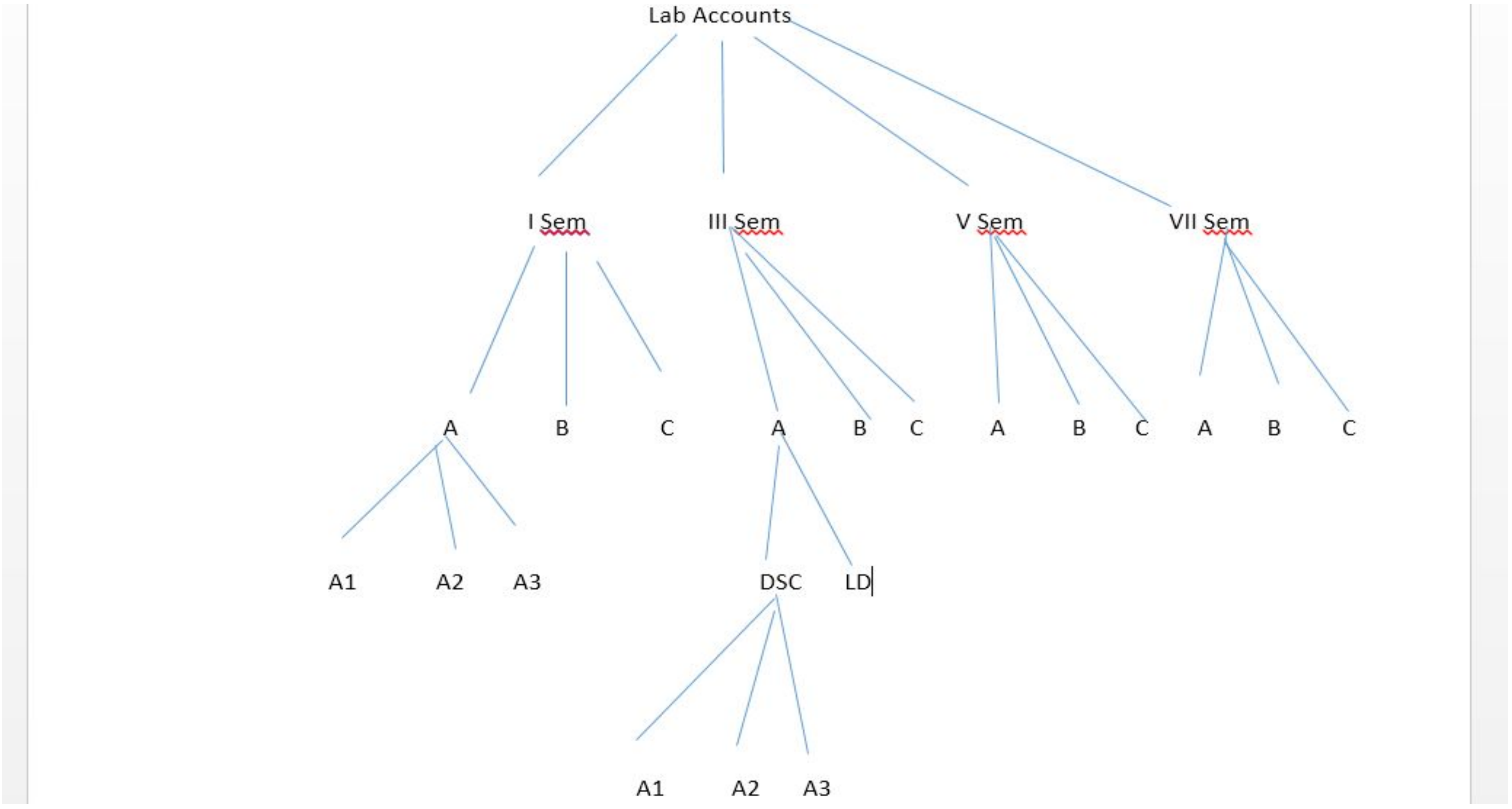
Trees

Outline

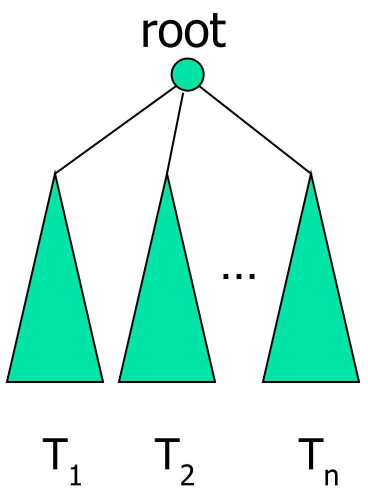
- Introduction
- Binary Trees
- Types of Binary Tree
- Binary Tree Traversals
- Additional Binary Tree Operations
- Binary Search Trees
- Binary Expression Trees
- Heaps

Introduction



The Definition of Tree

- A ***tree*** is a collection of one or more nodes and finite set directed lines called branches that connect the nodes. ('N' nodes->N-1 edges)
- Tree is a acyclic connected graph.
  - (1) There is a specially designated node called the ***root***.
  - (2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, ..., T_n, where each of these sets is a tree.
    We call T_1, ..., T_n, the sub-trees of the root.
  - No. of branches associated with a node is called degree of a node.



Terminologies of Tree

Root: First node in the tree: Indegree-0

Child: Nodes which are all reachable from node X with **only one edge** are called children of X and X is called parent node.

Siblings: Nodes with same parent.

Ancestors: All the node in the path from **root to a node X** are called Ancestors of X.

Descendents: All the nodes below a node X are called descendents of X.

Leaf: A node without a child.(outdegree-0)

Internal Nodes: Nodes except leaves.

External Nodes: All leaves.

Terminologies of Tree

Level/Depth of a node: Distance of a node from the root.

Length of the path(Unique) from root to a node X is called level or depth of X. Root is at level 0.

Depth of a tree is the depth of the deepest leaf node.

Height of a node: Length of the longest path from a node X to leaf node is called height of node X.

Root will be at the highest height.

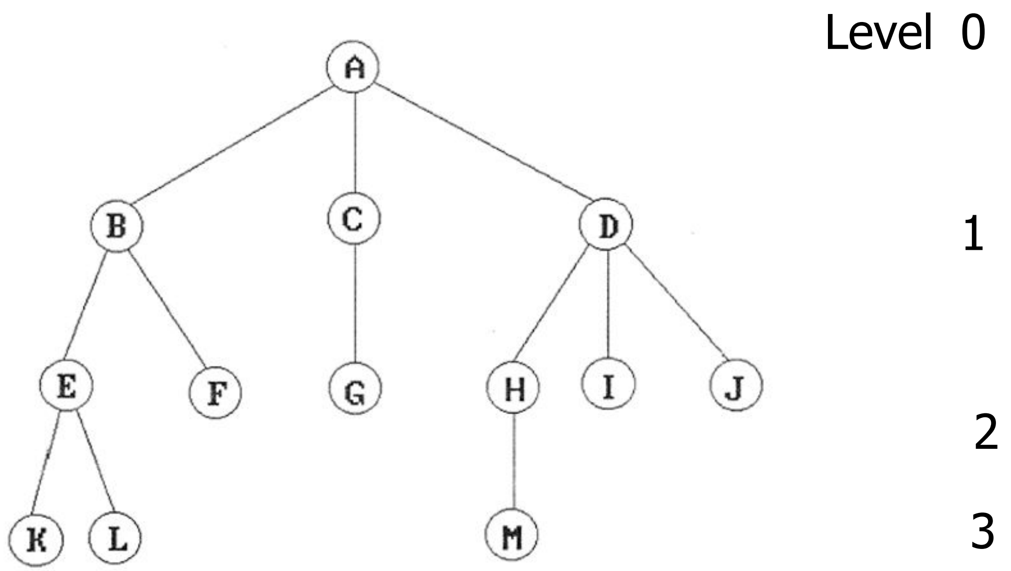
All leaves will be at height 0.

Height of the tree is the height of Root.

