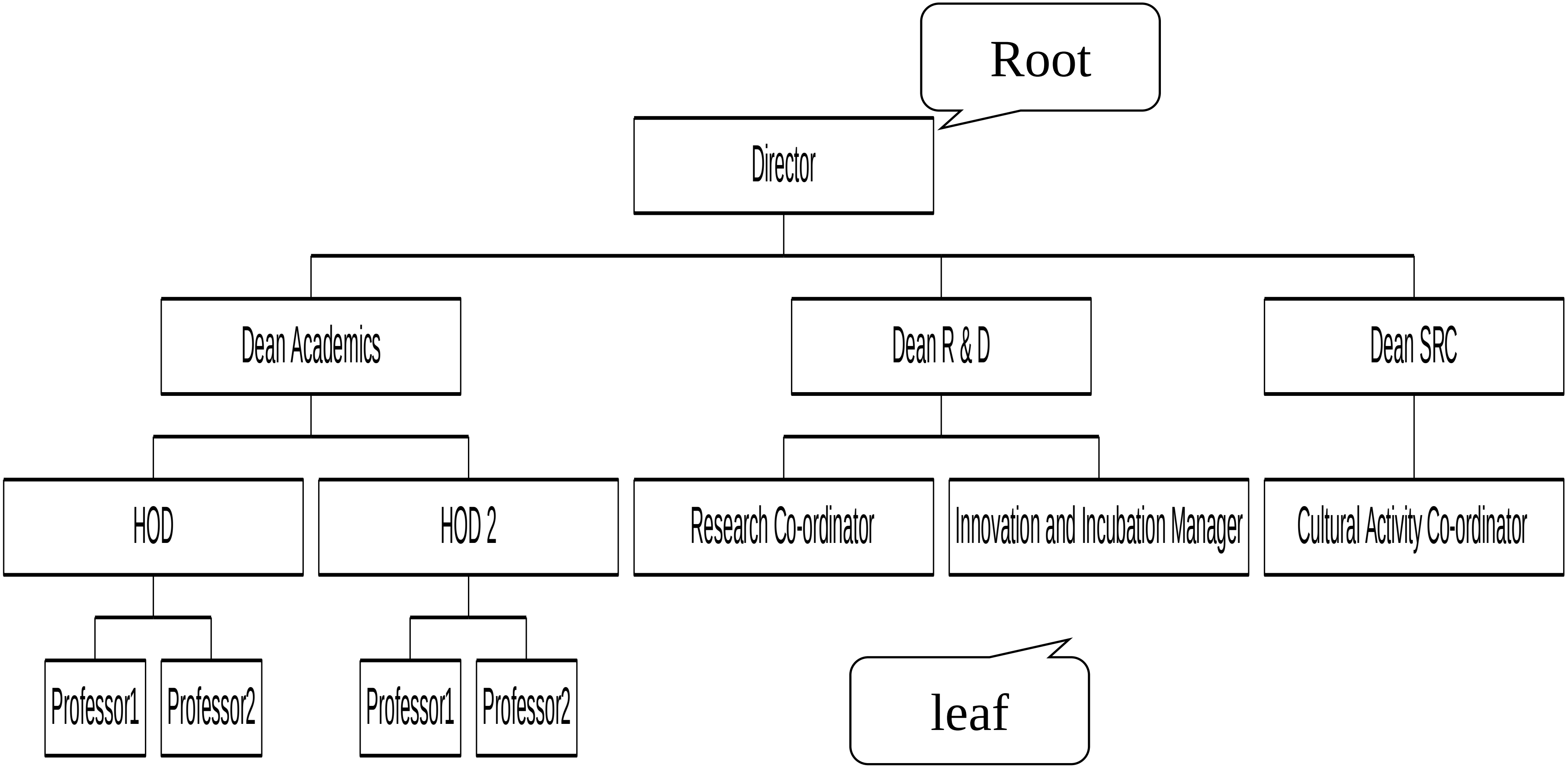
Unit 5

Trees

Trees

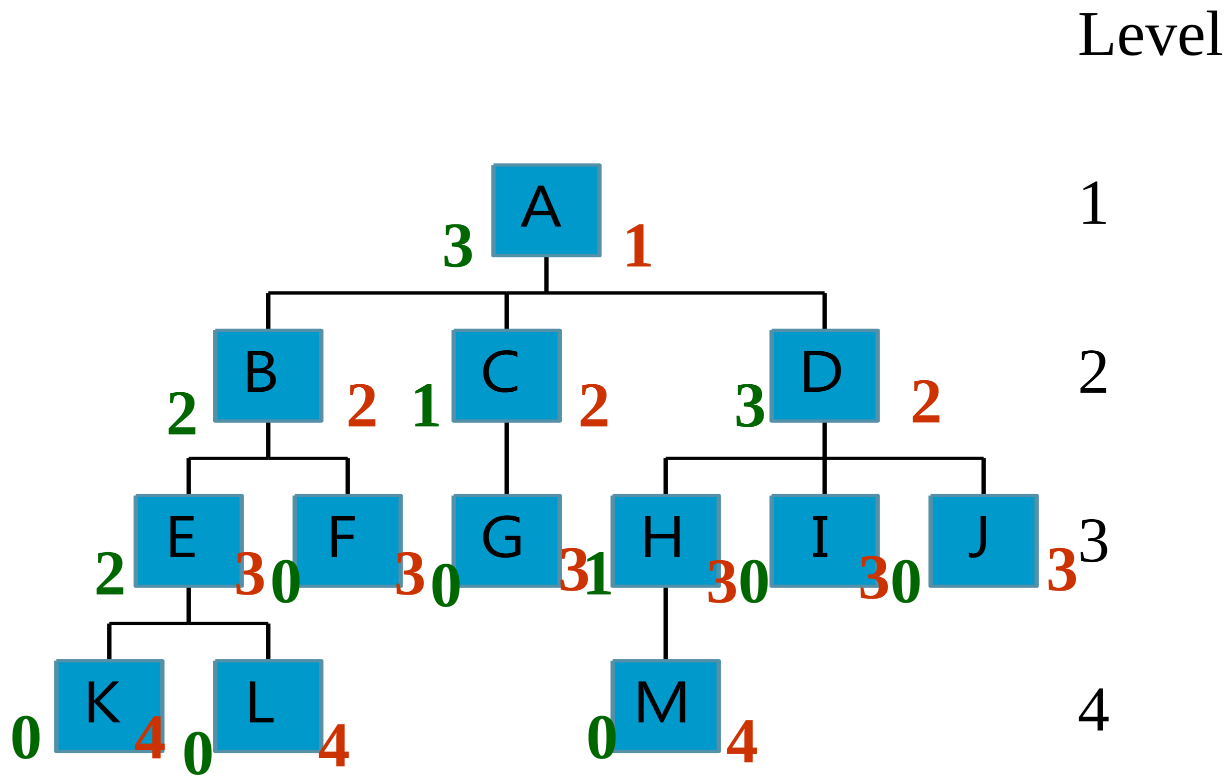

Root
Director
Dean Academics
Dean R & D
Dean SRC
HOD
HOD 2
Research Co-ordinator
Innovation and Incubation Manager
Cultural Activity Co-ordinator
Professor1
Professor2
Professor1
Professor2
leaf

Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root
- The remaining nodes are partitioned into n>=0 disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree
- We call $T_1, \ldots, T_n$ the subtrees of the root

Level and Depth

Node (13)
Degree of a node
leaf (terminal)
nonterminal
parent
children
sibling
degree of a tree (3)
ancestor
Level of a node
Height of a tree
Depth of a tree



Terminology

- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
  - Highest degree of a node is the degree of the tree
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.

