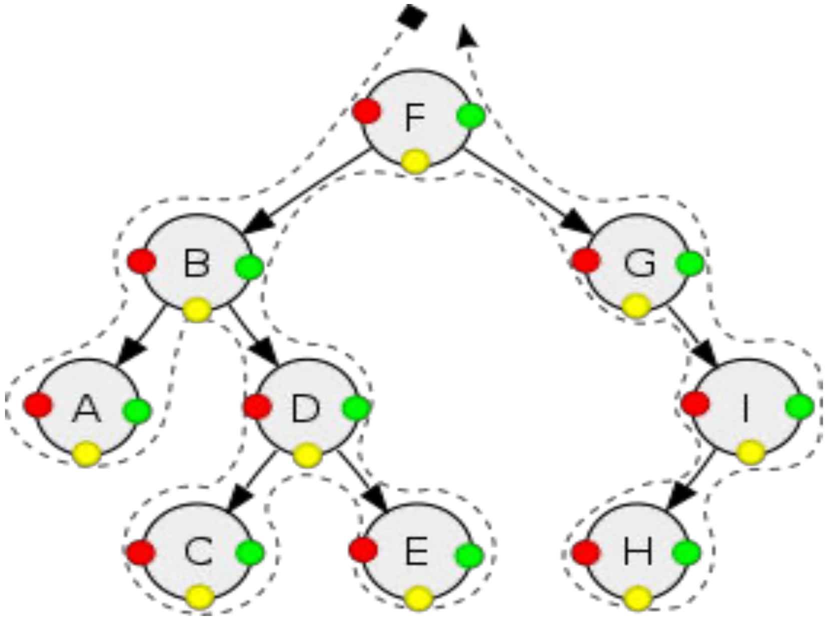
```
Algorithm preorder(treenode * temp)
/* preorder tree traversal */
{
    if (temp!=NULL) {
        print(temp->data);
        preorder(temp->left);
        predorder(temp->right);
    }
}
```



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);
   }
}
```



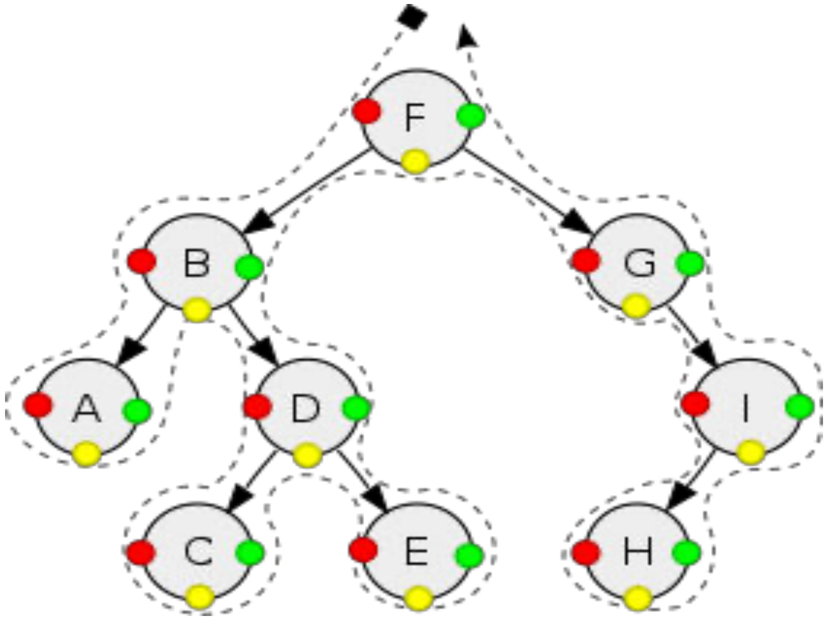
pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Stack for tree traversal

```
class stack
{
   int top;
   treenode *data[30];
   public:
    stack()
    {
       top=-1;
    }
    void push(treenode *temp);
     treenode *pop();
     int empty();
    friend class tree;
};
```

Nonrecursive Inorder Traversal

```
Algorithm inorder() {
     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
```



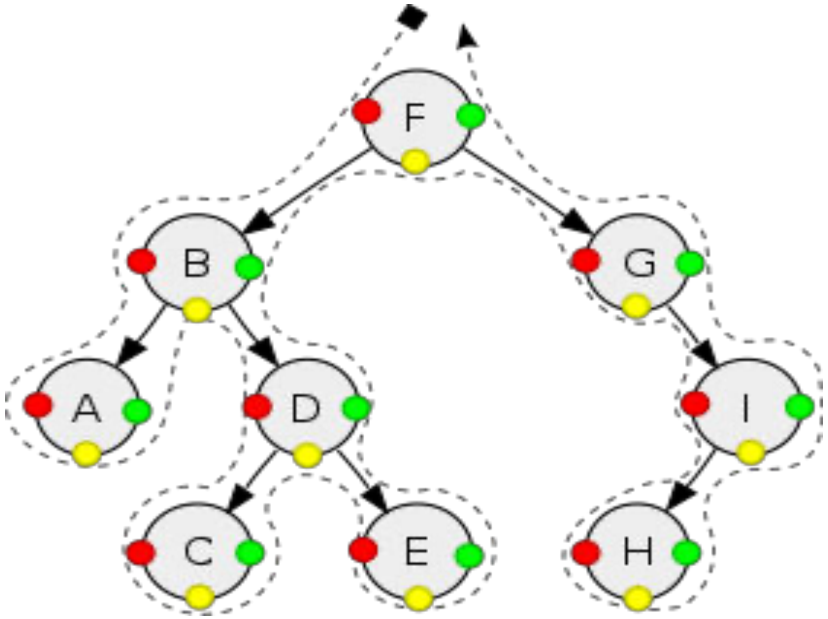
pre-order (red): F, B, A, D, C, E, G, I, H;

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() {
        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
```

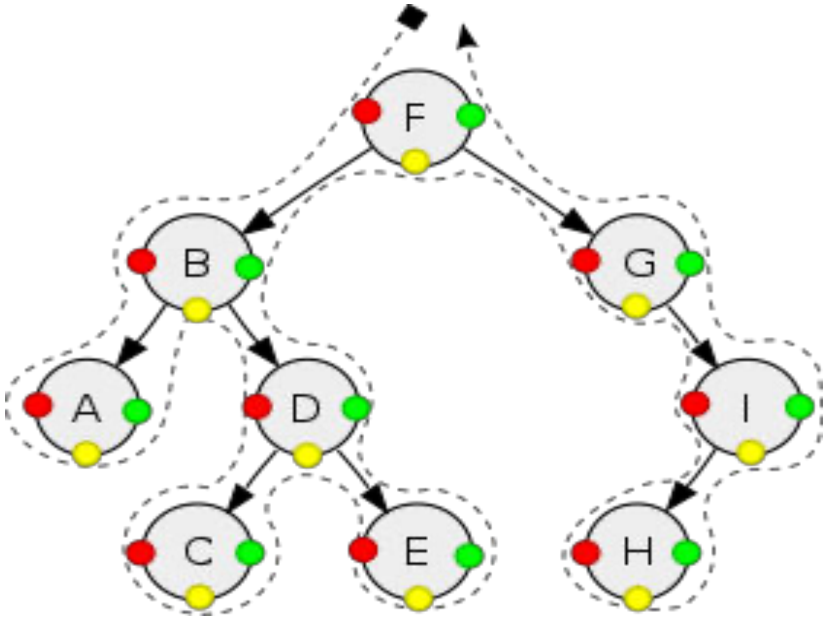


pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Nonrecursive Postorder Traversal

```
Algorithm postorder_nr()
{
  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;

    move temp to stack top right;
  }    // end while

}   // end algorithm
```

temp=st.data[st.top]->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.

```
void tree::dispbfs()
{
 int p=1,r=0;
 treenode *temp,*temp1;
 queue q,q1;
 temp=root;
 q.insqueue(temp);
 while(!q.empty())
 {
  for(int i=0;i<p;i++)
  {
   temp=q.delqueue();
   cout<<temp->data<<"\t";
   if(temp->left!=NULL)
   {
     q.insqueue(temp->left);
    r++;
   }
if(temp->right!=NULL)
  {
     q.insqueue(temp->right);
     r++;
    }
 }
cout<<endl;
p=r;
r=0;
}
}
```

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)

```
treenode  *copy(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;
}
```

```
void Tree::copy_runner(Tree *t2)
{
    t2->root = copy(root);
  }


main()
{        tree bst,bst_new;
         bst.create();
        bst.copy_runner(&bst_new);
        bst_new.inorder_r();
}
```

Copy of binary tree using non recursive is done through preorder

```
Algorithm copy_nr(tree t2)
{  //t2 is original tree
  Allocate memory for root
temp1=root;
temp2=t2.root;
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);
   }
    s1.push(temp1);
    s2.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()
{
Initialize d to 0;
temp=root;
while(1)
{
 while(temp!=NULL)
 {
    push temp;
    move temp to temp->left;
  if(d<st.top)
   d=st.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; }
```

```
Algorithm depth_r()
 {

  d=depth_r(root);
  print d;
 }
Algorithm 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()
{
 mirror_r(root);
 dispbfs();
 }

 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()
{
    temp=root;
    q.insqueue(temp);
    while(!q.empty())
  {
      temp=q.delqueue();
      swap left and right;
      if(temp->left!=NULL)
     q.insqueue(temp->left);
     if(temp->right!=NULL)
     q.insqueue(temp->right);
  }
  dispbfs();
 }
```

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.

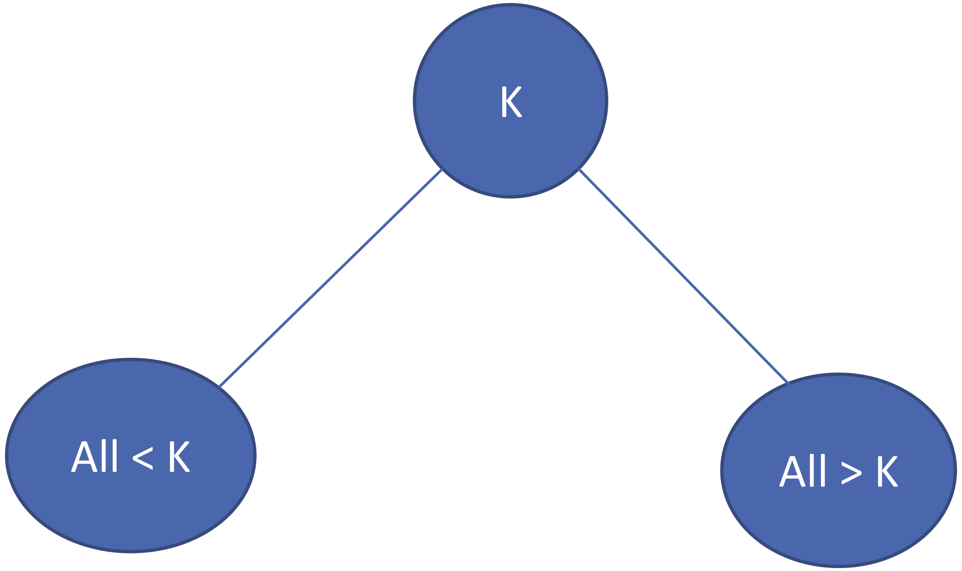
□The keys in a nonempty left subtree are smaller than the key in the root of subtree.