Depth(Tree) = Height(Tree)

Terminologies of Tree

- The root of this tree is node *A.*
- Definition:
  - Root:A
  - Children (E, F)
  - Siblings(B,C, D)
  -
  - Leaf / Leaves
    - K, L, F, G, M, I, J…
    - Ht(B) = 2, Ht(I) = 0 , Ht(A) = 3



Representation of Trees

Array Representation:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23……………………….32

A	B	C	D	E	-	F	-	G	-	H	I	J	K	-	L	-	-	-	-	-	-	-		-	-	-	-	M	-	-

If a node is at index 'i'

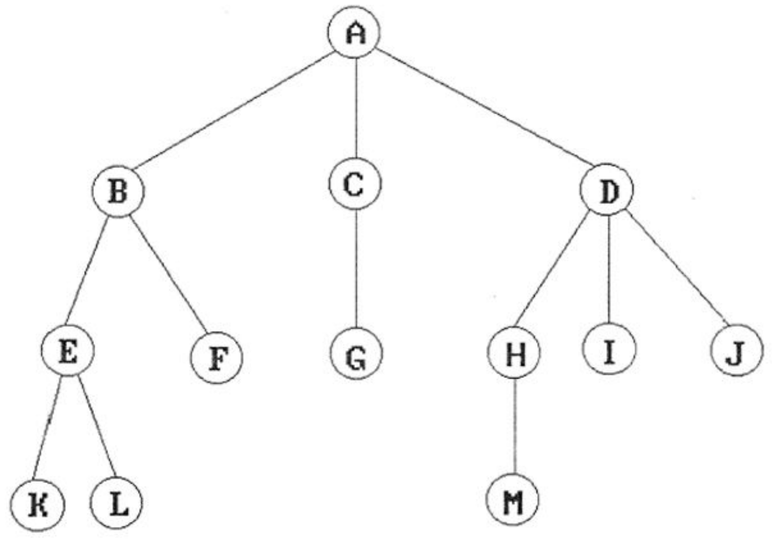
first child index = 3i +1

second child index = 3i +2

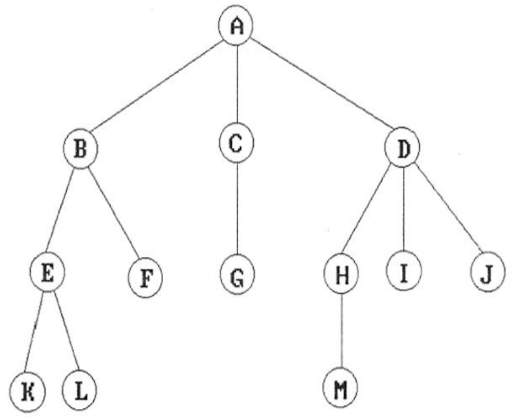
third child index = 3i +3

If a node is at index 'i'

Its parent index = (i-1)/3



Representation of Tre



- List Representation
  - The root comes first, followed by a list of sub-trees
  - Example: (A(B(E(K,L),F),C(G),D(H(M),I, J)))

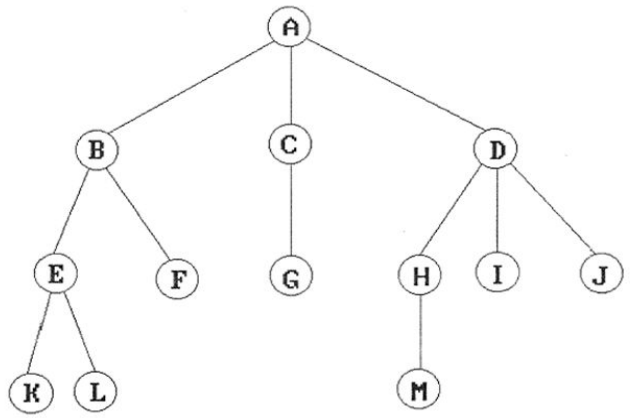
data	link 1	link 2	…	link n

A node must have a varying number of link fields depending on the number of branches

Representation of Trees

- Left Child-Right Sibling Representation

data	
left child	right sibling



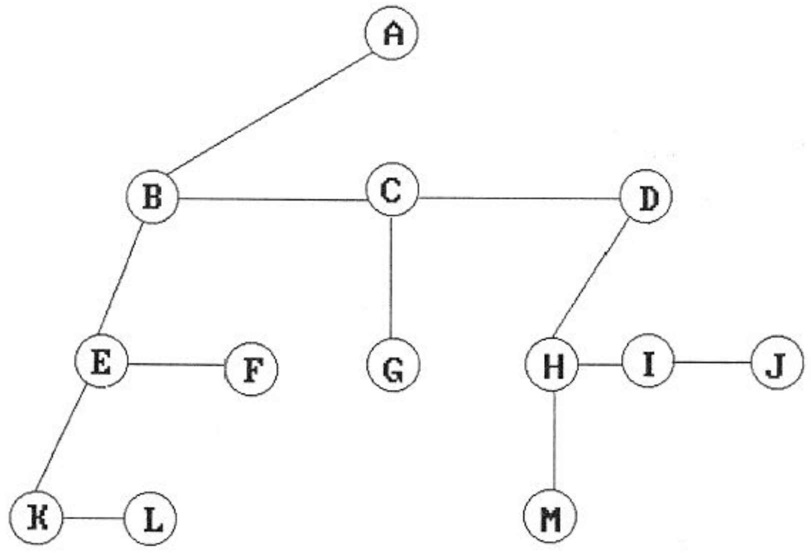
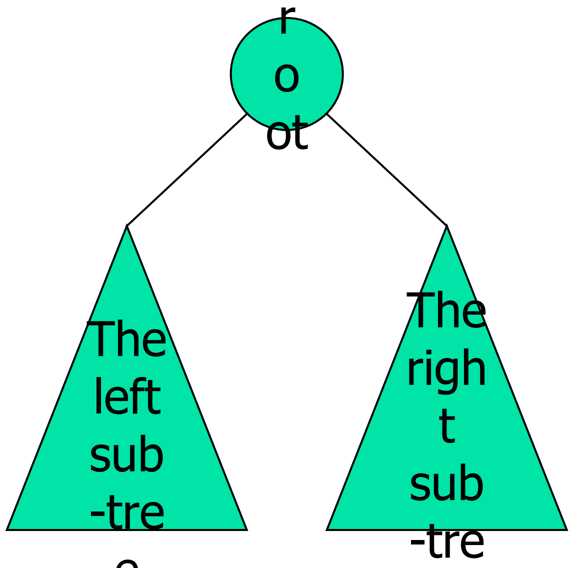


Figure 5.5: Left child-right sibling representation of a tree

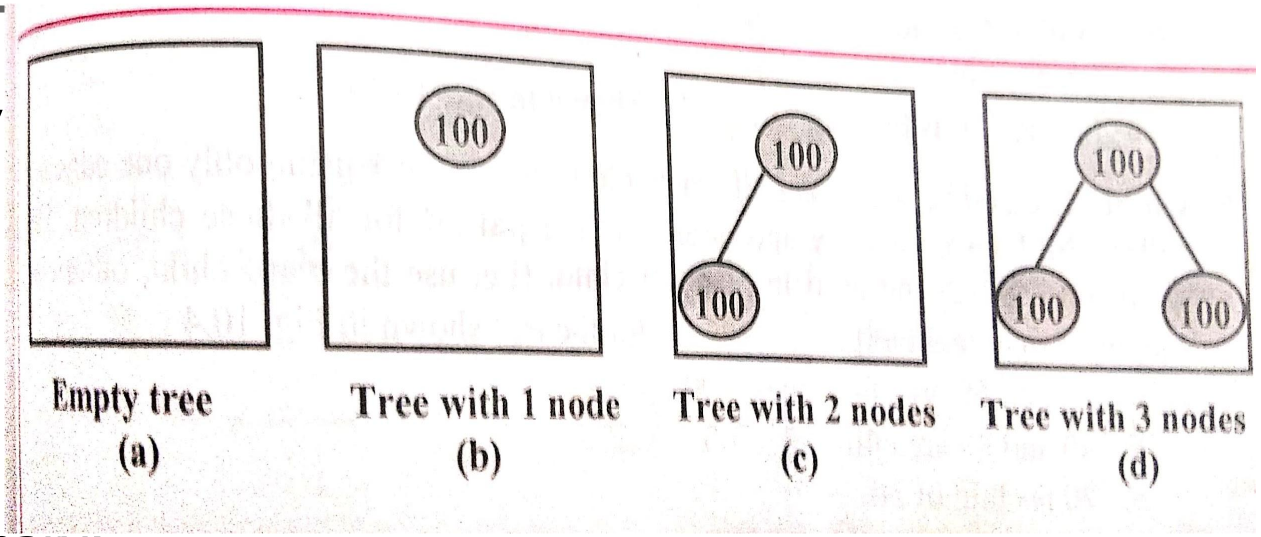
Binary Trees

- 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 sub-tree and the right sub-tree.
- Any tree can be transformed into a binary tree.
  - By using left child-right sibling representation
- The left and right subtrees are distinguished

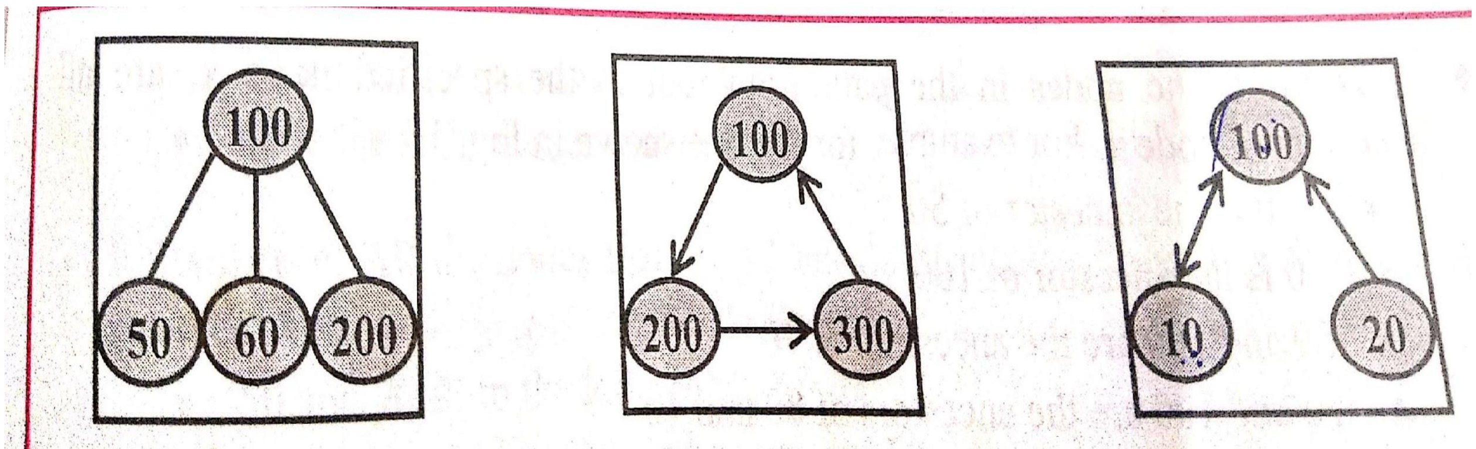


Binary Trees

Trees which are Binary



Trees Which are not Binary:

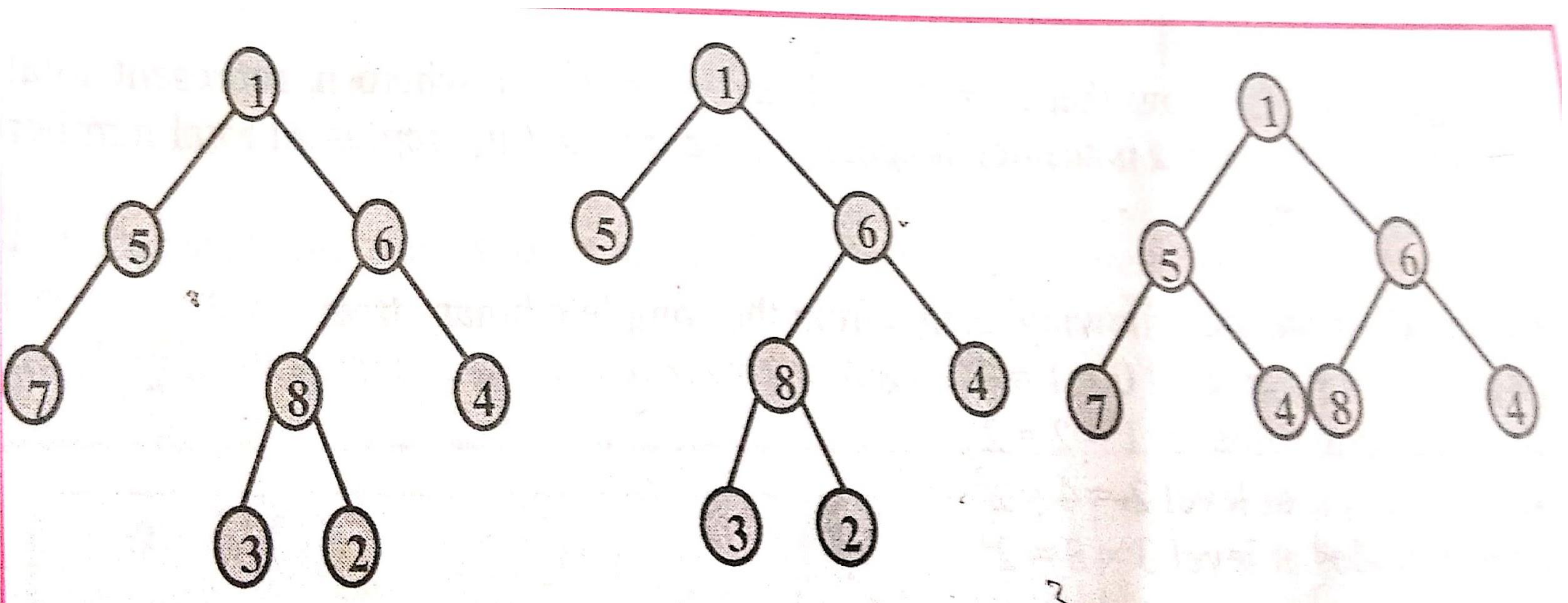


Types of Binary Trees

Strictly Binary Tree: A Binary tree in which every node has either 0 or two subtrees.

(a) is a BT but not SBT:

(b) and ©are SBTs

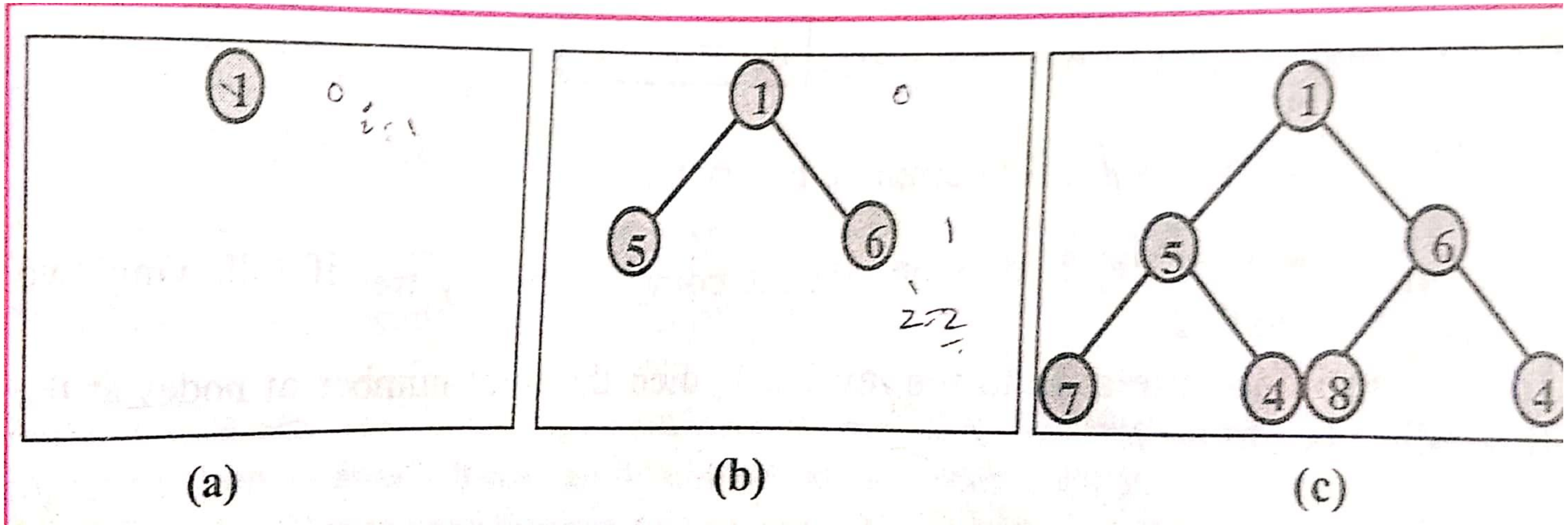


Types of Binary Trees

Complete Binary Tree: Is a SBT, in which the no. of nodes at any level i is 2^i

Total no. of nodes in CBT of depth 'd' $N = 2^{d+1} - 1$

$2^{d+1} - 1 = N \quad d = \log(N+1) - 1 = \log N$



CBT

Total no. of nodes in CBT of depth 'd' = $2^{d+1} - 1$

Proof: No. of nodes at level 0 = 2^0

No. of nodes at level 1 = 2^1

No. of nodes at level 2 = 2^2

…

No. of nodes at level d = 2^d

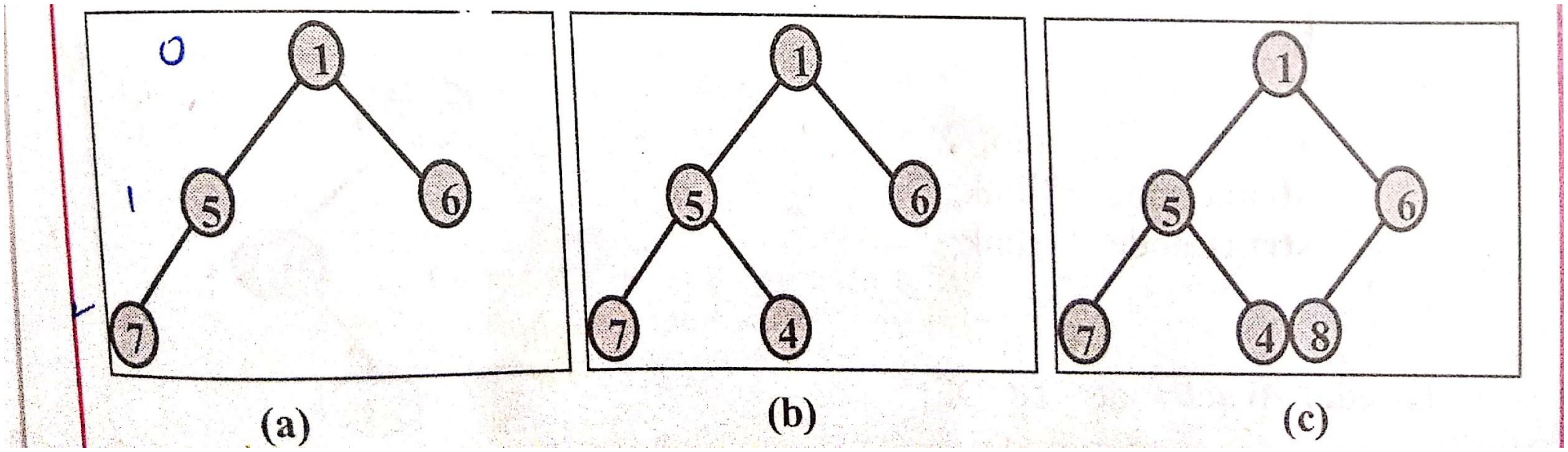
Total no of nodes = $2^0 + 2^1 + 2^2 + \ldots\ldots + 2^d = 2^{d+1} - 1$

Total no of leaf nodes = 2^d

Total no of non- leaf nodes = $2^{d+1} - 1 - 2^d = 2^d - 1$

Almost Complete Binary tree

A Binary tree of depth ‘d’ is almost complete BT, if it is complete up to level ‘ d-1’ with 2^{d-1} of nodes at level d-1 and which is not complete at level ‘d’(no. of nodes at level ‘d’ are less than 2^d) with the nodes present at level ‘d’ only from left to right.