Terminology

- ***Level*** of a node

  A measure of its distance from the root:

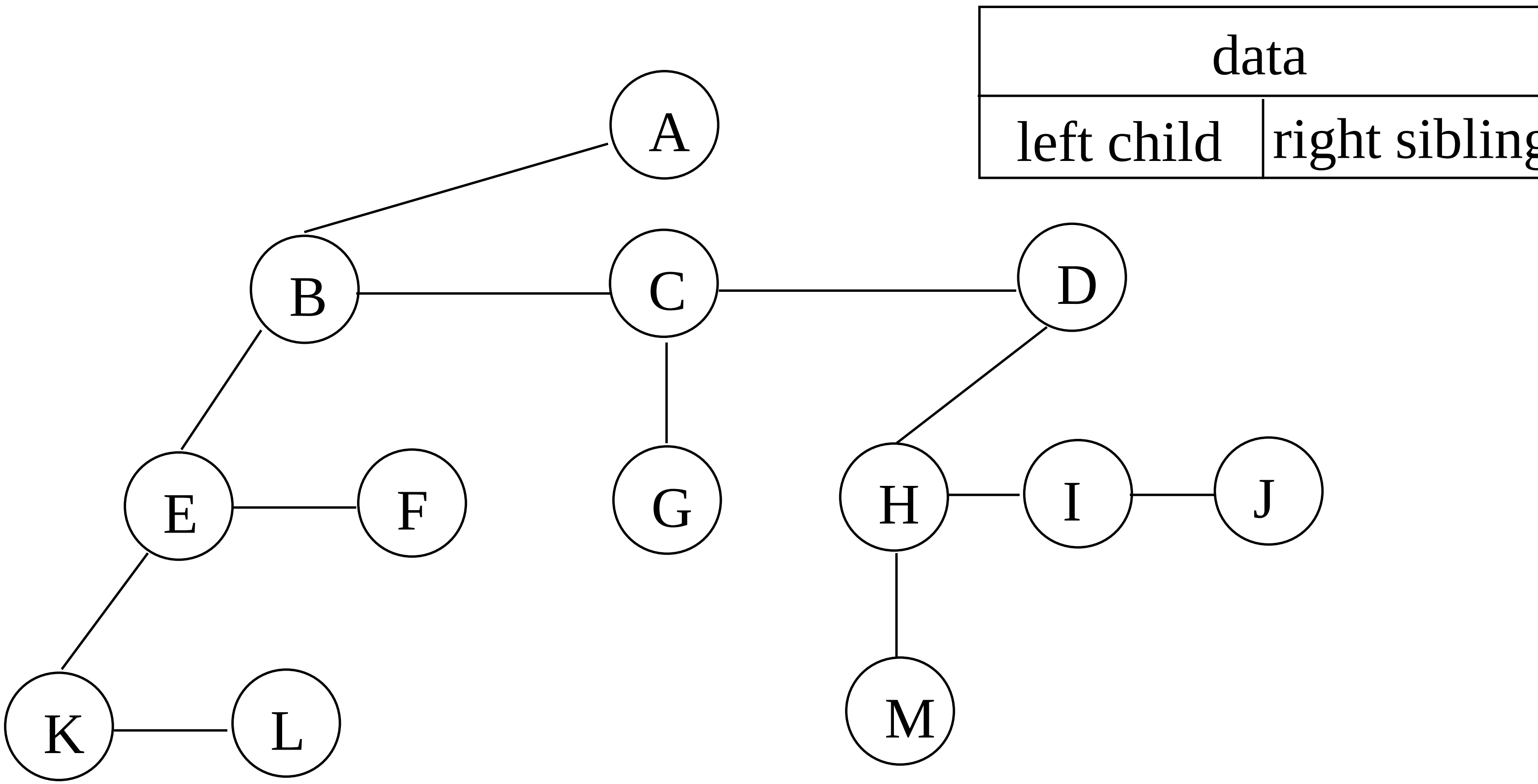
  Level of the root = 1

  Level of other nodes = 1 + level of parent
- Height of a Node
  - The no. of edges on the path from a node to the deepest leaf [plus 1]
- Depth of a node
  - The no. of edges on the path from root to that node [plus 1]

General (Non-Binary) Tree to Binary Tree

- Root of General Tree = Root of Binary tree
- Left child of a node in general tree = Left child of a node in binary tree
- Right sibling of a node in general tree = right child of that node in binary tree

Left Child - Right Sibling



Binary Trees

- 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***
- Any tree can be transformed into binary tree
  – by left child-right sibling representation
- The left subtree and the right subtree are distinguished