□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



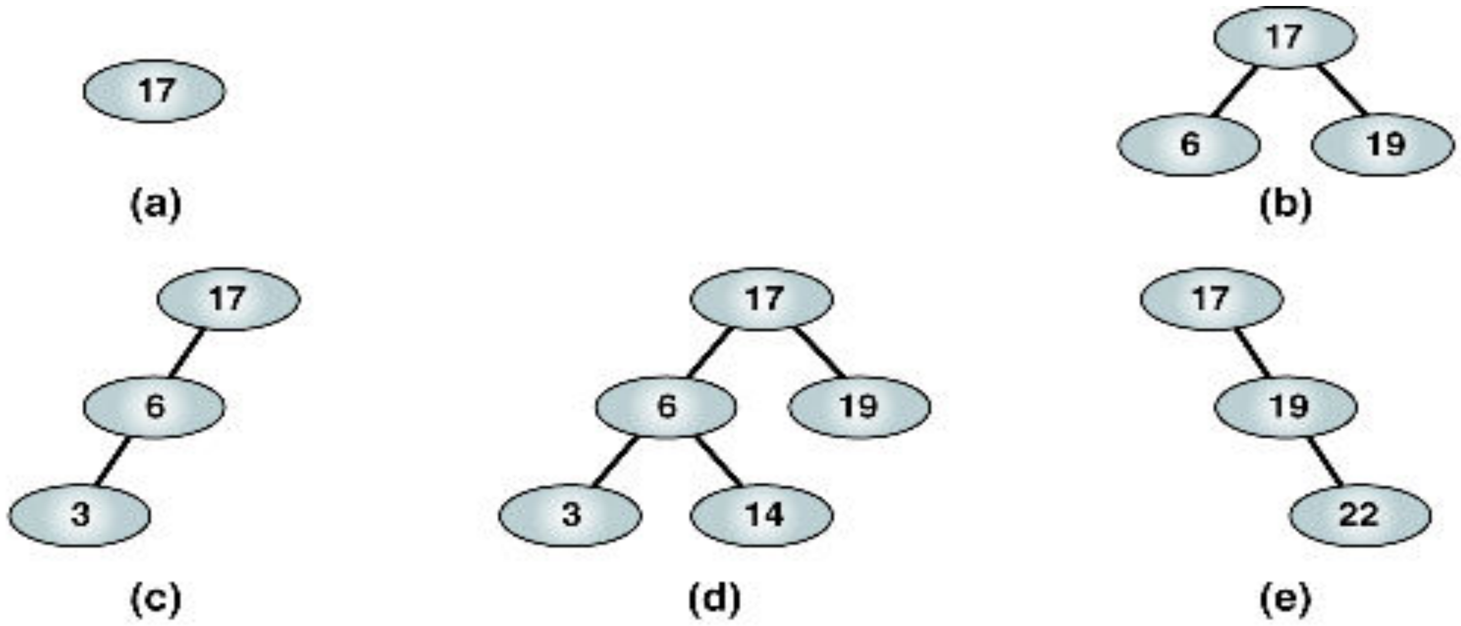


FIGURE 7-2 Valid Binary Search Trees

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

(d) – nearly complete and balanced tree;

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

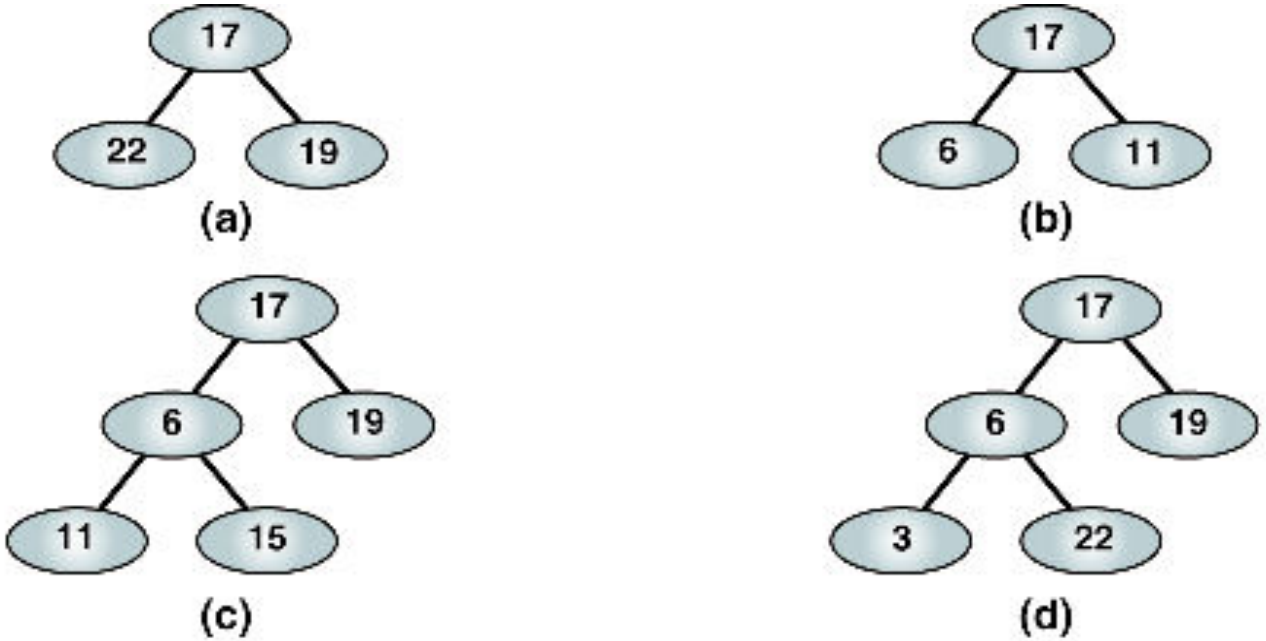


FIGURE 7-3 Invalid Binary Search Trees

```
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 ()
{
  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(temp,  string)
{
  Initialize f to 0;
  if(temp!=NULL)
  {
    if(string =temp->data)
           return 1;
    if(string< temp->data)
           f=search_r(temp->left, str);
     if(string >temp->data)
          f=search_r(temp->right, str);
   }
   return f;
}
```

```
Algorithm search_nr()
{
  Initialize  flag to 0;
 temp=root;
 Accept string to be searched;
 while(flag=0)
 {
   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 algo
```

Function DeleteItem

First, find the item; then, delete it

Important: binary search tree property must be preserved!!

We need to consider following different cases:

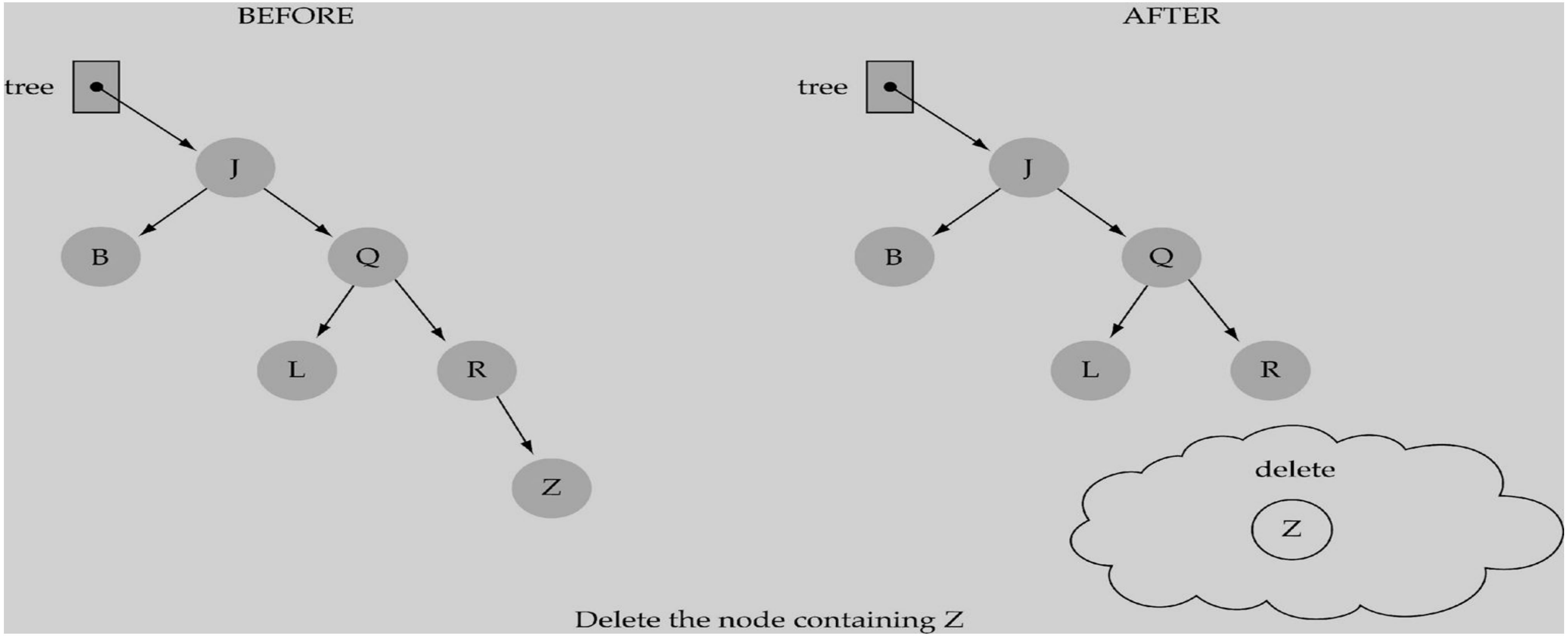
(1) Deleting a leaf

(2) Deleting a node with only one child

(3) Deleting a node with two children

(4) Deleting the root node

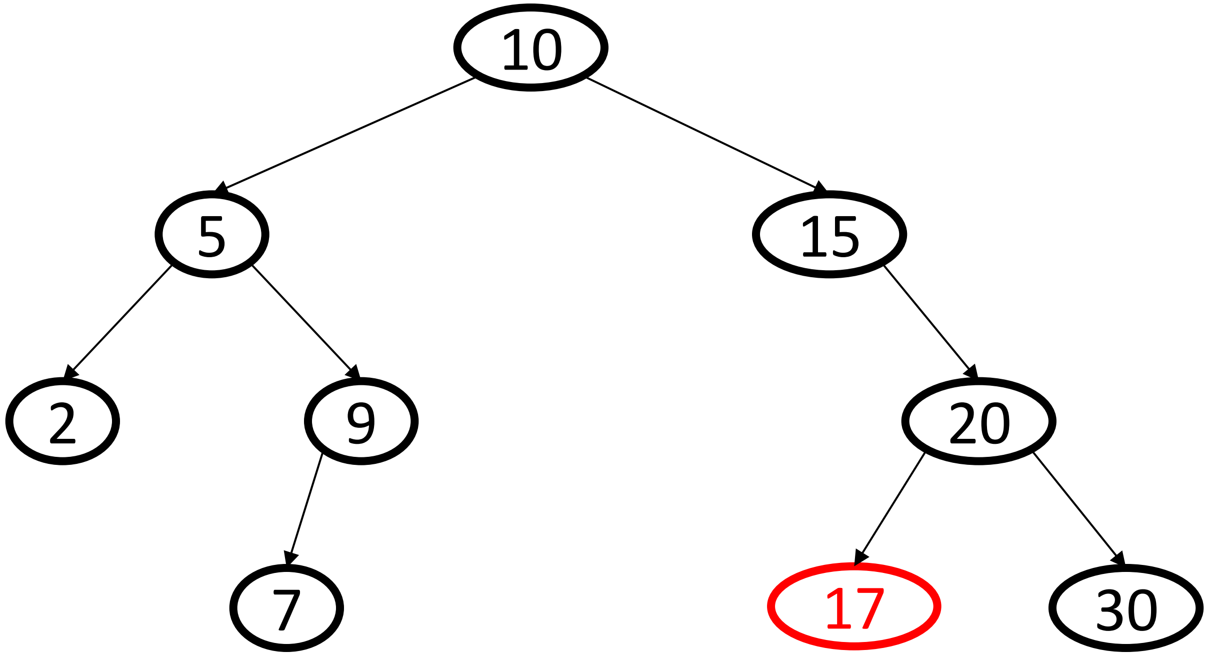
(1) Deleting a leaf



Deletion - Leaf Case

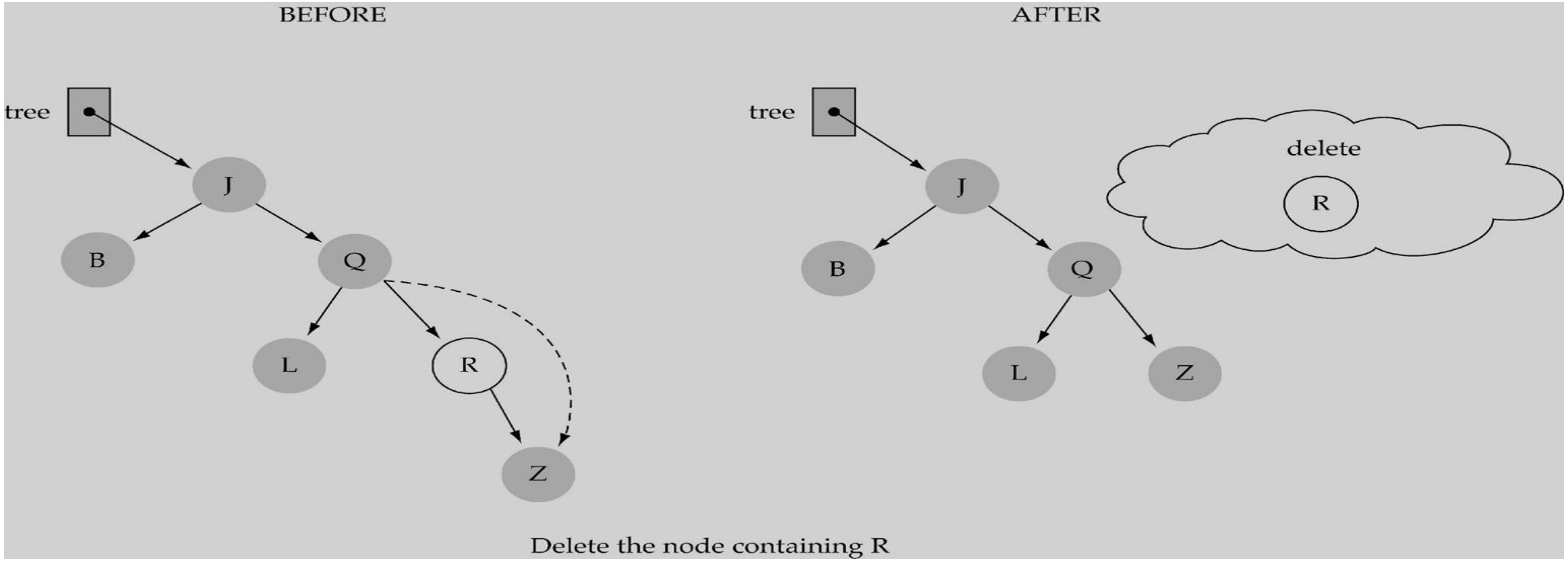
Algorithm sets corresponding link of the parent to NULL and disposes the node

Delete(17)



(2) Deleting a node with only one child

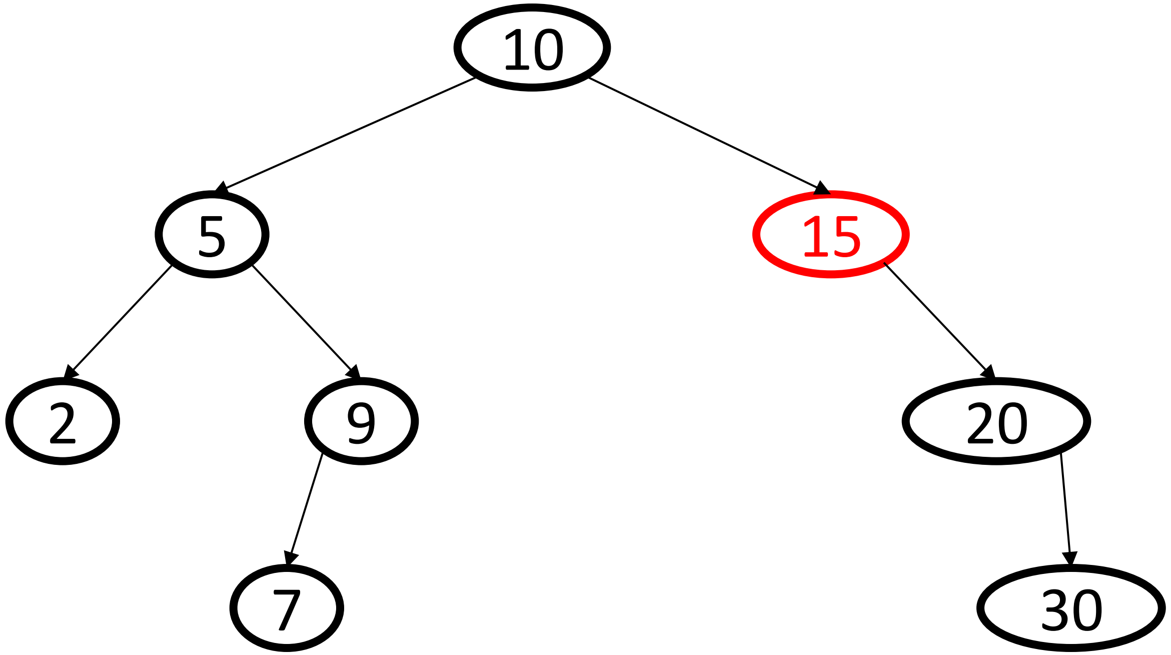
It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.



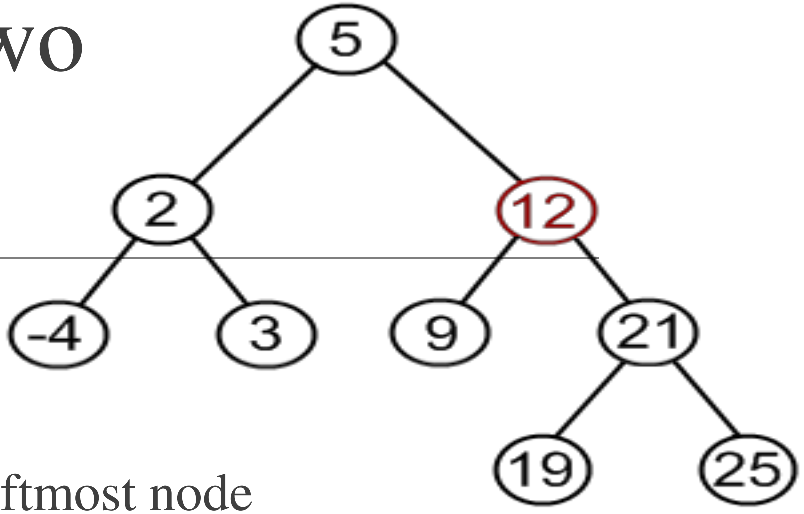
Delete the node containing R

Deletion - One Child Case

Delete(15)



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

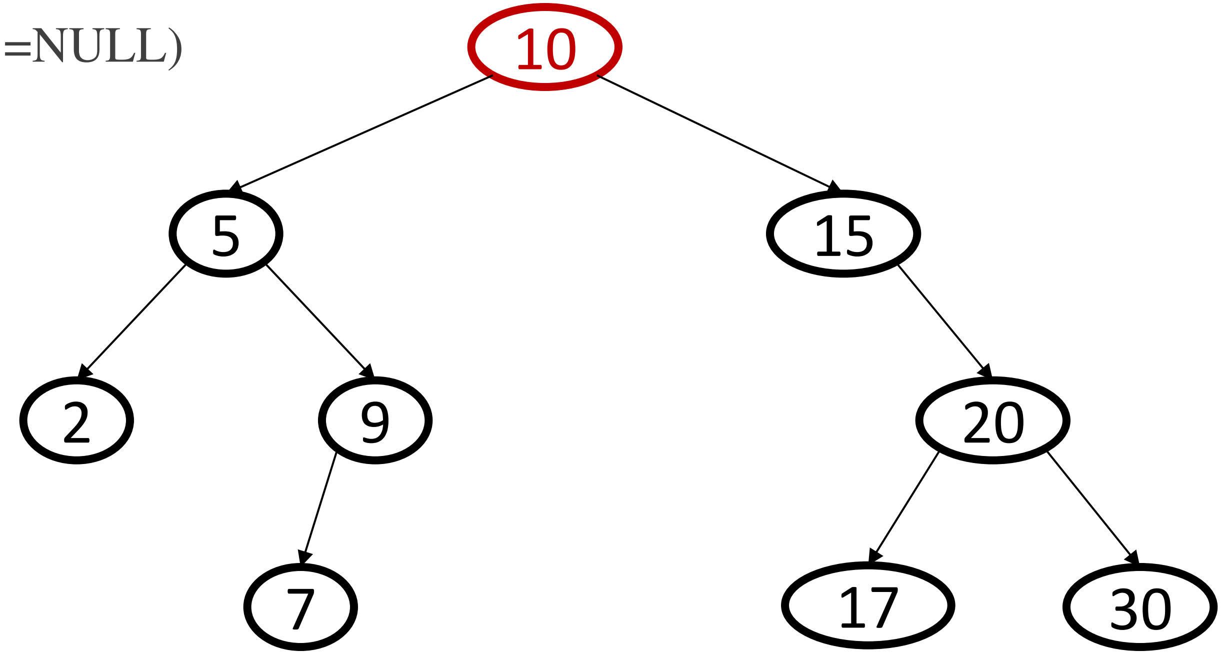


Find inorder successor

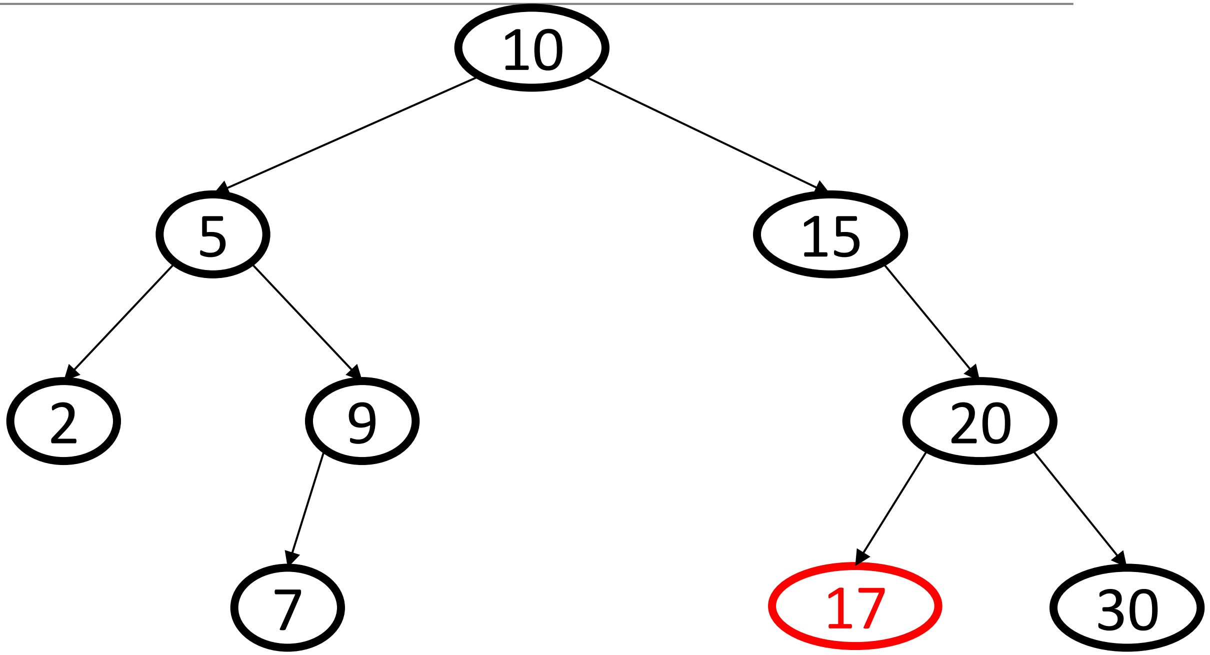
- Go to the right child and then move to the left till we get NULL for the leftmost node