Special Binary Trees

- Skewed Binary Trees
  - Fig.5.9 (a)
- Complete Binary Trees
  - Fig.5.9 (b)
  - This will be defined shortly

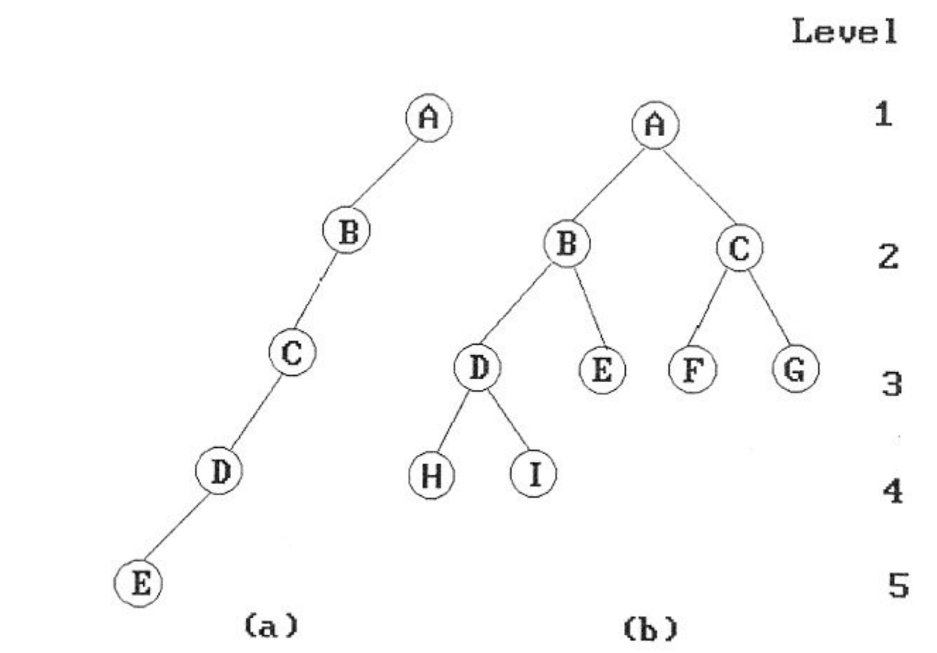


Figure 5.9: Skewed and complete binary trees

Binary Search Tree : Implement Dictionary

Binary Expression Tree: To store Binary expressions.

Representation of Binary Trees

Array Representation:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	6	7	-	8	4	-	-			3	2		

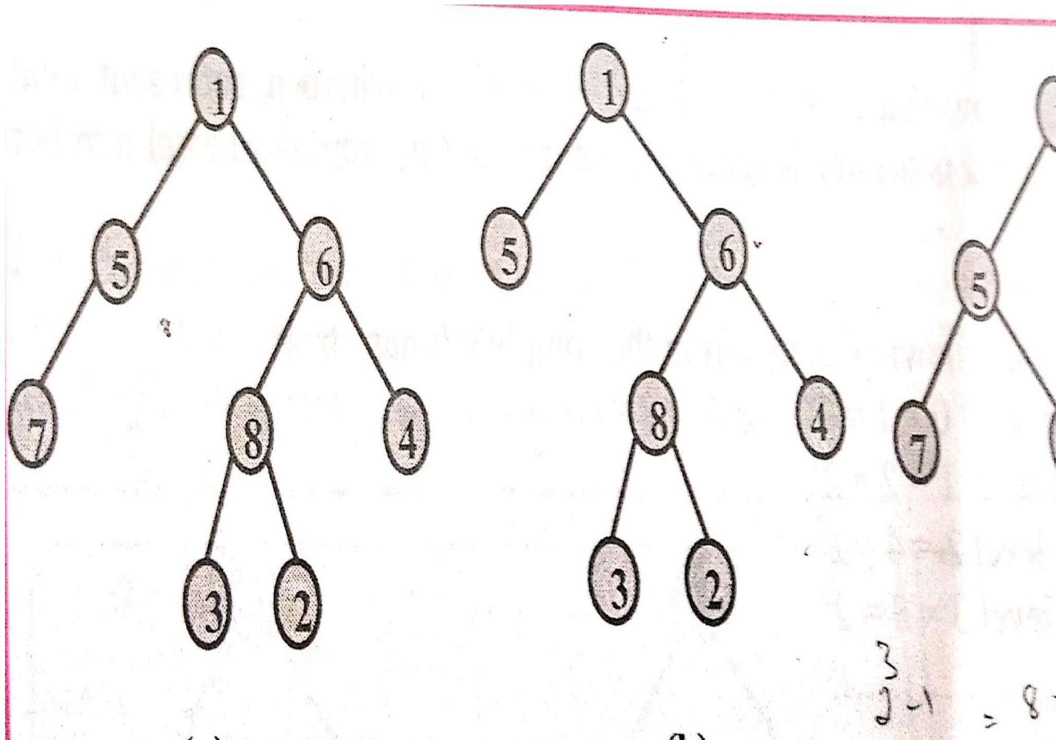
If a node is at index ‘i’

Left child index = 2i +1

Right child index = 2i +2

If a node is at index ‘i’

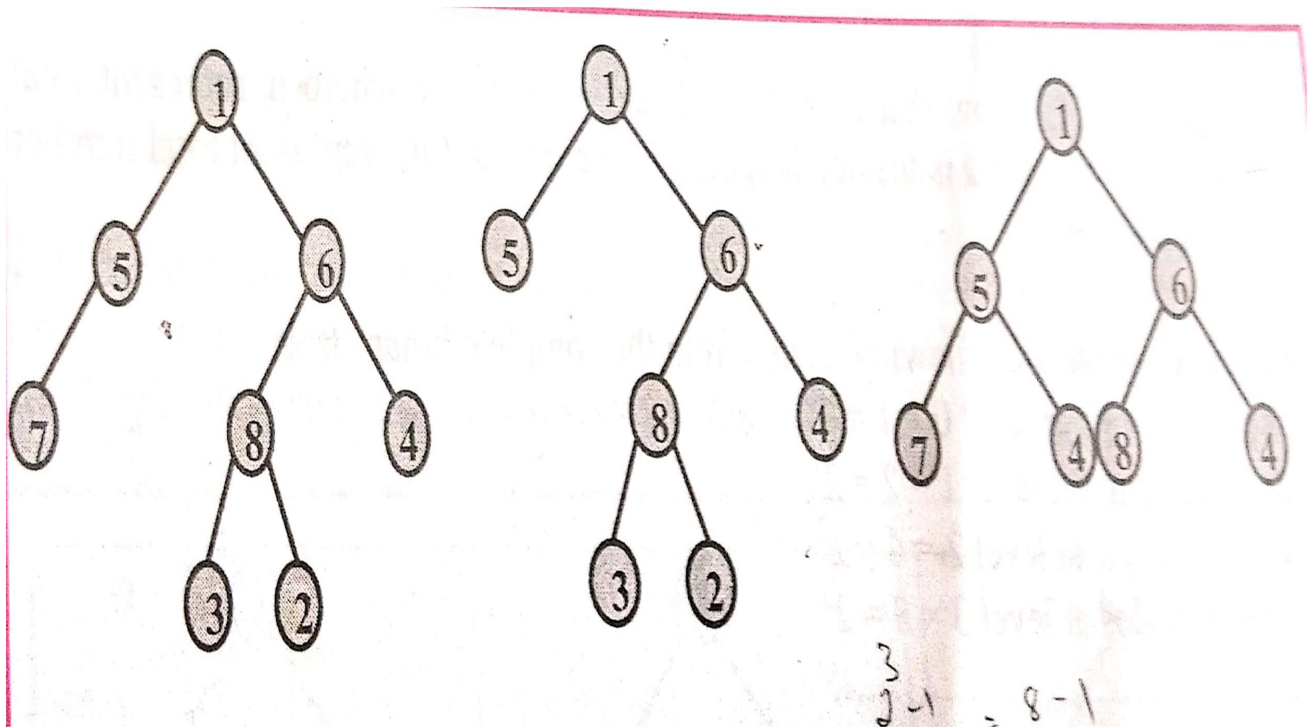
Its parent index = (i-1)/2



Representation of Binary Trees

Linked Representation: Double linked list

Left link	Info	Right Link



Binary Tree Operations(ADT)