Abstract Data Type Binary Tree

Structure *Binary_Tree*(abbreviated *BinTree*) is

object: **a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree***

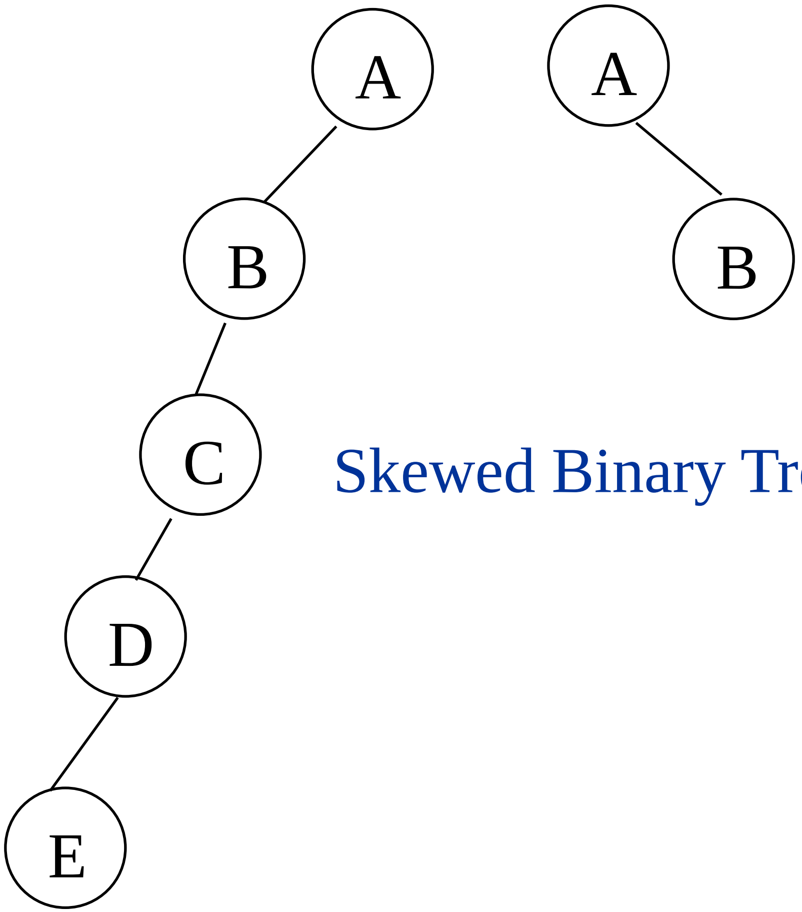
Functions:

Boolean **IsEmpty**(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

Abstract Data Type Binary Tree

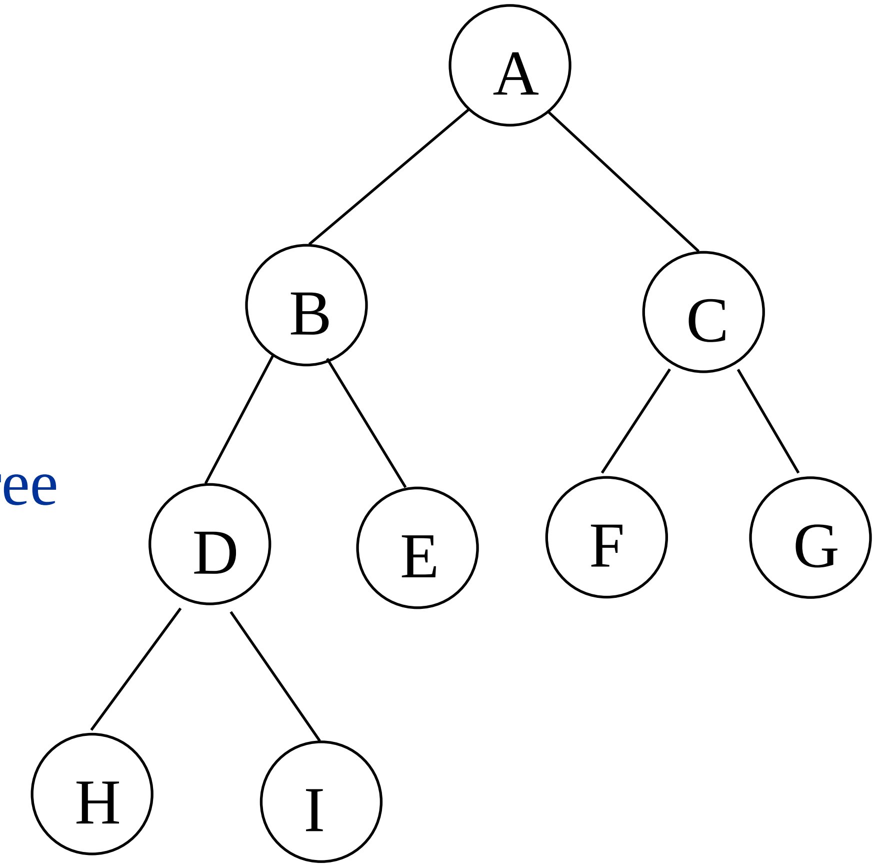
- *BinTree* **MakeBT**(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*
- *Bintree* **Lchild**(*bt*)::= if (IsEmpty(*bt*)) return error
  else return the left subtree of *bt*
- *element* **Data**(*bt*)::= if (IsEmpty(*bt*)) return error
  else return the data in the root node of *bt*
- *Bintree* **Rchild**(*bt*)::= if (IsEmpty(*bt*)) return error
  else return the right subtree of *bt*

Samples of Trees



Skewed Binary Tree

Complete Binary Tree



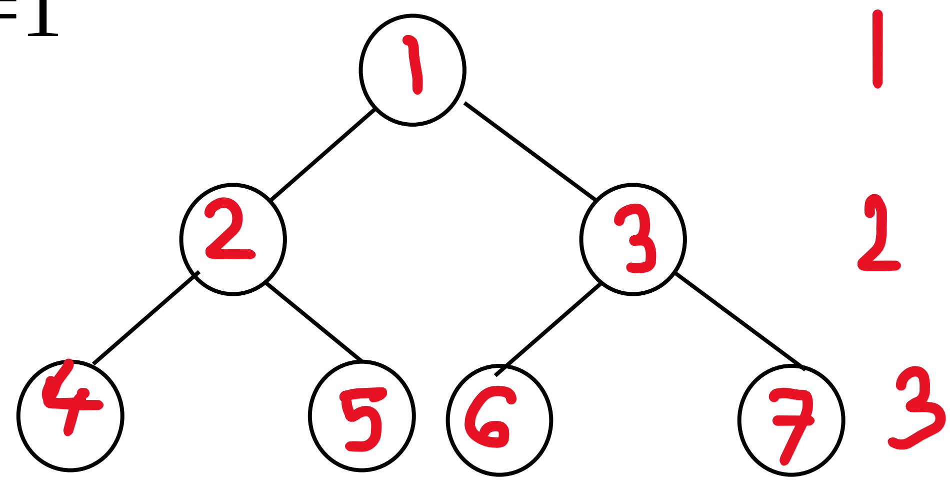
Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i>=1$

$2^2 = 4$

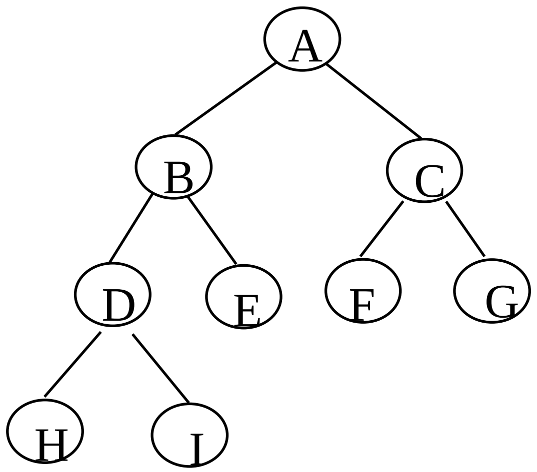
- The maximum number of nodes in a binary tree of depth k is 2^k-1, $k>=1$