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

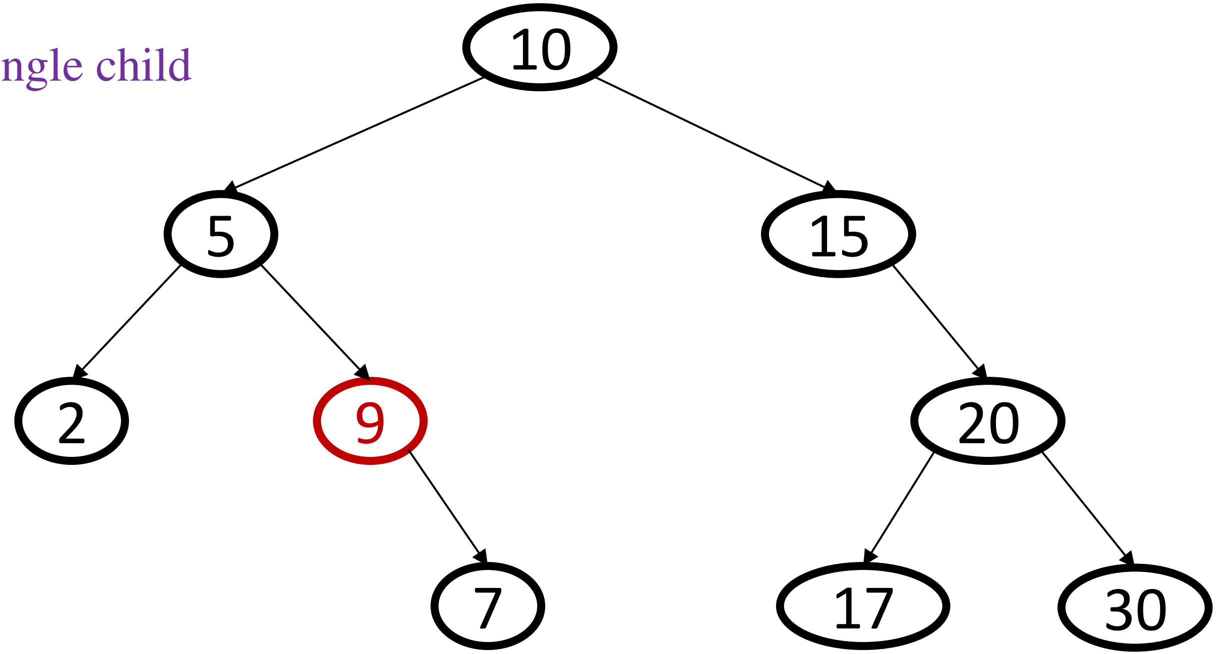
```
if(curr==root)                              //deletion of root
{
 if(curr->rightc==NULL)
            root=root->leftc;
 else if(curr->leftc==NULL)
            root=root->rightc;
 else if(curr->rightc!=NULL && curr->leftc!=NULL)
 {
            temp=curr->leftc;
            root=curr->rightc;
    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

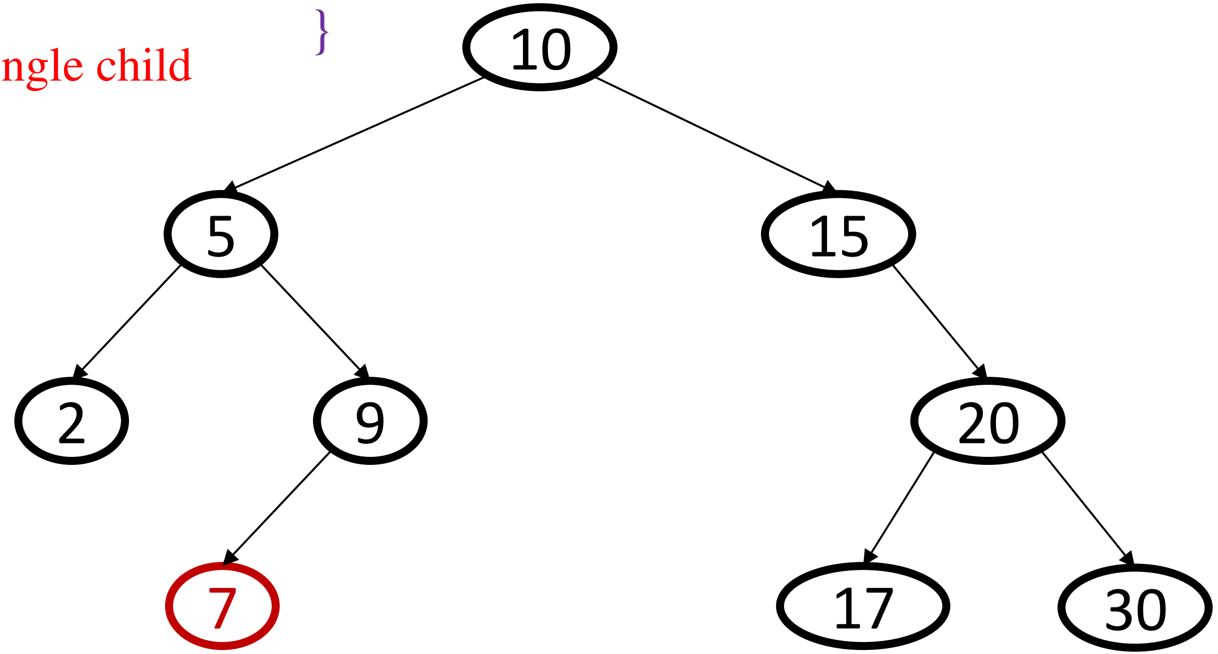


```
else if(curr!=root)            //deletion of node which is not root
{
 if(curr left and right is NULL )        //deletion of a leaf
 {
          if(parent->leftc==curr)
          parent->leftc=NULL;
          else
          parent->rightc=NULL;
 }
 else if(curr->leftc is NULL)      //deletion of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

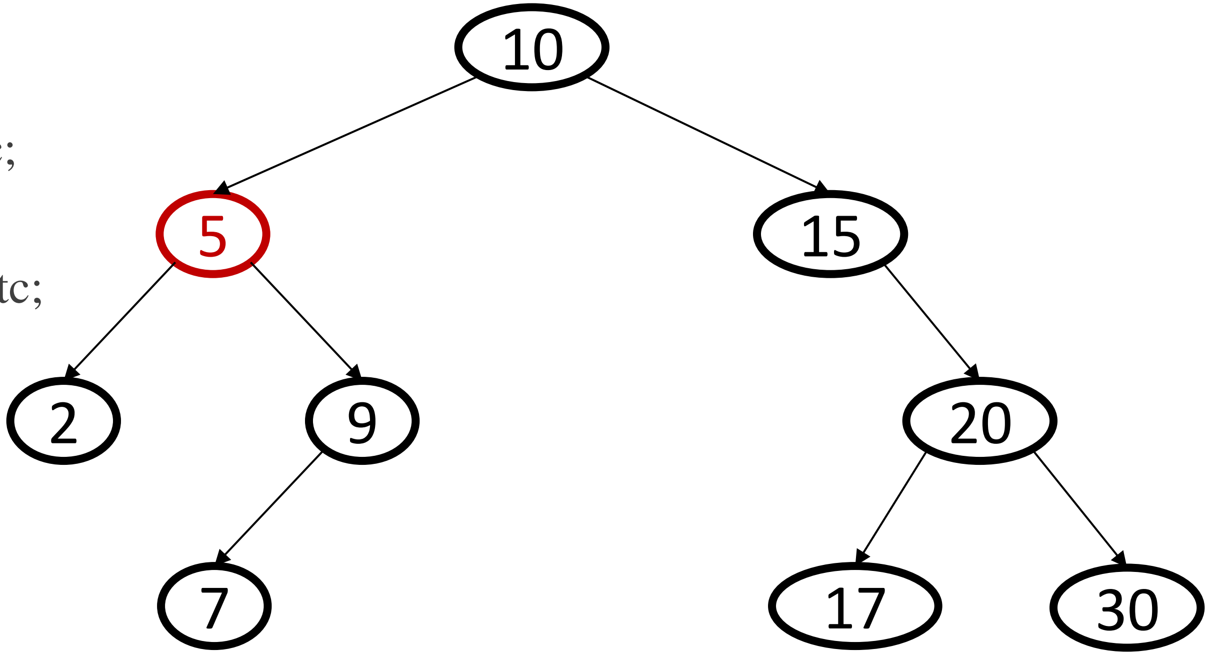


```
else if(curr!=root)          //deletion of node which is not root
{
 if(curr left and right is NULL )       //deletion of a leaf
 {
            if(parent->leftc==curr)
            parent->leftc=NULL;
            else
            parent->rightc=NULL;
 }
 else if(curr->leftc is NULL)      //deletion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



Assignment no 2

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

```
class dnode
{
    char word[20];
    char meaning[20];
    dnode *rightc;
    dnode *leftc;
    friend class dictionary;
    };

class dictionary
{
    dnode *root;
    public:
    void insert();
    void remove();
    void modify();
```

```
void displayBFS ();
dictionary();
};
dictionary::dictionary()
{
 root=NULL;
}
```

```
void dictionary::insert()
{
 dnode *curr;
 dnode *temp;
 do
    {
      temp=root;
      Allocate a memory for curr
      Accept word and meaning
     if(root==NULL)
    {
                    root=curr;
    }
    else
    {
       int flag=0;
       while (flag==0)
      {
        i= istrcmp(curr->word,temp->word);
```

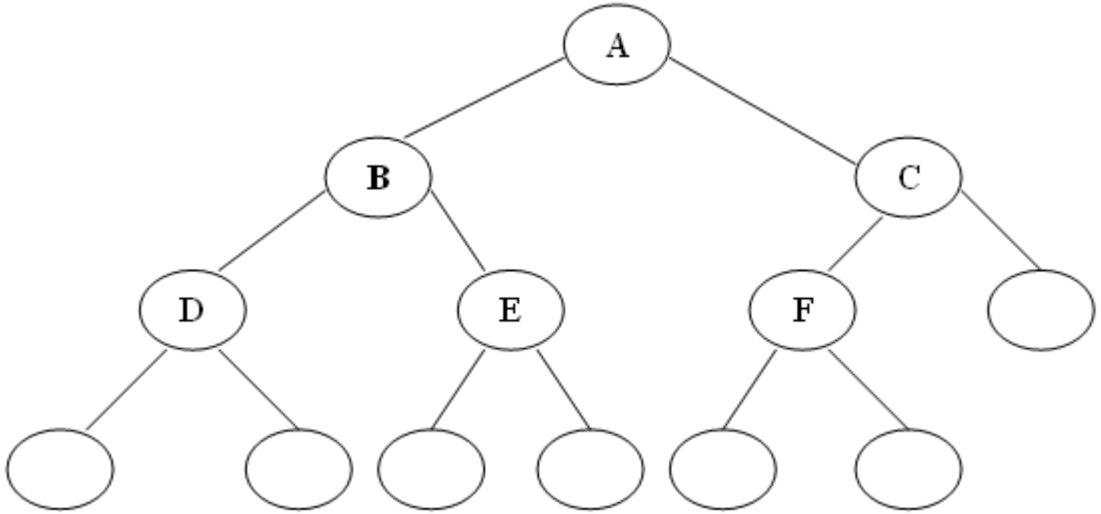
```
if(i<0)
{
     if(temp->left==NULL)
     {
       temp->left=curr;
        flag=1
      }
       temp=temp->left
}
  else if(i>0)
  {
      if(temp->right==NULL)
      {
                 temp->right=curr;
                  flag=1
        }
           temp=temp->right
   }
 else
       cout<<"\n Word already exists";
      }
      }
```