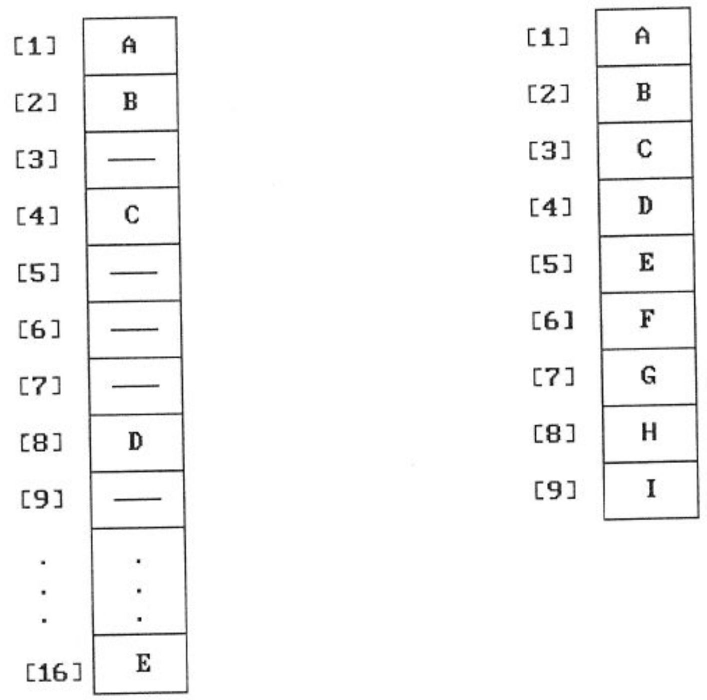
- Creation(Through Insertion)
- Traversal
- Search
- Delete
- Update
- Creating Copy
- Find Height

```
            struct node
        {
           int info;
           struct node *llink, *rlink;
        };
      typedef  struct node *NODE
   void main()
{
     NODE Root = NULL;
     //Design menu
}
```

```
NODE Insert(NODE R)
{
     NODE NN,  CN, PN; //  New node, child node, parent node
      char dir[10];  // to read the direction
       // create new node  with malloc
      // Read and assign info
      NN->llink = NN->rlink = NULL;
    if(R==NULL)
       return NN;
    // Read direction
```

Binary Tree Representation

- Array Representation (Fig. 5.11)
- Linked Representation (Fig. 5.13)



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

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

Figure 5.11: Array representation of binary trees of Figure 5.9

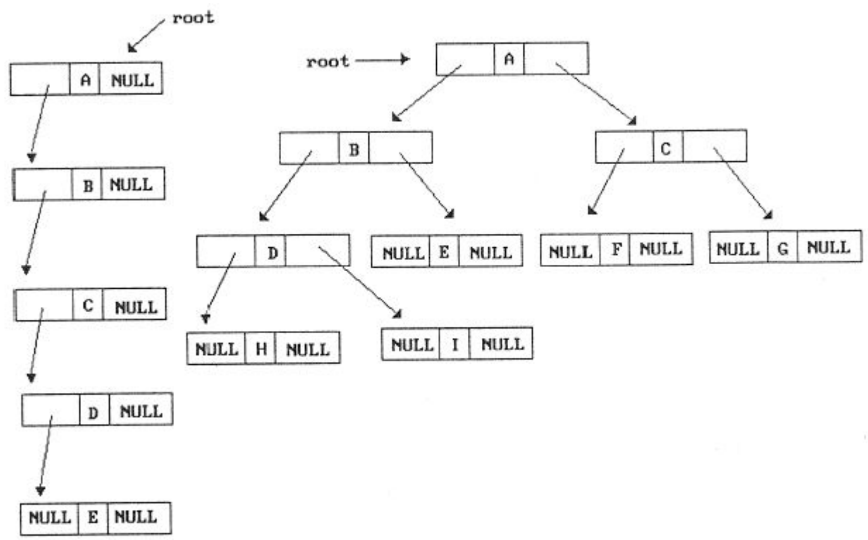


Figure 5.13: Linked representation for the binary trees of Figure 5.9

5.3 Binary Tree Traversals

- Traversing order : *L, V, R*
  - *L* : moving left
  - *V* : visiting the node
  - *R* : moving right
- Inorder Traversal : *LVR*
- Preorder Traversal : *VLR*
- Postorder Traversal : *LRV*

For Example

- Inorder Traversal : A / B * C * D + E
- Preorder Traversal : + * * / A B C D E
- Postorder Traversal : A B / C * D * E +

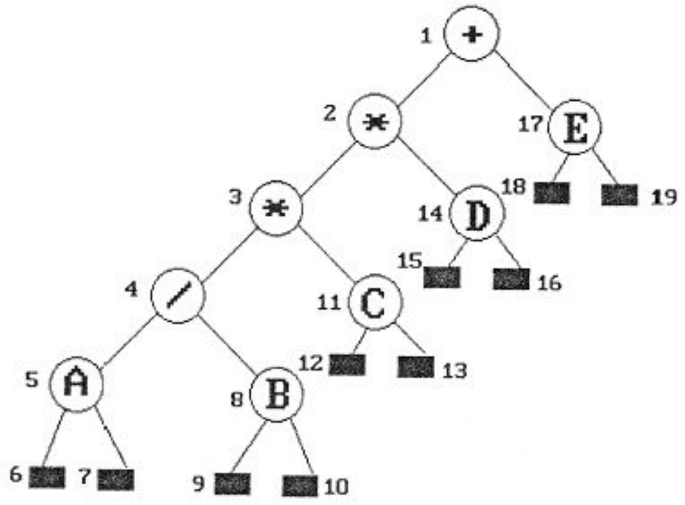
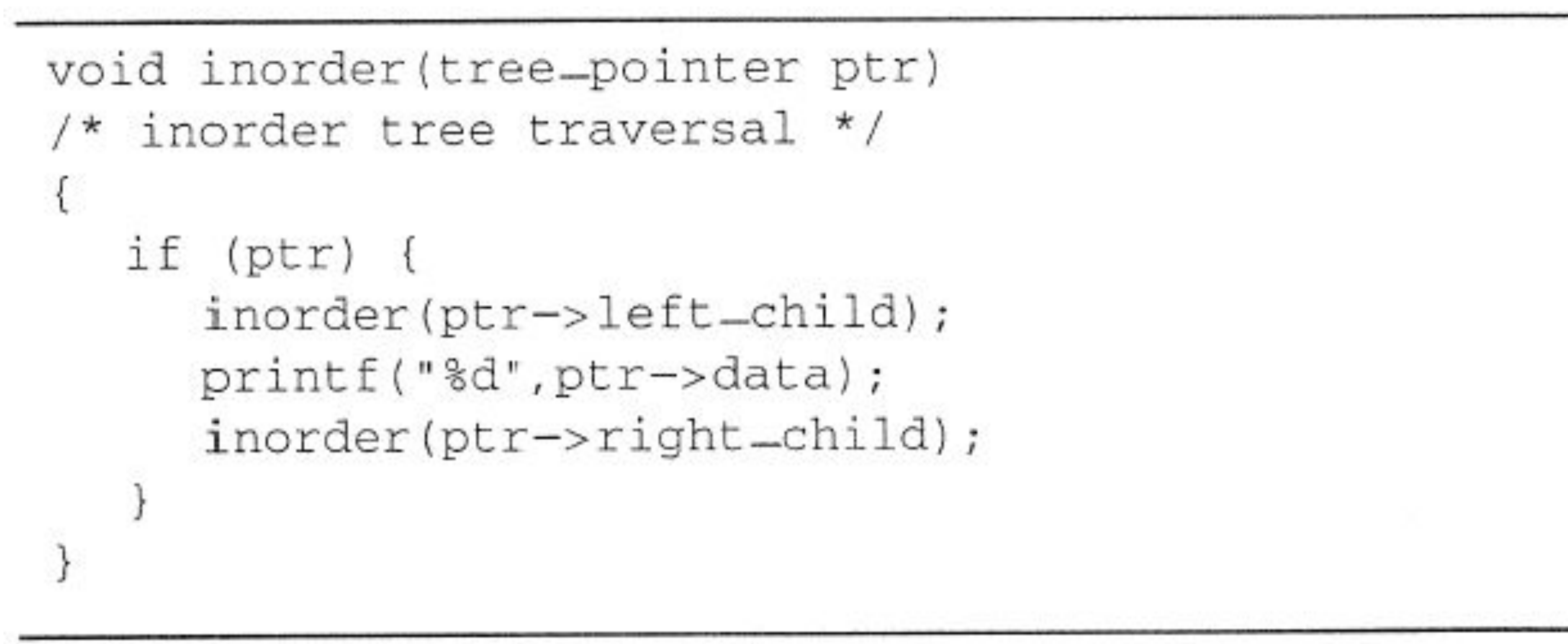


Figure 5.15: Binary tree with arithmetic expression

Inorder Traversal (1)

- A recursive function starting from the root
  - Move left Visit node Move right