$2^3 - 1 = 7$

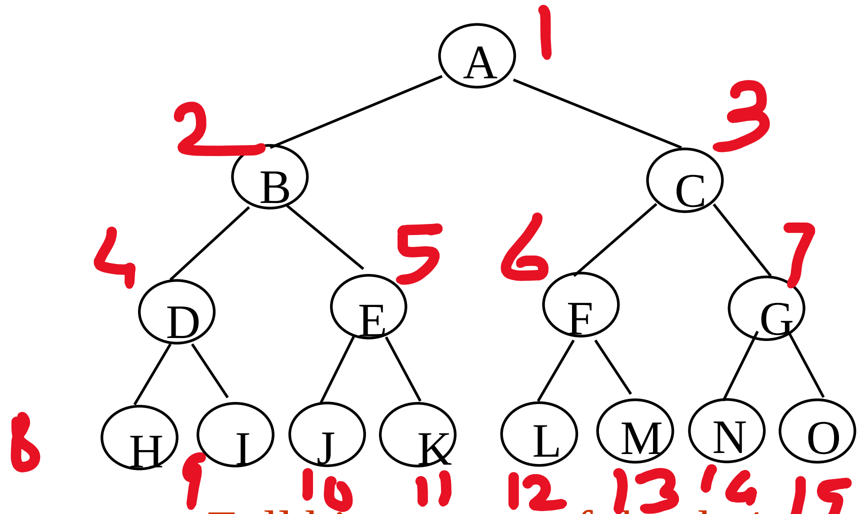


Full BT VS Complete BT

- A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, k>=0.
- A binary tree with n nodes and depth k is complete *iff* its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.