```
Algorithm modify()
{
   current=root
  Accept key for which meaning has to be  modified temp
 while(strcmp(temp,current->data)!=0)
{
       if(strcmp(temp,current->data)<0)
       {
              if(current->left !=NULL)
                     current=current->left;
              else
              {
                print Data does not exist”;
               break;
              }
        }
       else if(strcmp(temp ,current->data)>0)
          {
                if(current->right!=NULL)
                      current=current->right;
               else
               {
                      Print  Data does not exist”;
                      break;
                }
          }
       if(strcmp(temp,current->data)==0)
       {
         Accept new meaning in curr->meaning;
        }
}
```

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.

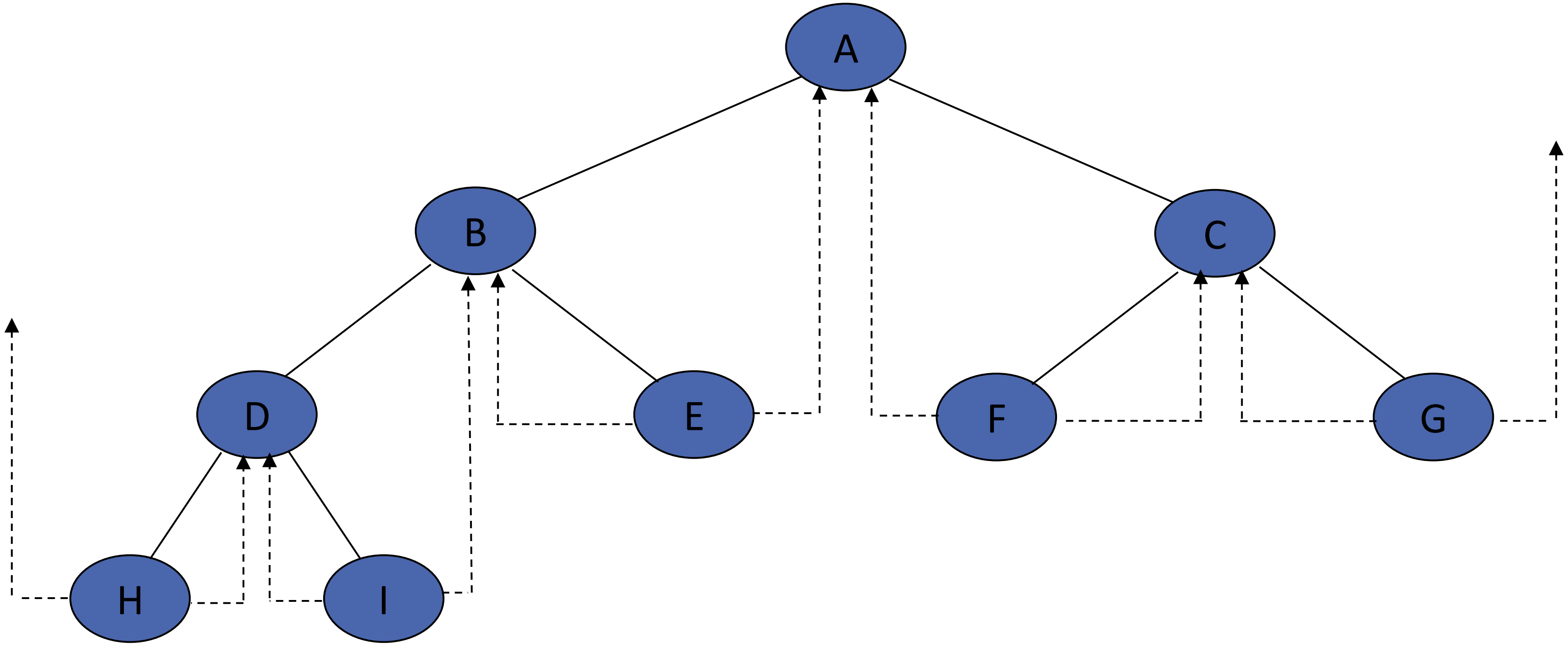
- According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.
- And binary tree with such pointers are called threaded tree.
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and 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



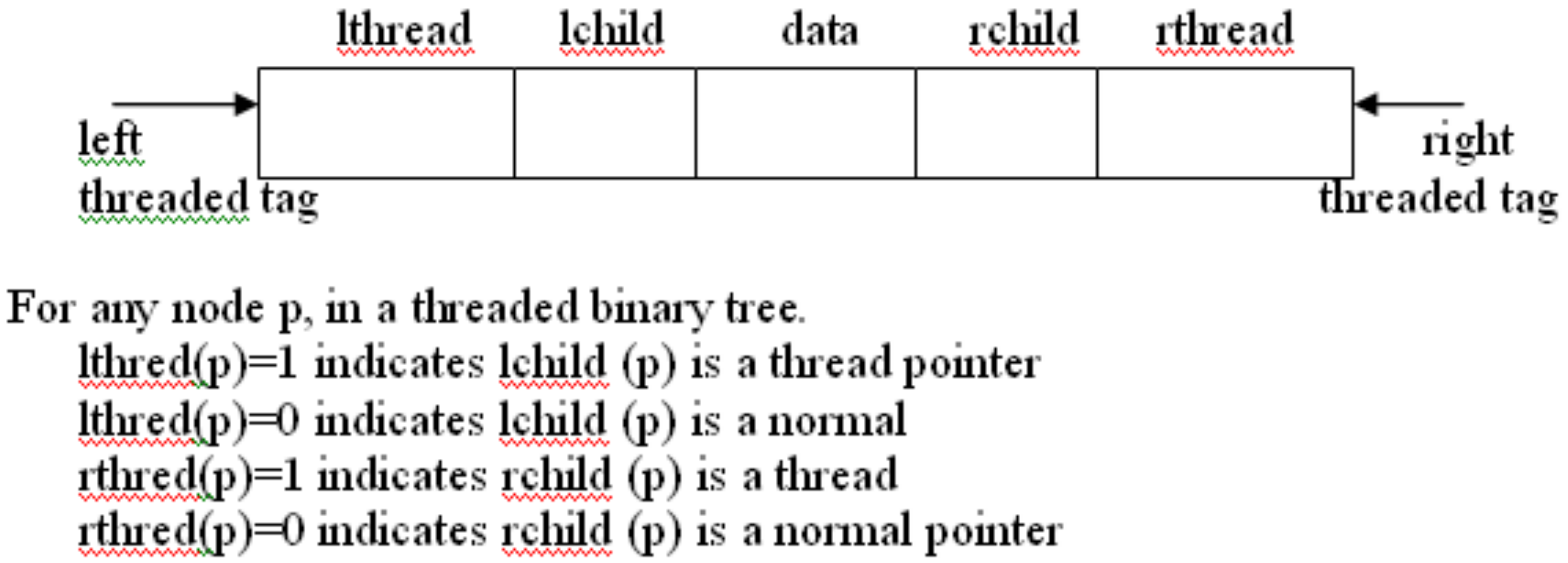
Inorder sequence: H, D, I, B, E, A, F, C, G

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)=1 indicates lchild (p) is a thread pointer

lthred(p)=0 indicates lchild (p) is a normal

rthred(p)=1 indicates rchild (p) is a thread

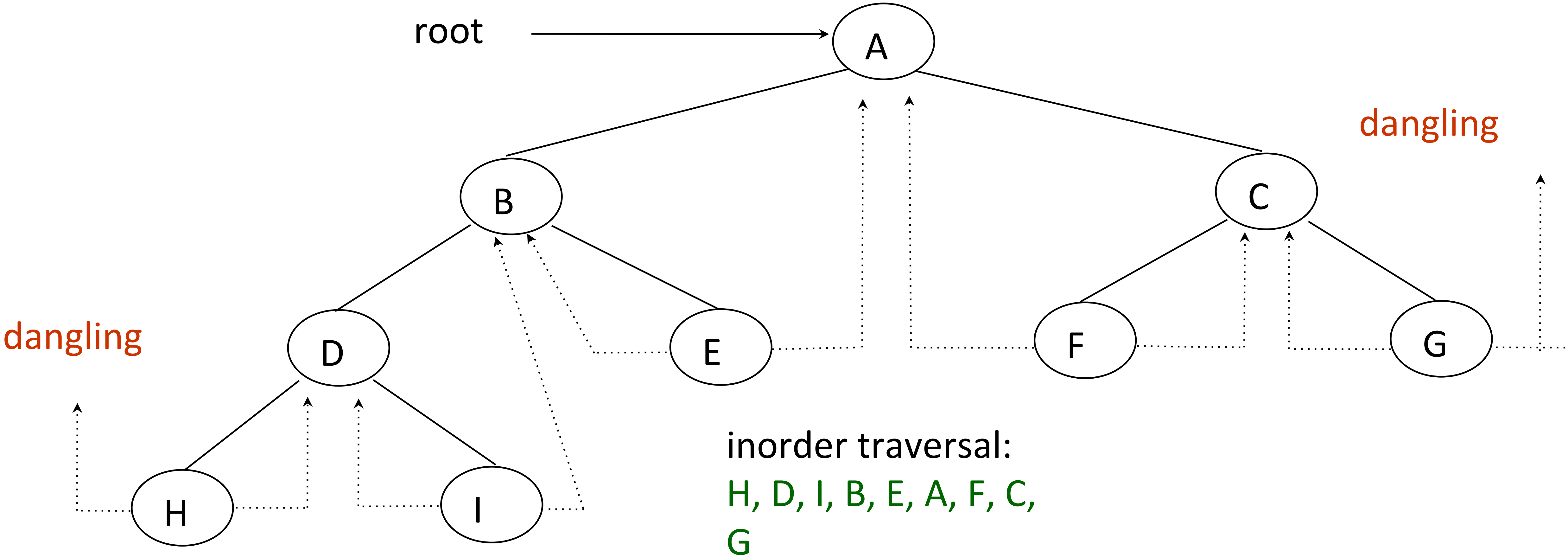
rthred(p)=0 indicates rchild (p) is a normal pointer

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:

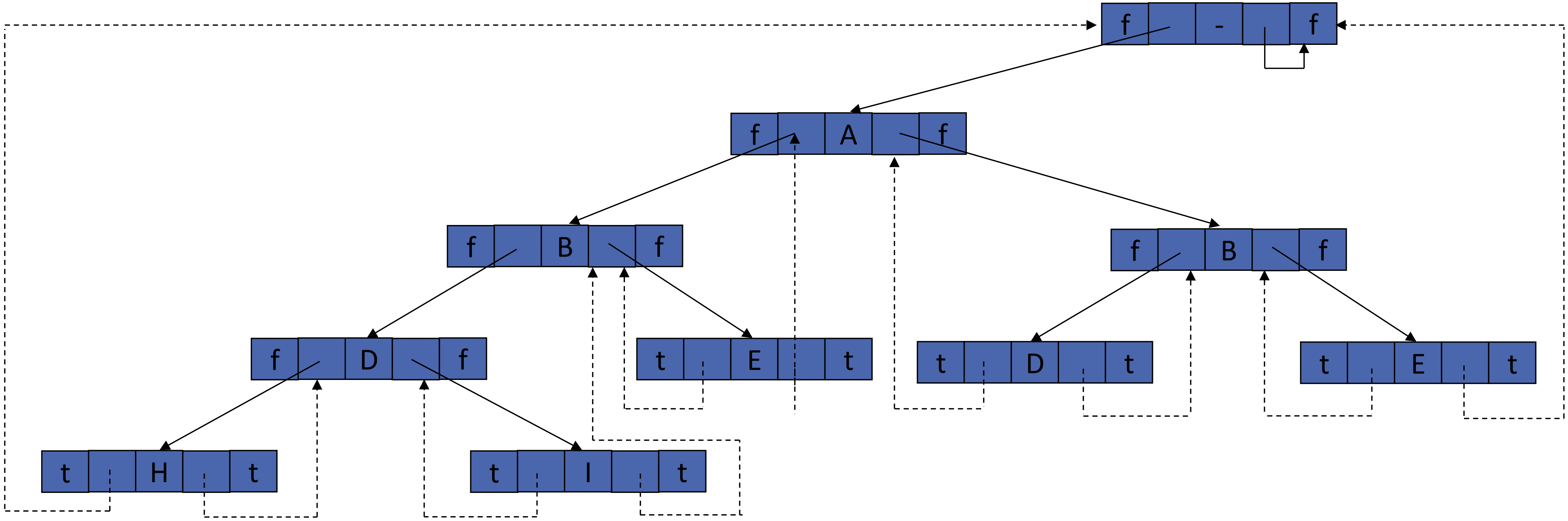
H, D, I, B, E, A, F, C, G

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

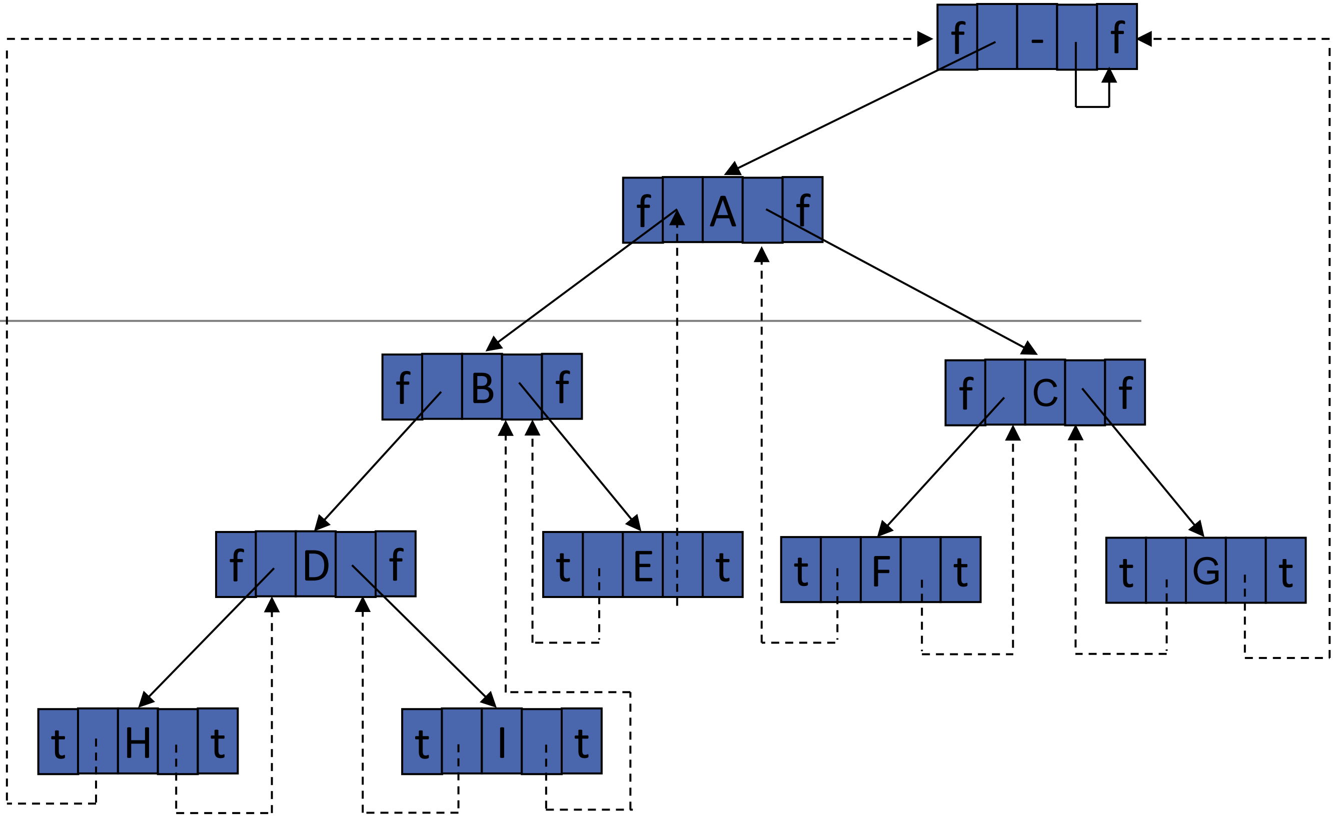