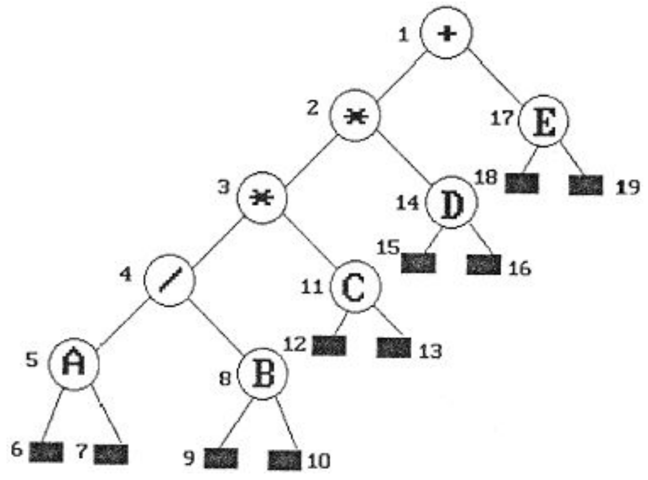
```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d",ptr->data);
      inorder(ptr->right_child);
   }
}
```

Program 5.1: Inorder traversal of a binary tree

Inorder Traversal (2)

Call of inorder	Value in root	Action	inorder	in root	Value 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*	

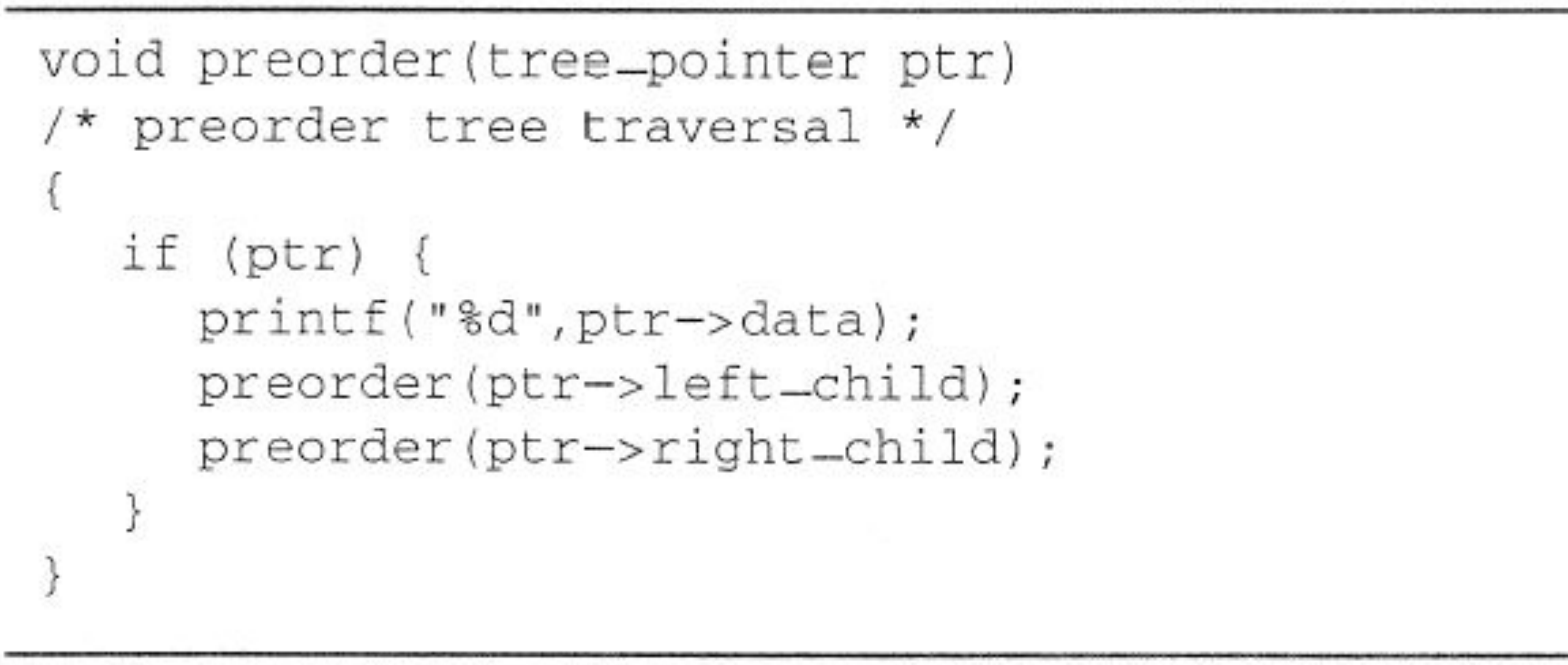
Figure 5.16: Trace of Program 5.1



In-order Traversal :
<u>A / B * C * D + E</u>

Preorder Traversal

- A recursive function starting from the root
  - Visit node Move left Move right

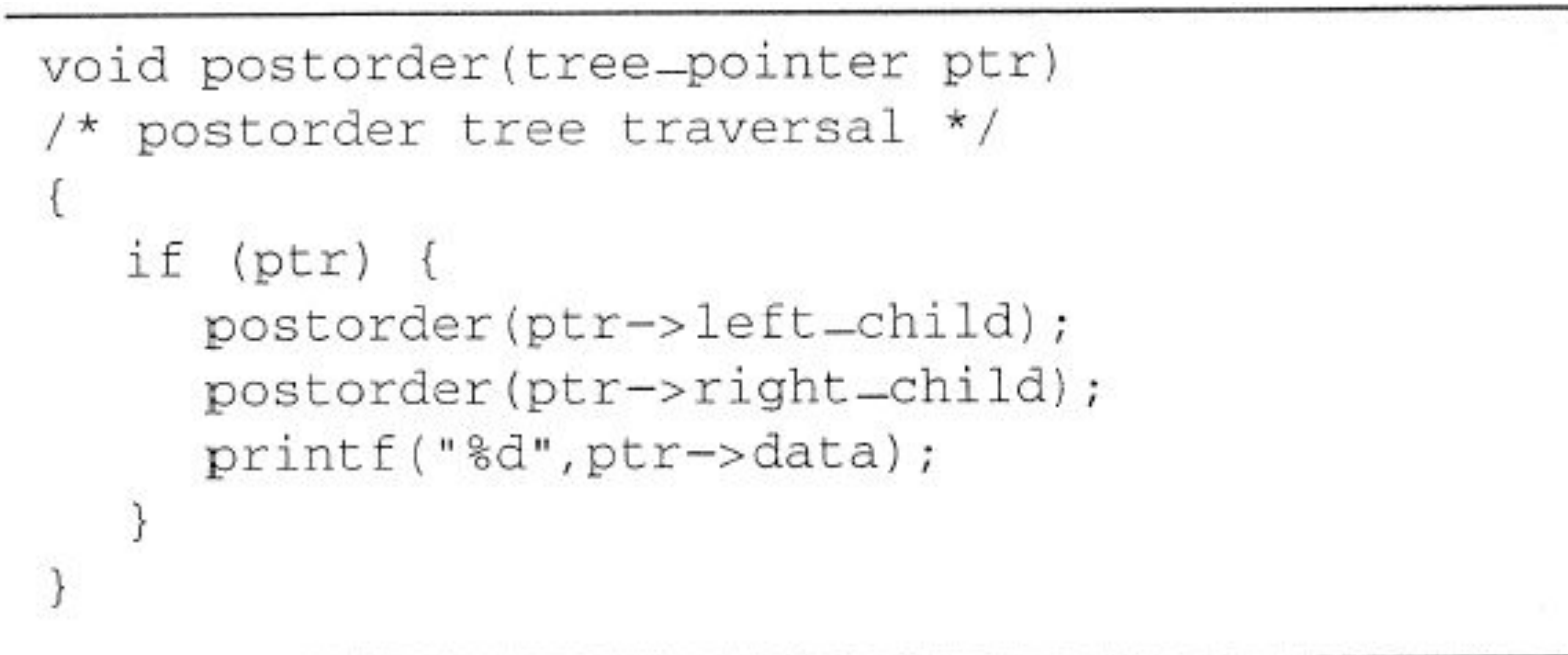


```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
  if (ptr) {
     printf("%d",ptr->data);
     preorder(ptr->left_child);
     preorder(ptr->right_child);
  }
}
```

Program 5.2: Preorder traversal of a binary tree

Postorder Traversal

- A recursive function starting from the root
  - Move left →Move right →Visit node



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

Program 5.3: Postorder traversal of a binary tree

Other Traversals

- Iterative Inorder Traversal
  - Using a stack to simulate recursion
  - Time Complexity: O(*n*), *n* is #num of node.
- Level Order Traversal
  - Visiting at each new level from the left-most node to the right-most
  - Using Data Structure : Queue

Iterative In-order Traversal (1)