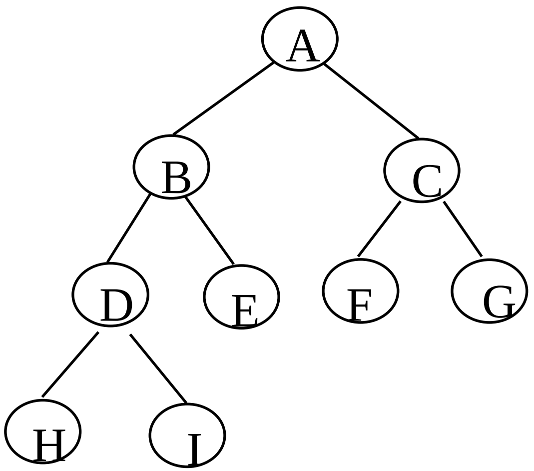
Complete binary tree



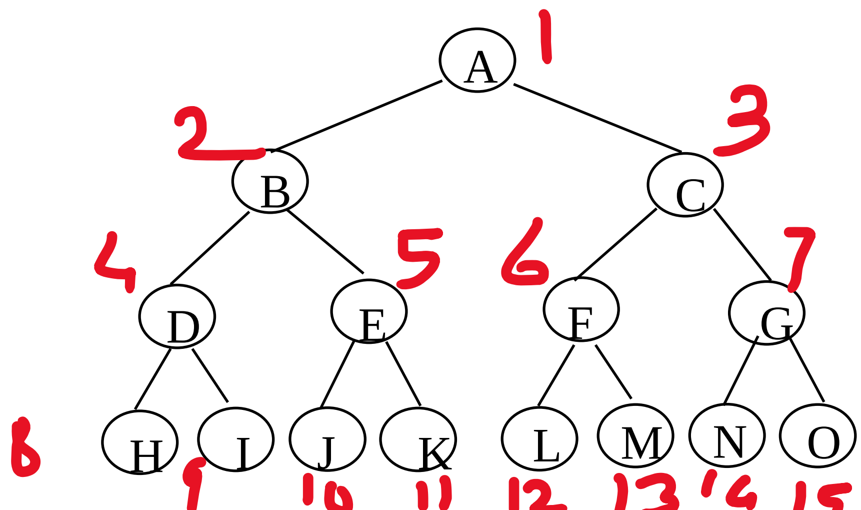
Full binary tree of depth 4

Full BT VS Complete BT

k



Complete binary tree

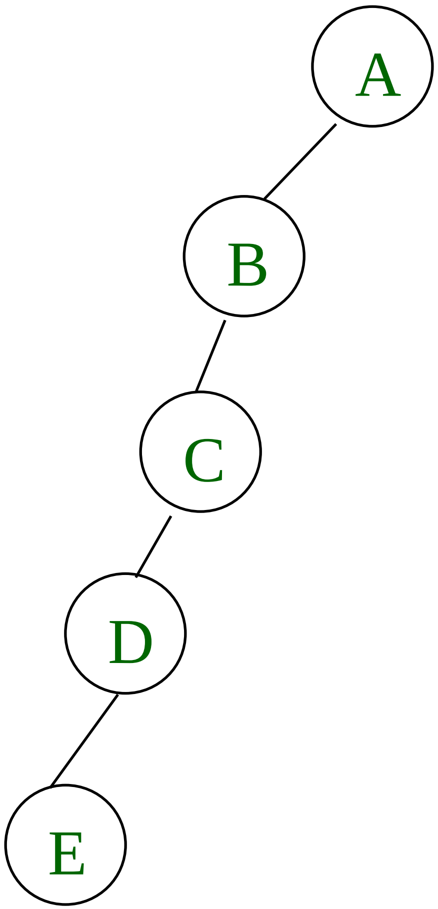


Full binary tree of depth 4

Binary Tree Representations

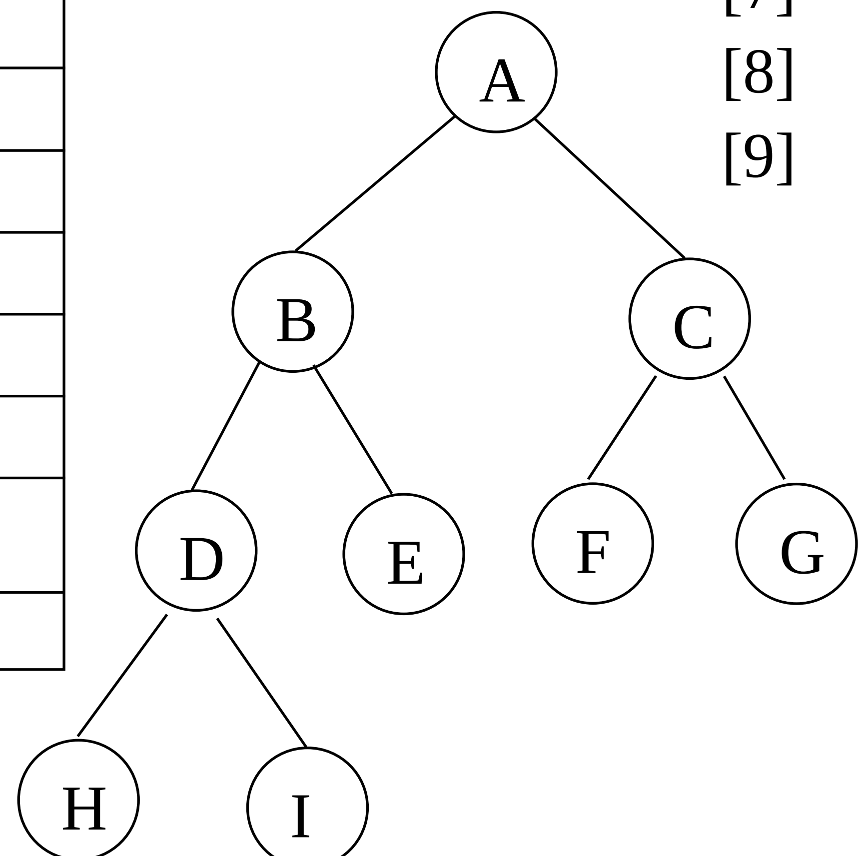
- If a complete binary tree with n nodes (depth = $\log n + 1$) is represented sequentially, then for any node with index i, we have:
  - *parent*(i) is at $i/2$ if i!=1. If i=1, i is at the root and has no parent.
  - *left_child*(i) is at 2*i, if 2i<=n, else i has no left child*
  - *right_child*(i) is at 2*i*+1, if 2i+1<=n , *else i has no right child*

Sequential Representation



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

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

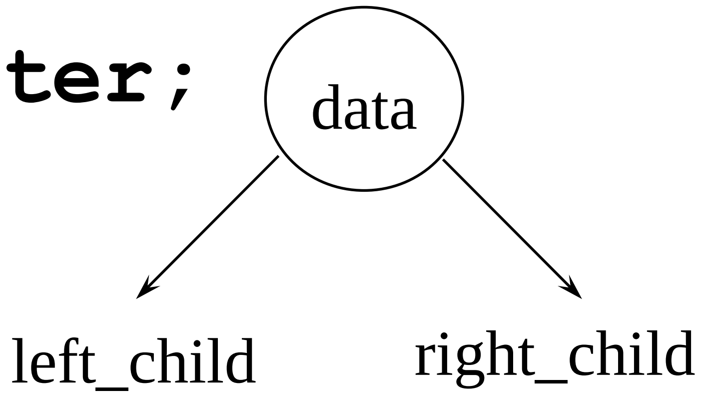


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

```
struct node {
   int data;
   struct Node * left_child;
    struct Node *right_child;
};
struct node *tree_pointer;
```

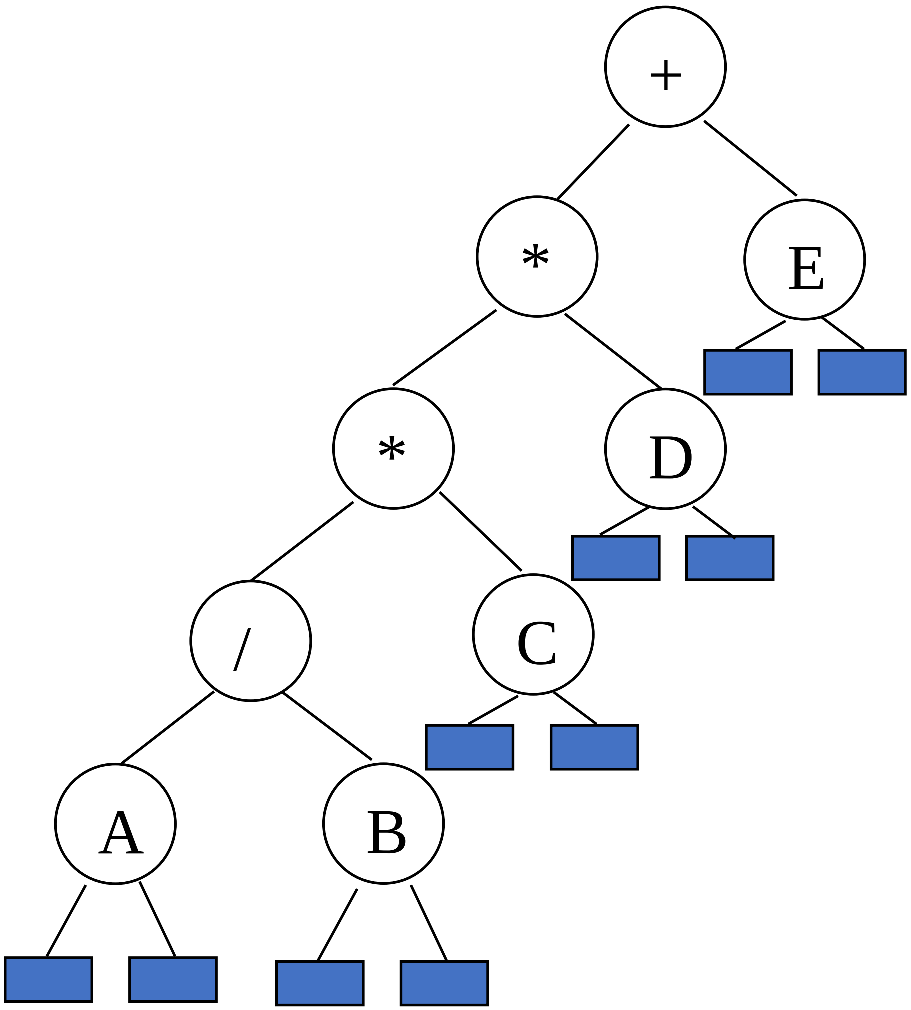
left_child	data	right_child



Binary Tree Traversals

- Let L, V, 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**

Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        indorder(ptr->right_child);
    }
}
```

A / B * C * D + E

Examples

- Covered in class

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf(“%d”, ptr->data);
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postdorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

A B / C * D * E +

Iterative Inorder Traversal

(using stack)

```
void iter_inorder(struct Node * curr){
   Stack s;
   createStack(s);/* initialize stack */
  while(true){
    while(curr<>NULL)
       add(&top, curr);/* add to stack */
       curr=curr.left_child
    curr = pop(stack)  /* delete from stack */
    if (curr==NULL) break; /* empty stack */
    print curr.data
    curr = curr.right_child;
 }
}
```

O(n)

Level Order Traversal