```
class tbtnode
{
  char data;
  bool rbit;
  bool lbit;
  tbtnode *rightc;
  tbtnode *leftc;
  friend class tbt;
 public:
        tbtnode();
};

tbtnode::tbtnode()
{
  lbit=1;
  rbit=1;
}
```

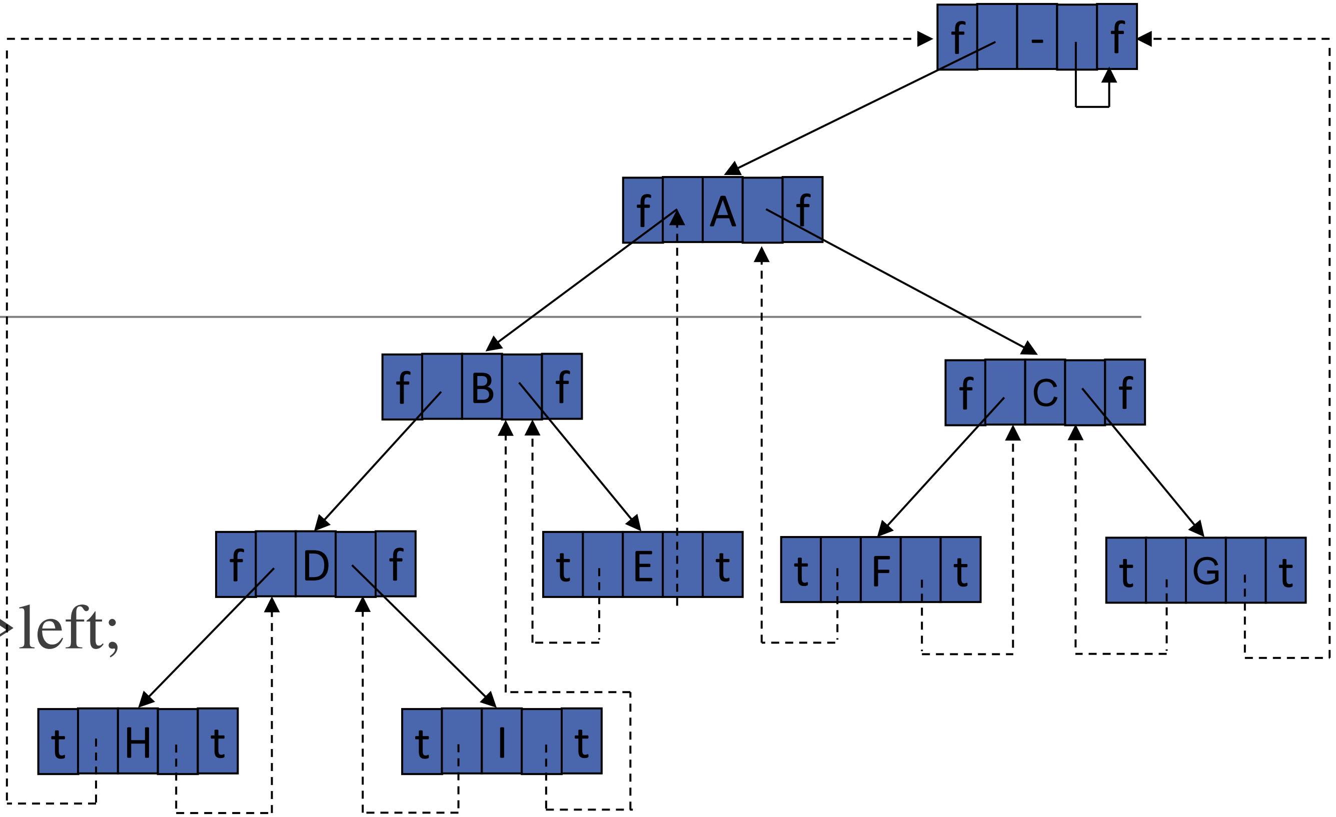
```
class tbt
{
  tbtnode *head;
  public:
  void create();
  void preorder();
  tbtnode* presuccr(tbtnode *temp);
  void inorder();
  tbtnode* insuccr(tbtnode *temp);
  tbt();
};

tbt::tbt()
{
  Allocate memory for head;
  Set rbit to 0;
  Assign head->left and right to head;
}
```

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

do
{
    Initialize  flag to 0;
    temp=root;
    Allocate memory to curr  and accept curr->data;
while(flag==0)
 {
     Accept choice left or right;
   if  ch1='l'
   {
       if(temp->lbit==1)
     {

          curr->rightc=temp;
          curr->leftc=temp->leftc;
          temp->leftc=curr;
          temp->lbit=0;
          flag=1;
    }
   else
          temp=temp->leftc;

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

}while(ch=='y');

} //end algo
```

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

}
Algorithm node * inordersucc(temp)
{
      x=temp->right;
     if(temp->rbit==0)
       {
         while(x->lbit==0)
                  x=x->left;
        }
      return x;
}
```



```
Algorithm preorder()
{
 Assign temp to head->left;
 while(temp != head)
 {
        print temp->data;
        while(temp->lbit != 1)
        {
               move temp to temp->left;
               print temp->data;
        }
        while(temp->rbit == 1)
                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.

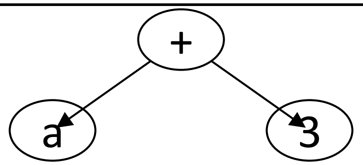
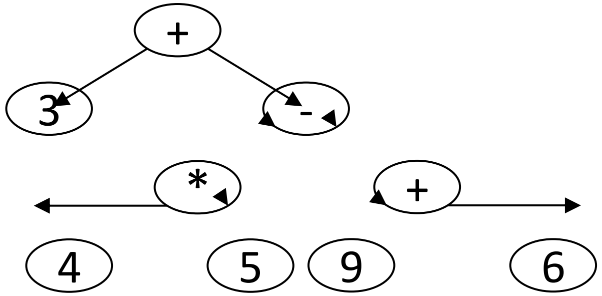
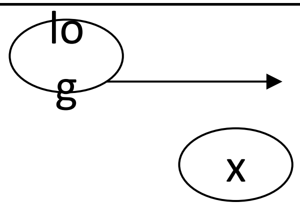
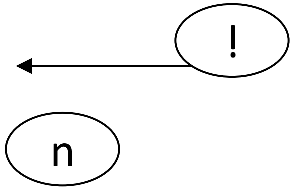
What is an Expression tree?