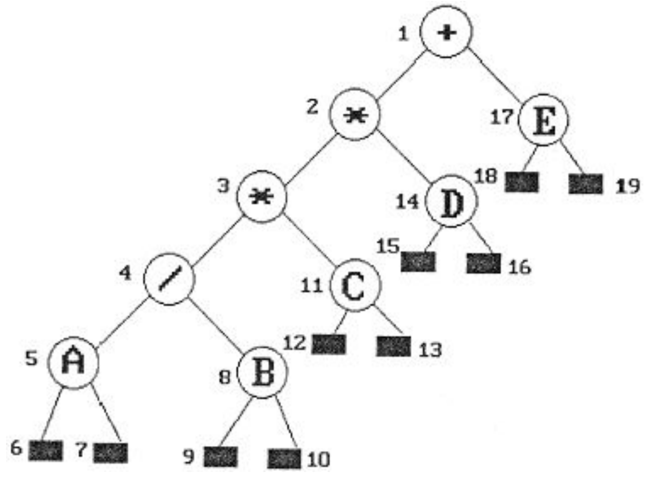
```
void iter_inorder(tree_pointer node)
{
   int top = -1; /* initialize stack */
   tree_pointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node->left_child)
         add(&top, node); /* add to stack */
      node = delete(&top); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node->data);
      node = node->right_child;
   }
}
```

Program 5.4: Iterative inorder traversal

Iterative In-order Traversal (2)

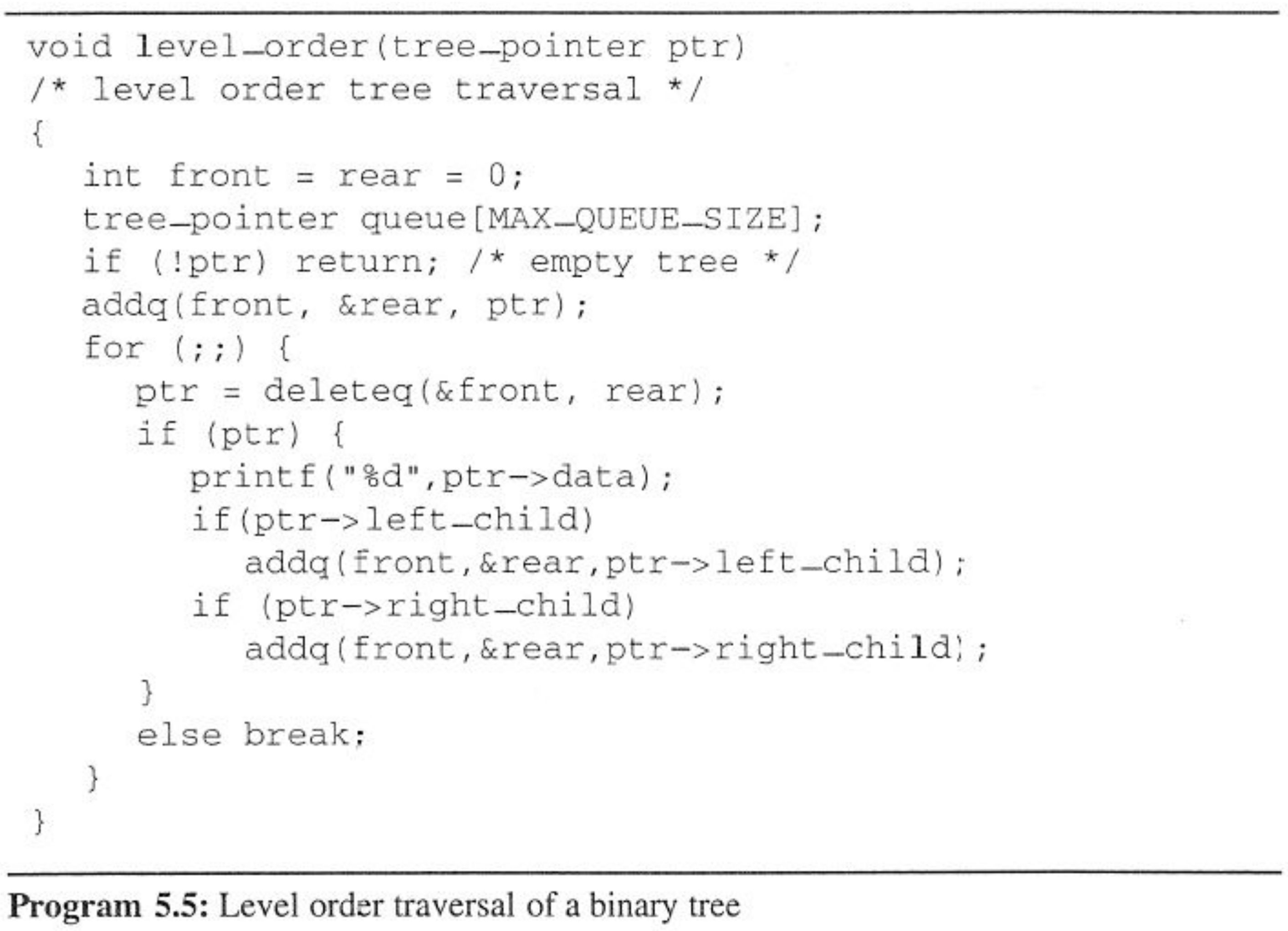
Add "+" in stack
Add "*"
Add "*"
Add "/"
Add "A"
Delete "A" & Print
Delete "/" & Print
Add "B"
Delete "B" & Print
Delete "*" & Print
Add "C"

Delete "C" & Print
Delete "*" & Print
Add "D"
Delete "D" & Print
Delete "+" & Print
Add "E"
Delete "E" & Print



In-order Traversal :
A / B * C * D + E

Level Order Traversal (1)



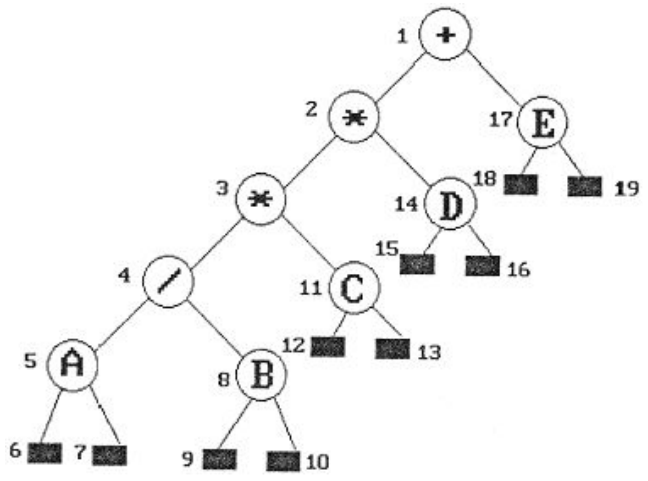
```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
  int front = rear = 0;
  tree_pointer queue[MAX_QUEUE_SIZE];
  if (!ptr) return; /* empty tree */
  addq(front, &rear, ptr);
  for (;;) {
    ptr = deleteq(&front, rear);
    if (ptr) {
      printf("%d",ptr->data);
      if(ptr->left_child)
        addq(front,&rear,ptr->left_child);
      if (ptr->right_child)
        addq(front,&rear,ptr->right_child);
    }
    else break;
  }
}
```

Program 5.5: Level order traversal of a binary tree

Level Order Traversal (2)

Add "+" in Queue
Deleteq "+"
Addq "*"
Addq "E"
Deleteq "*"
Addq "*"
Addq "D"
Deleteq "E"
Deleteq "*"

Addq "/"
Addq "C"
Deleteq "D"
Deleteq "/"
Addq "A"
Addq "B"
Deleteq "C"
Deleteq "A"
Deleteq "B"



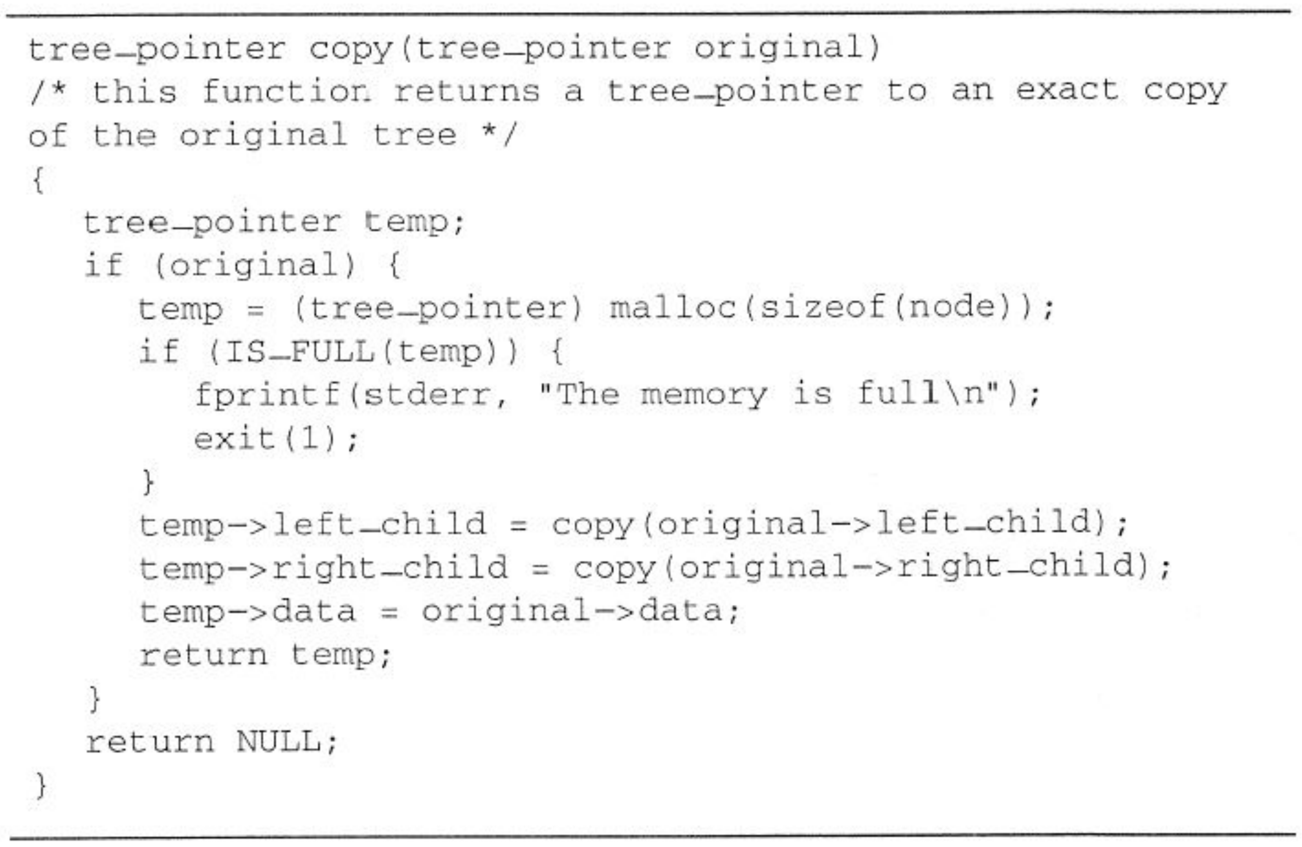
Level-order Traversal :
<u>+ * E * D / C A B</u>