```
void level_order(tree_pointer ptr){
/* level order tree traversal */
   int front = rear = 0;
   Queue q;
   createQueue(q)
  if (!ptr) return; /* empty queue */
   add(q,ptr);
   for (;;) {
     ptr = delete(q);
```

```
    if (ptr) {
      printf("%d", ptr->data);
      if (ptr->left_child)
        add(q,ptr->left_child);
      if (ptr->right_child)
          add(q,ptr->right_child);
    }else
      break;
  }
}
```

+ * E * D / C A B

Create Linked Binary from Array

- Refer the shared code

Copying Binary Trees

```
tree_pointer copy(tree_pointer original){
 tree_pointer temp;
 if (original) {
  temp <= createNode()
  if (temp=NULL) {
   print “the memory is full”
  }else{
   temp.left_child=copy(original.left_child);
   temp.right_child=copy(original.right_child);
   temp.data=original.data;
   return temp;
  }
 }
 return NULL;
}
```

postorder

Equality of Binary Trees

```
int equal(tree_pointer first, tree_pointer second){
/* function returns FALSE if the binary trees first and
   second are not equal, otherwise it returns TRUE */

  return ((!first && !second) || (first && second &&
       (first->data == second->data) &&
       equal(first->left_child, second->left_child) &&
       equal(first->right_child, second->right_child))
}
```

Threaded Binary Trees

By: A.J. Perlis and C. Thornton

- Two many null pointers in current representation of binary trees

  n: number of nodes

  number of non-null links: n-1

  total links: 2n

  null links: 2n-(n-1)=n+1

- Replace these null pointers with some useful "threads".

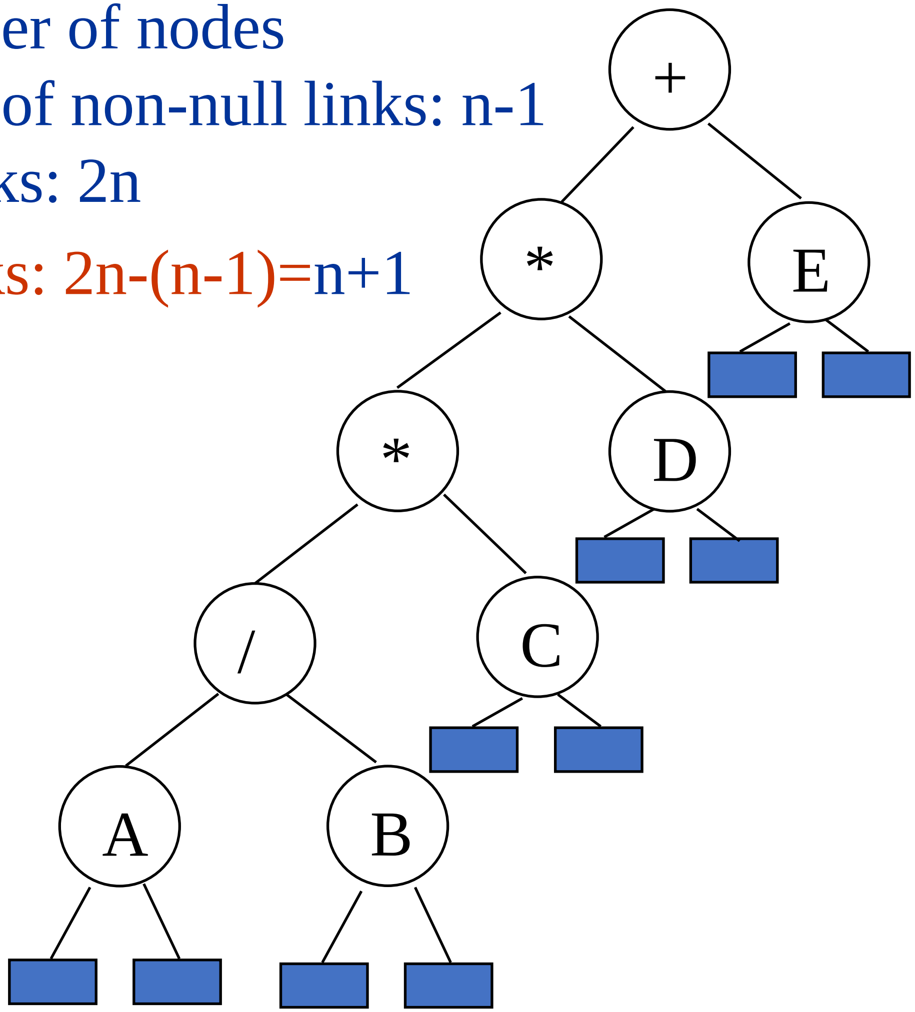
Threaded Binary Trees

n: number of nodes

number of non-null links: n-1

total links: 2n

null links: 2n-(n-1)=n+1

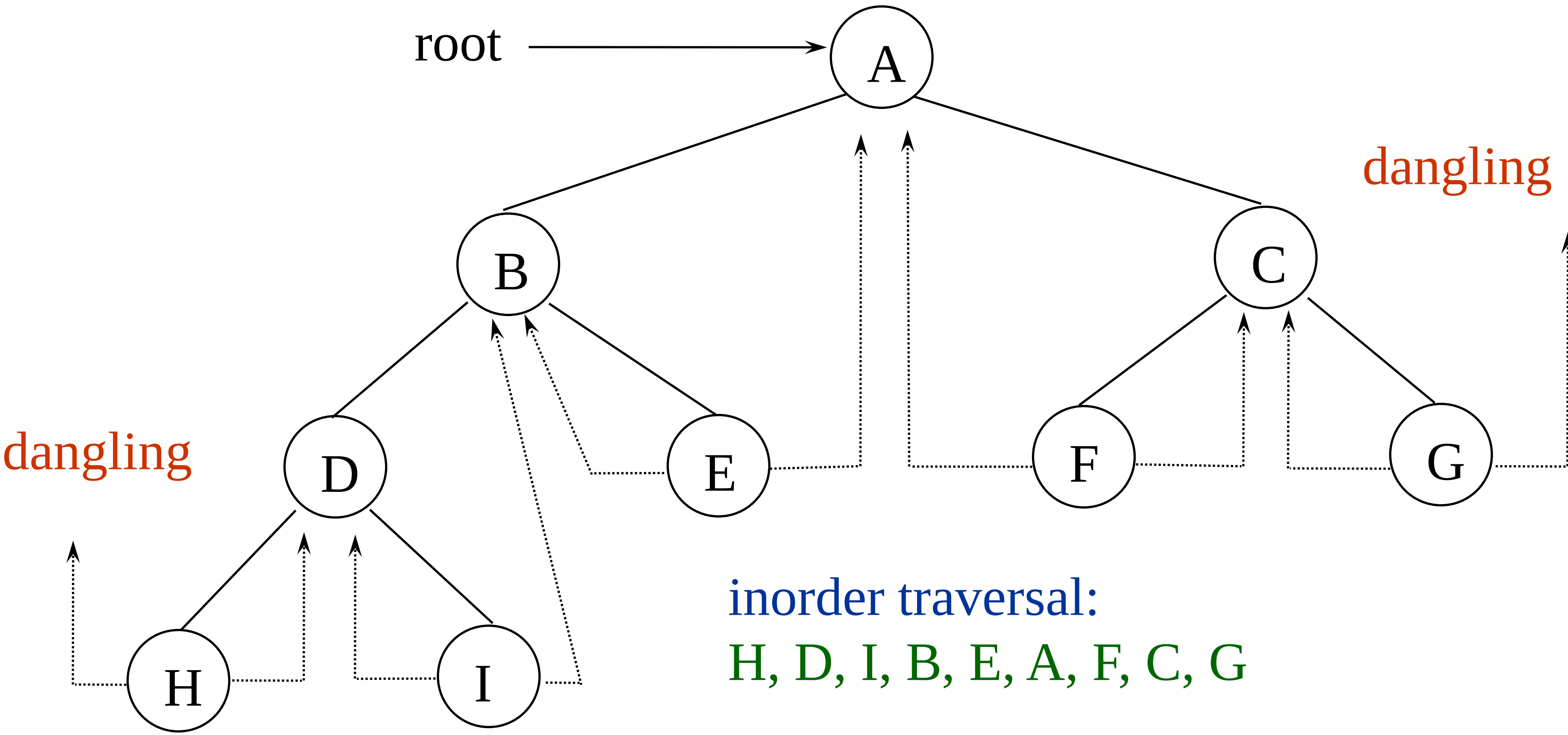


Threaded Binary Trees (*Continued*)

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

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

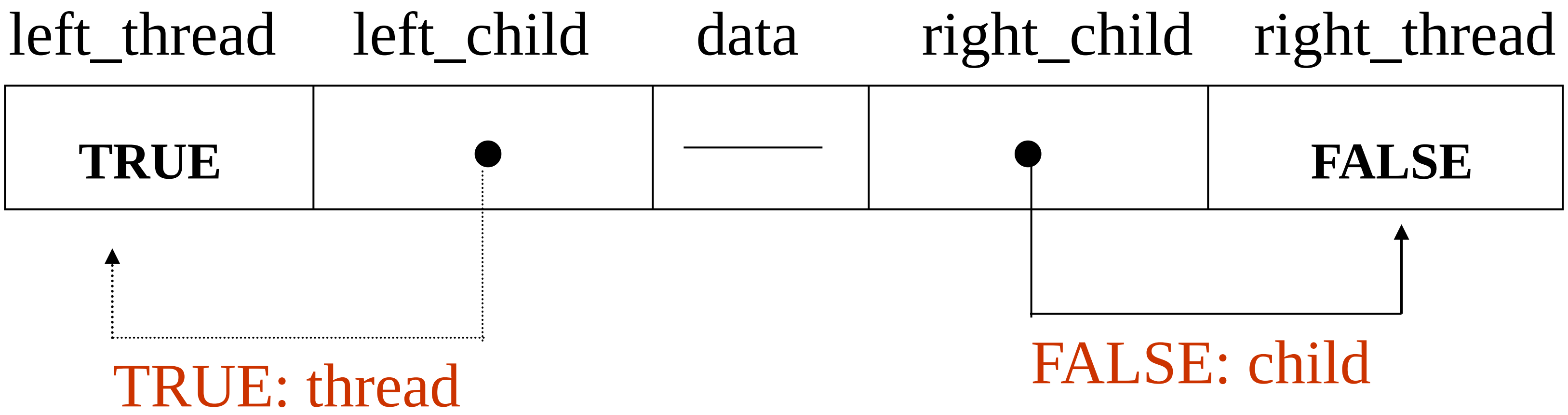
A Threaded Binary Tree



inorder traversal:

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

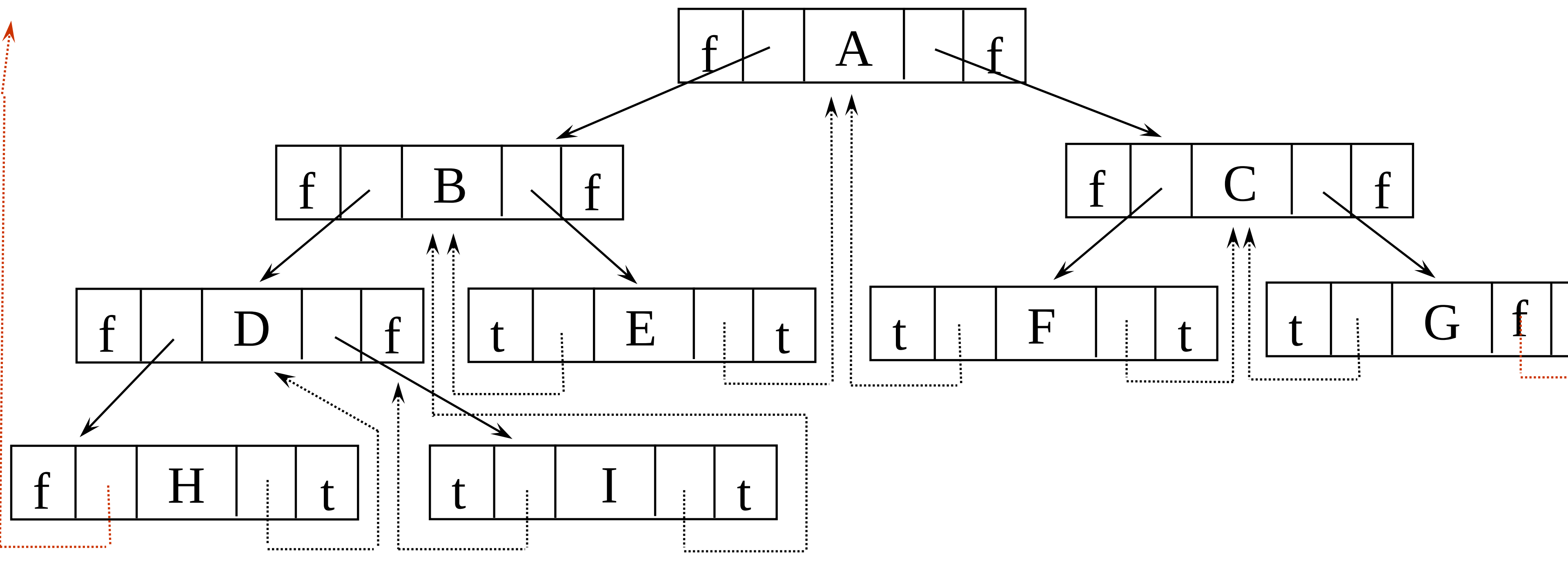
Data Structures for Threaded BT