An expression tree for an arithmetic, relational, or logical expression is a binary tree in which:

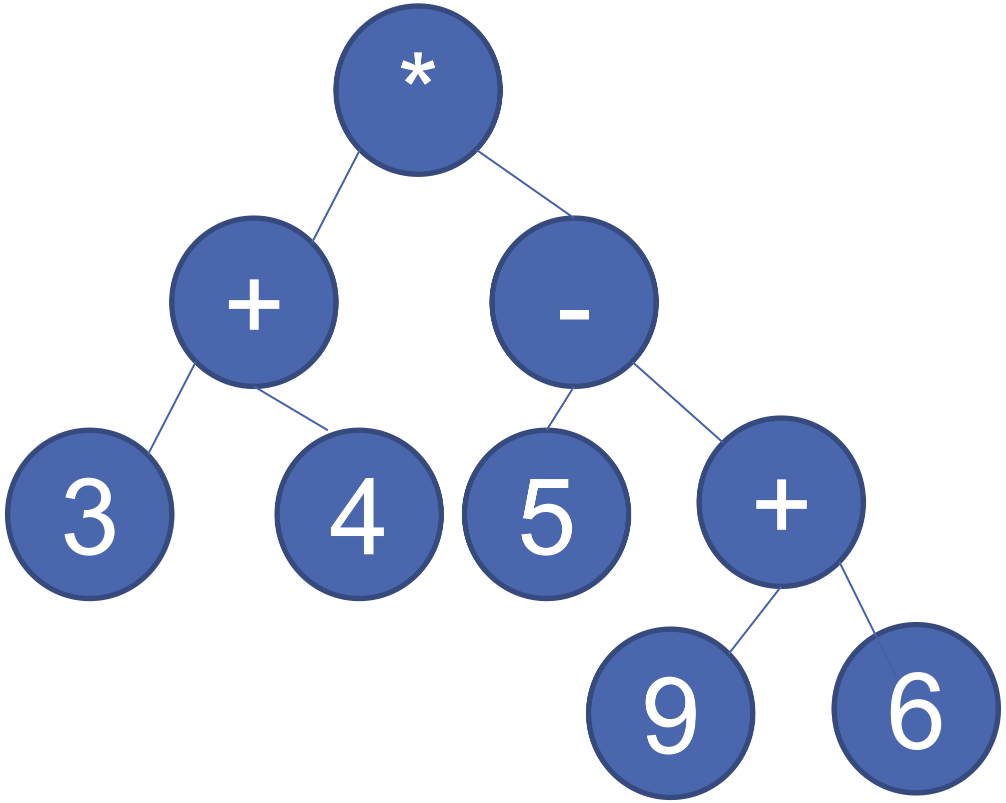
- The parentheses in the expression do not appear.
- The leaves are the variables or constants in the expression.
- The non-leaf nodes are the operators in the expression:
  - A node for a binary operator has two non-empty subtrees.
  - A node for a unary operator has one non-empty subtree.

The operators, constants, and variables are arranged in such a way that an inorder traversal of the tree produces the original expression without parentheses.

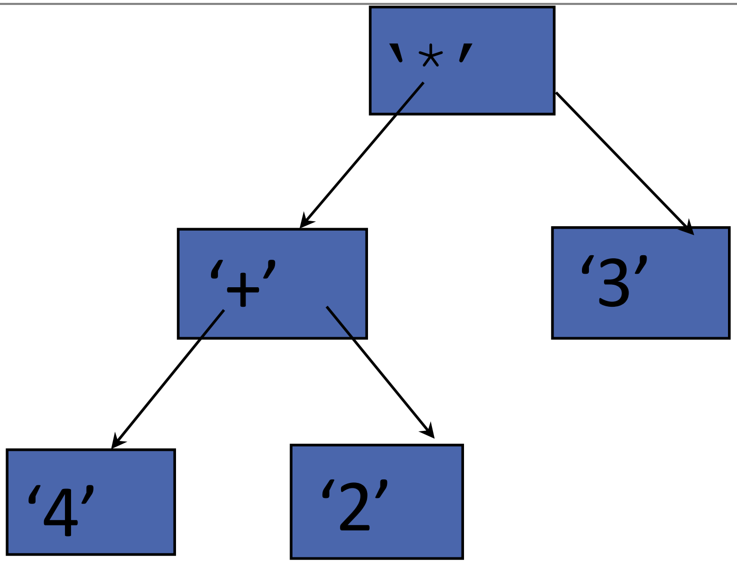
Expression Tree Examples

Inorder Traversal Result	Expression Tree	Expression
a + 3		(a+3)
3+4*5-9+6		3+(4*5-(9+6))
log x		log(x)
n !		n!

3+4*5-9+6



A Binary Expression Tree



What value does it have?

(4 + 2) * 3 = 18

Why Expression trees?

Expression trees are used to remove ambiguity in expressions.

Consider the algebraic expression 2 - 3 * 4 + 5.

Without the use of precedence rules or parentheses, different orders of evaluation are possible:

((2-3)*(4+5)) = -9

((2-(3*4))+5) = -5

(2-((3*4)+5)) = -15

(((2-3)*4)+5) = 1

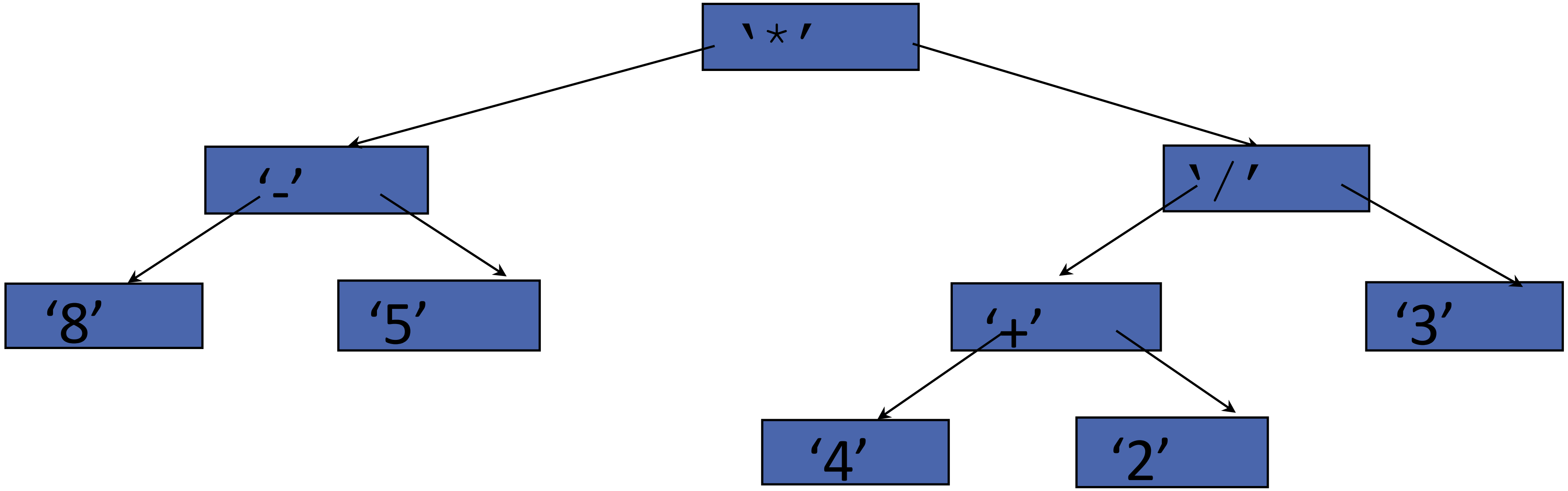
(2-(3*(4+5))) = -25

The expression is ambiguous because it uses infix notation: each operator is placed between its operands.

Why Expression trees? (contd...)

- Storing a fully parenthesized expression, such as ((x+2)-(y*(4-z))), is wasteful, since the parentheses in the expression need to be stored to properly evaluate the expression.
- A compiler will read an expression in a language like Java, and transform it into an expression tree.
- Expression trees impose a hierarchy on the operations in the expression. Terms deeper in the tree get evaluated first. This allows the establishment of the correct precedence of operations without using parentheses.
- Expression trees can be very useful for:
  - Evaluation of the expression.
  - Generating correct compiler code to actually compute the expression's value at execution time.
  - Performing symbolic mathematical operations (such as differentiation) on the expression.

Easy to generate the infix, prefix, postfix expressions (how?)

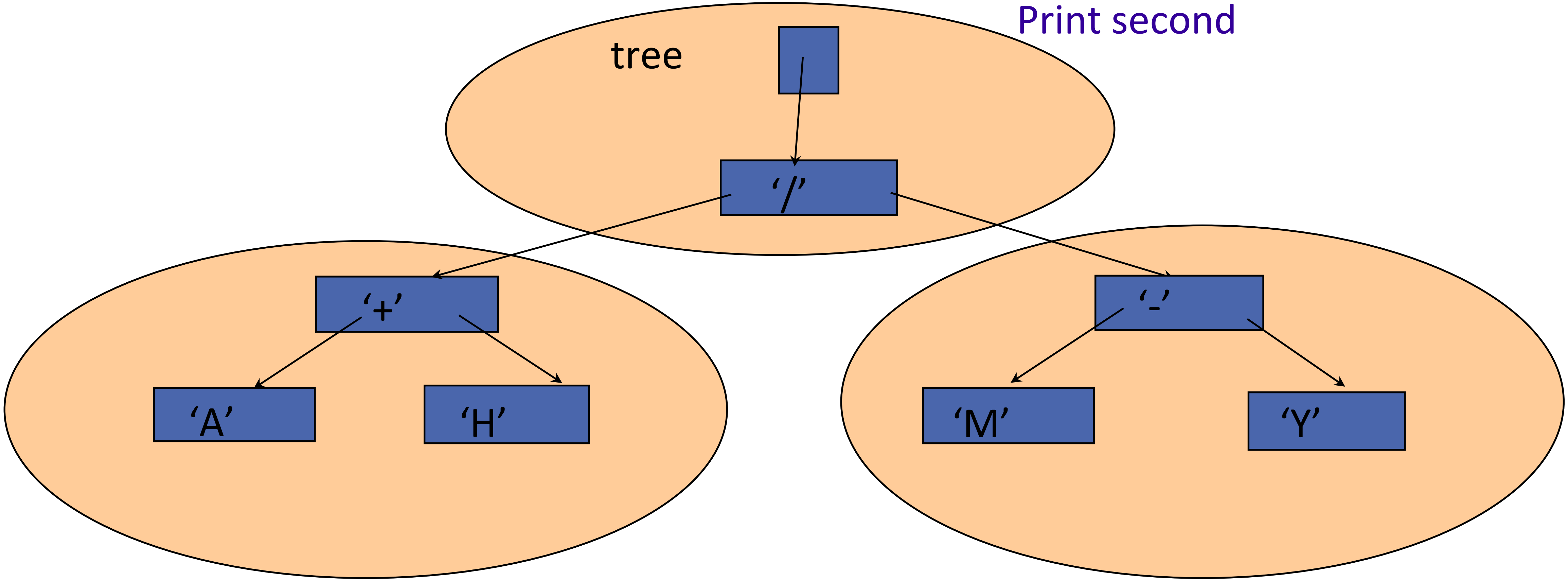


Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / *

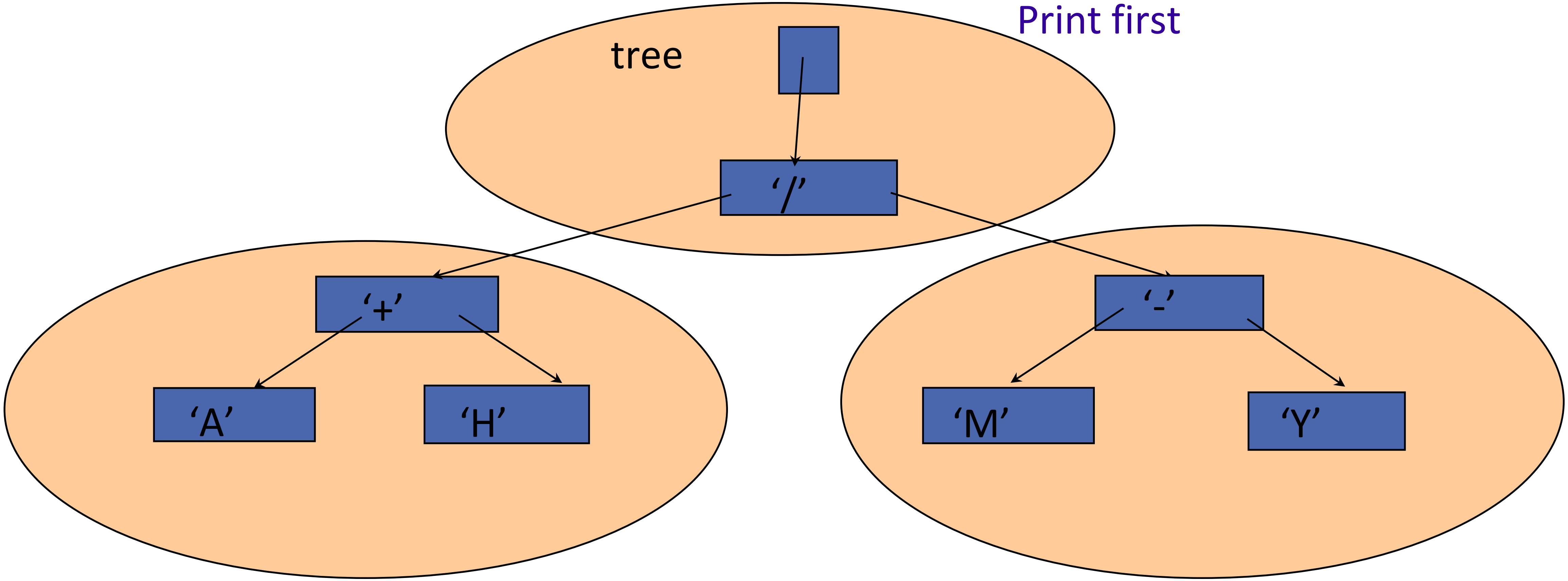
Inorder Traversal: (A + H) / (M - Y)



Print left subtree first

Print right subtree last

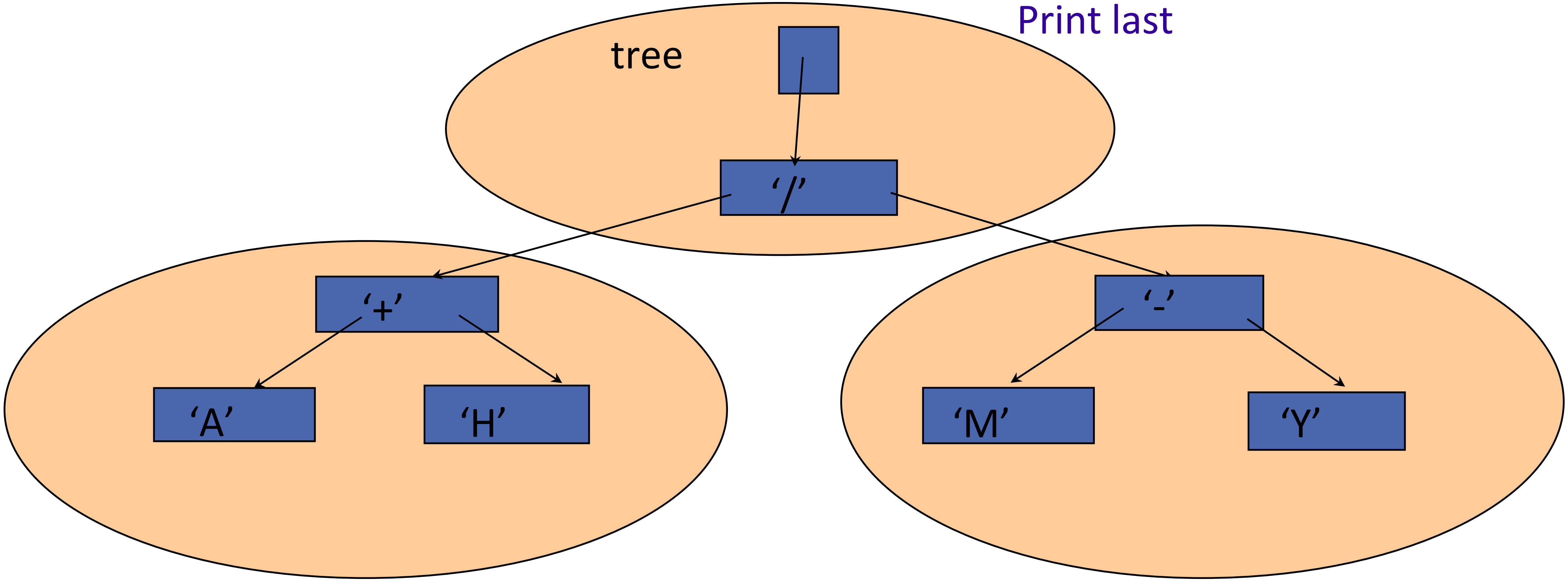
Preorder Traversal: / + A H - M Y



Print left subtree second

Print right subtree last

Postorder Traversal: A H + M Y - /



Print left subtree first

Print right subtree second

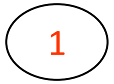
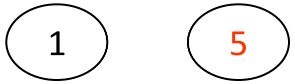
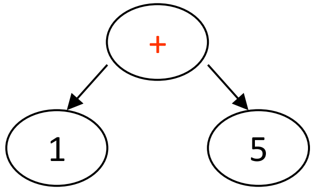
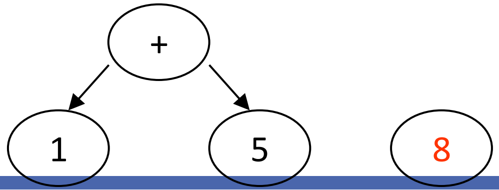
Building an Expression Tree

Procedure ExpressionTree(E)

//E is an expression in postfix notation.

```
begin
    for i=1 to |E| do
        if E[i] is an operand then
            create a node  for the operand;
            add it to the stack
        else if E[i] is an operator then
             create a node  for the operator(root/parent)

             pop from the stack ;

             make the operand the right subtree of the operator node

             pop from the stack;

                  make the operand the left subtree of the operator node
             add it to the stack
    end-for
```

Convert RPN expression to expression tree

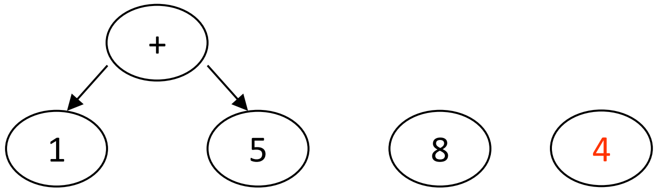
	stack	comments	Binary tree
↑	1 ← top	Create leaf node 1 and push address onto stack	
↑	5 ← top 1	Create leaf node 5 and push address onto stack	
$15 + 841 - - \times$ ↑	← top	Create node “+” and pop 5, 1 from stack as its children.	
	+ ← top	Push address of node “+” to stack	
$5 + 841 - - \times$ ↑	8 ← top +	Create leaf node 8 and push address onto stack	

Convert RPN expression to expression tree contd…

stack | comments | Binary tree

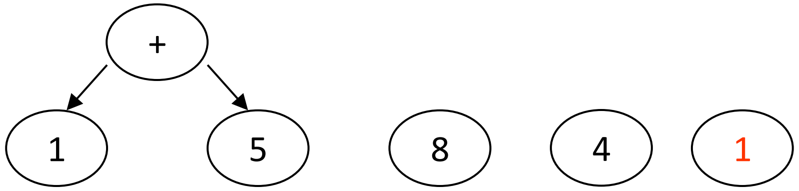
4	← top
8	
+	

Create leaf node 4 and push address onto stack



1	← top
4	
8	
+	

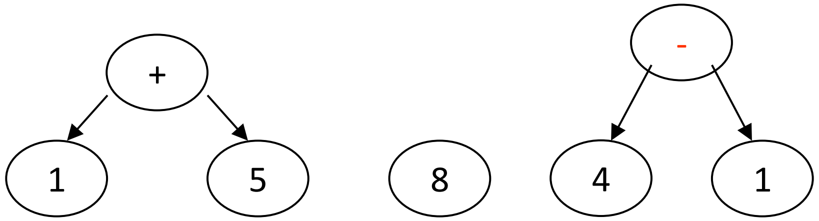
Create leaf node 1 and push address onto stack



41 − − ×

8	← top
+	

Create node “-” and pop 1, 4 from stack as its children.



-	← top
8	
+	

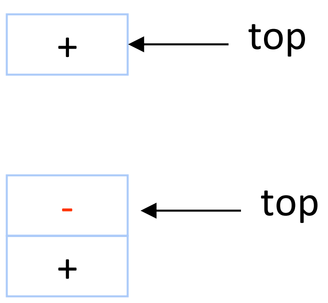
Push node ‘-’ onto stack

Convert RPN expression to expression tree contd…

stack

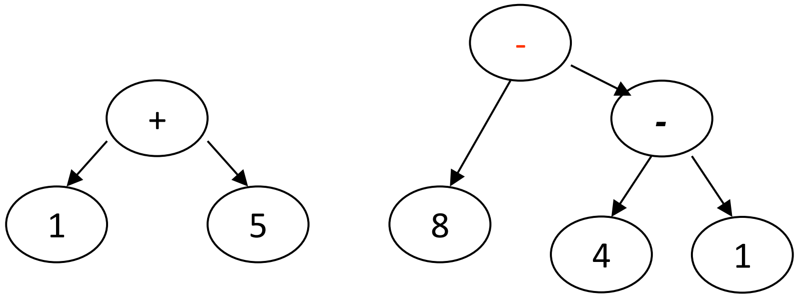
comments

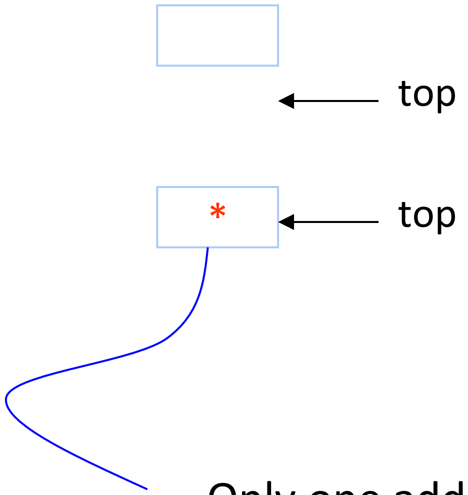
Binary tree



Create node “-” and pop “-”, 8 from stack as its children.

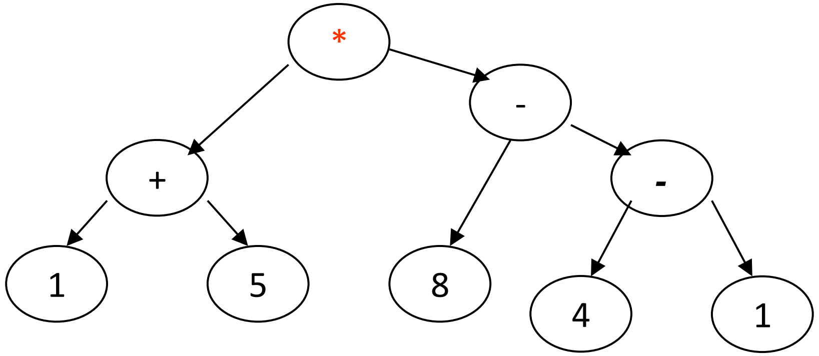
Push node “-” onto stack





Create node “*” and pop “-”, “+” from stack as its children.

Push node “*” onto stack



Only one address on the stack, this address is ***root*** of the tree

Practice Problems

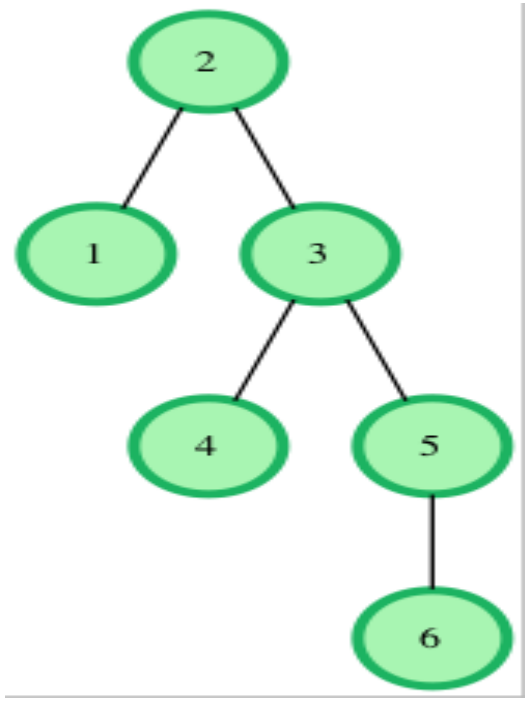
1.Given a pointer to the root of a binary tree, print the top view of the binary tree.

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
```