```
struct threaded_Node {
    short int left_thread;
    struct threaded_Node* left_child;
    char data;
    struct threaded_Node* right_child;
    short int right_thread;
};
```

Memory Representation of A Threaded BT

inorder traversal:
H, D, I, B, E, A, F, C, G



Inorder Traversal of Threaded BT

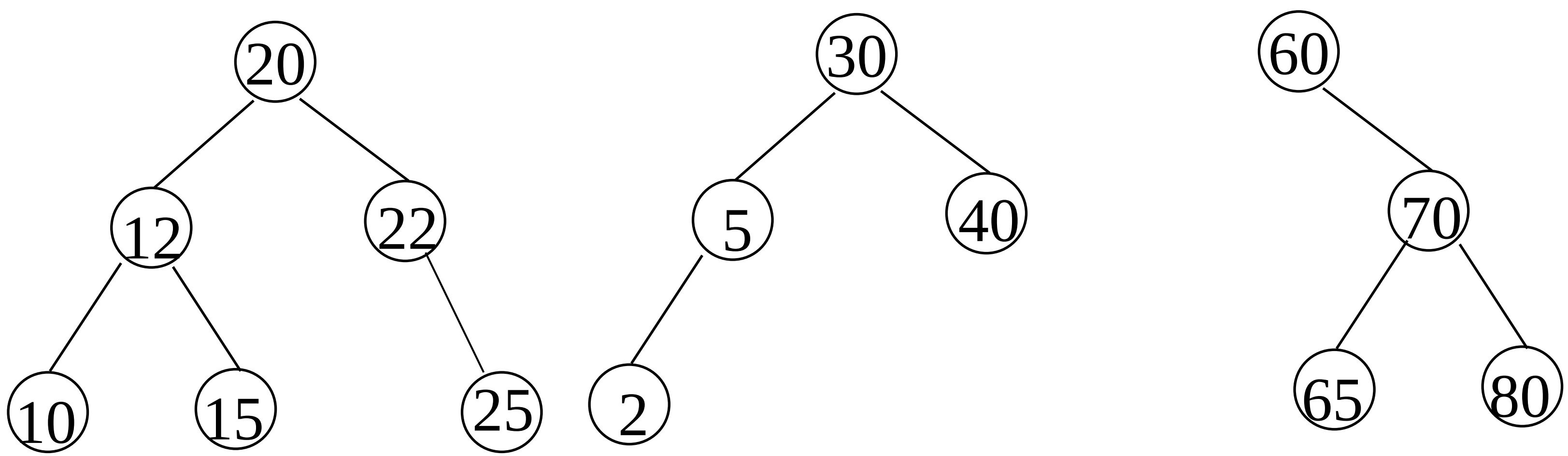
```
void tinorder(threaded_Node ptr){
/* traverse the threaded binary tree
  inorder */
threaded_Node temp=getLeftmostNode(ptr)
while(temp!=NULL){
 print temp.data
 if(temp.rightThread==true)
        temp=temp.right_child
 else
        temp=getLeftmostNode(temp.right)
}
}
```

O(n)

Binary Search Tree

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

Examples of Binary Search Trees



Searching a Binary Search Tree

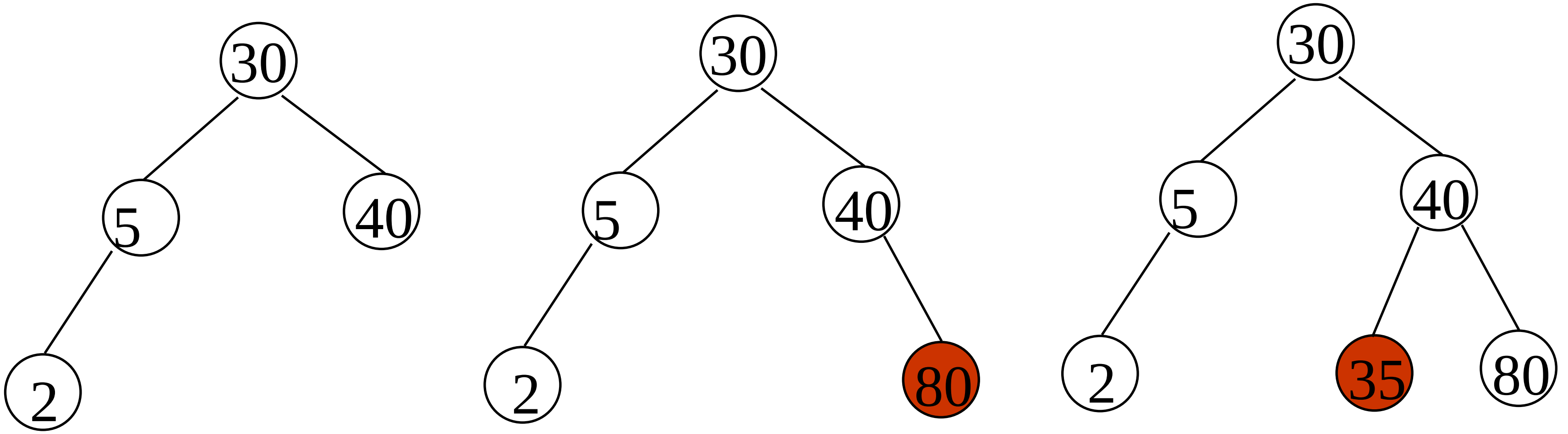
```
tree_pointer search(tree_pointer root,
              int key){
/* return a pointer to the node that
  contains key. If there is no such
  node, return NULL */

   if (root==NULL) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
       return search(root->left_child,
                     key);
   return search(root->right_child,key);
}
```

Another Searching Algorithm

```
tree_pointer search2(tree_pointer tree,
  int key){
   while (tree) {
     if (key == tree->data) return tree;
     if (key < tree->data)
         tree = tree->left_child;
     else tree = tree->right_child;
   }
   return NULL;
}
```

O(h)

Insert Node in Binary Search Tree



Insert 80

Insert 35

Insertion/Deletion in a Binary Search Tree

- Refer class notebook for iterative and recursive algorithm

Important Assignments

- Find height of Binary tree
- Count no. of nodes in Binary tree
- Count no. of leaf nodes in Binary tree
- Count no. of non-leaf nodes in Binary tree. (Also count the root node)
- Find the diameter of a binary tree

- [Get your solutions verified from me]