5.4 Additional Binary Tree Operations

- Copying Binary Trees
  - Program 5.6
- Testing for Equality of Binary Trees
  - Program 5.7
- The Satisfiability Problem (SAT)

Copying Binary Trees

- Modified from postorder traversal program



```
tree_pointer copy(tree_pointer original)
/* this function returns a tree_pointer to an exact copy
of the original tree */
{
   tree_pointer temp;
   if (original) {
      temp = (tree_pointer) malloc(sizeof(node));
      if (IS_FULL(temp)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      temp->left_child = copy(original->left_child);
      temp->right_child = copy(original->right_child);
      temp->data = original->data;
      return temp;
   }
   return NULL;
}
```

Program 5.6: Copying a binary tree

5.5 Threaded Binary Trees (1)

- Linked Representation of Binary Tree
  - more null links than actual pointers (waste!)
- Threaded Binary Tree
  - Make use of these null links
  - Threads
    - Replace the null links by pointers (called threads)
    - If ptr -> left_thread = TRUE
      - Then ptr -> left_child is a thread (to the node before ptr)
      - Else ptr -> left_child is a pointer to left child
    - If ptr -> right_thread = TRUE
      - Then ptr -> right_child is a thread (to the node after ptr)
      - Else ptr -> right_child is a pointer to right child

5.5 Threaded Binary Trees (2)

```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
  short int left_thread;
  threaded_pointer left_child;
  char data;
  short int right_child;
  threaded_pointer right_child;
}
```

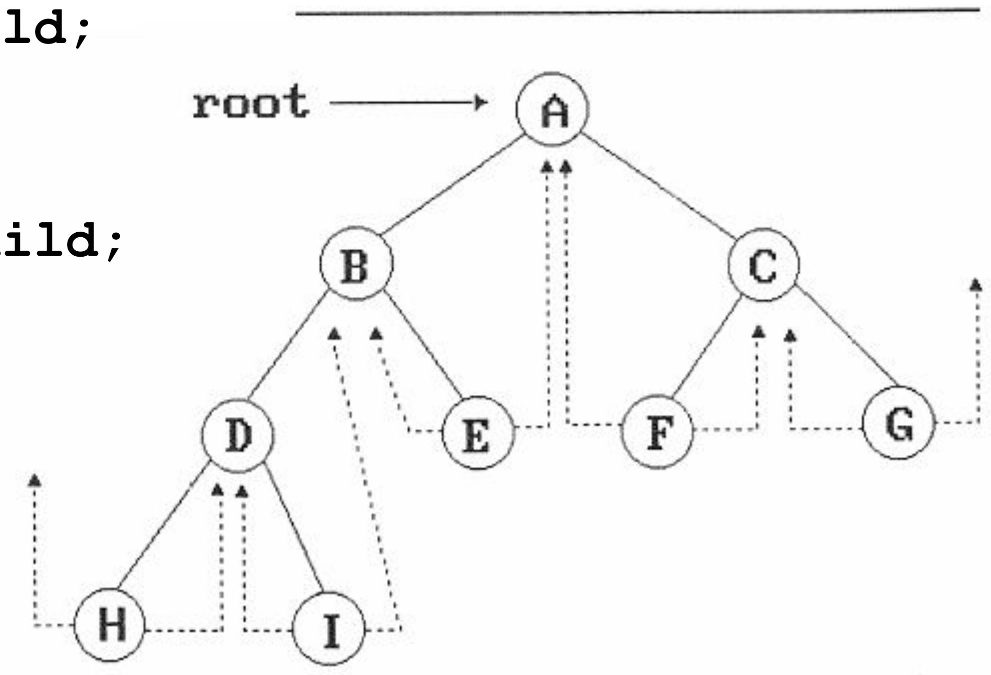
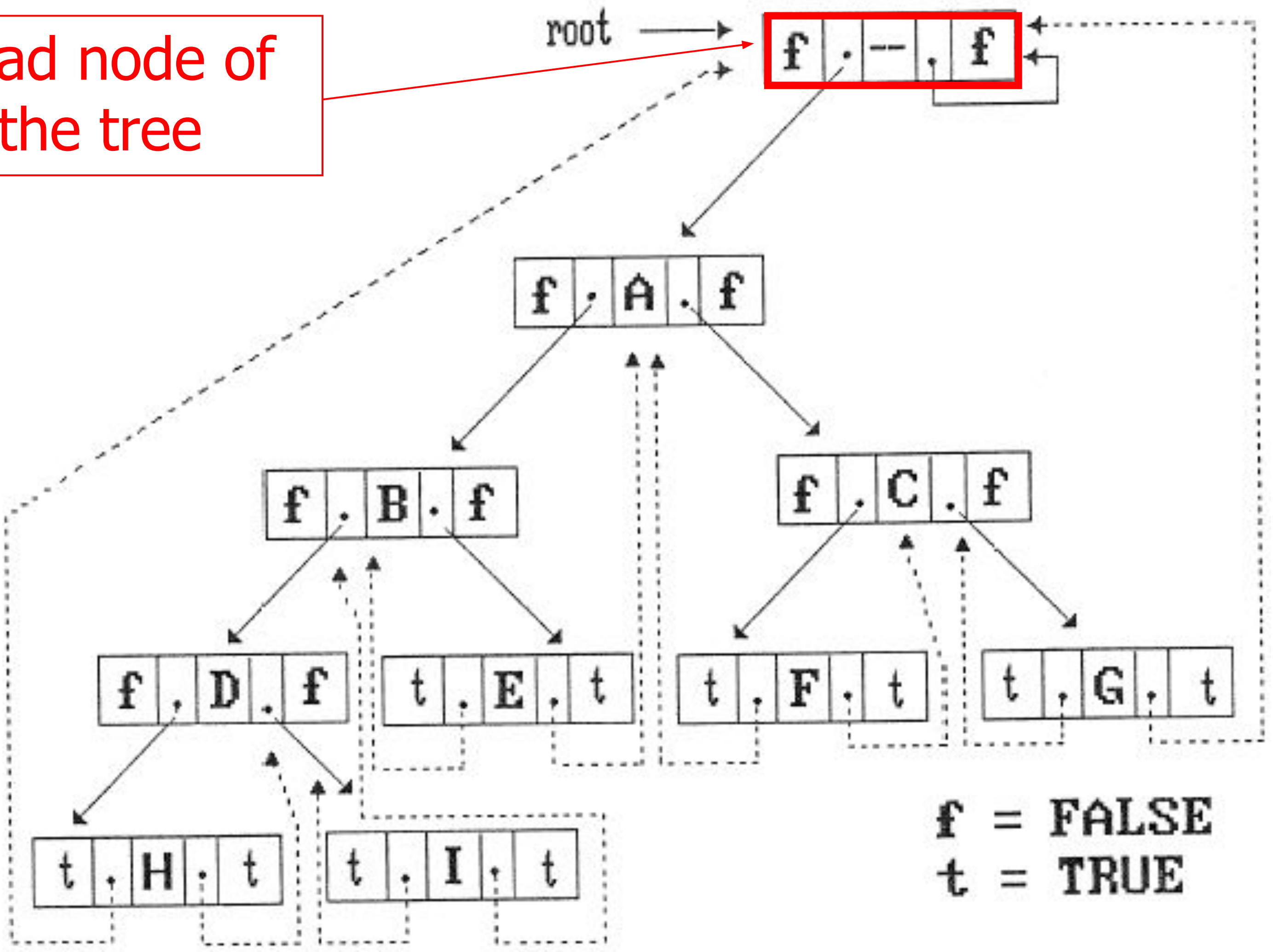


Figure 5.21: Threaded tree corresponding to Figure 5.9(b)

5.5 Threaded Binary Trees (3)

Head node of the tree

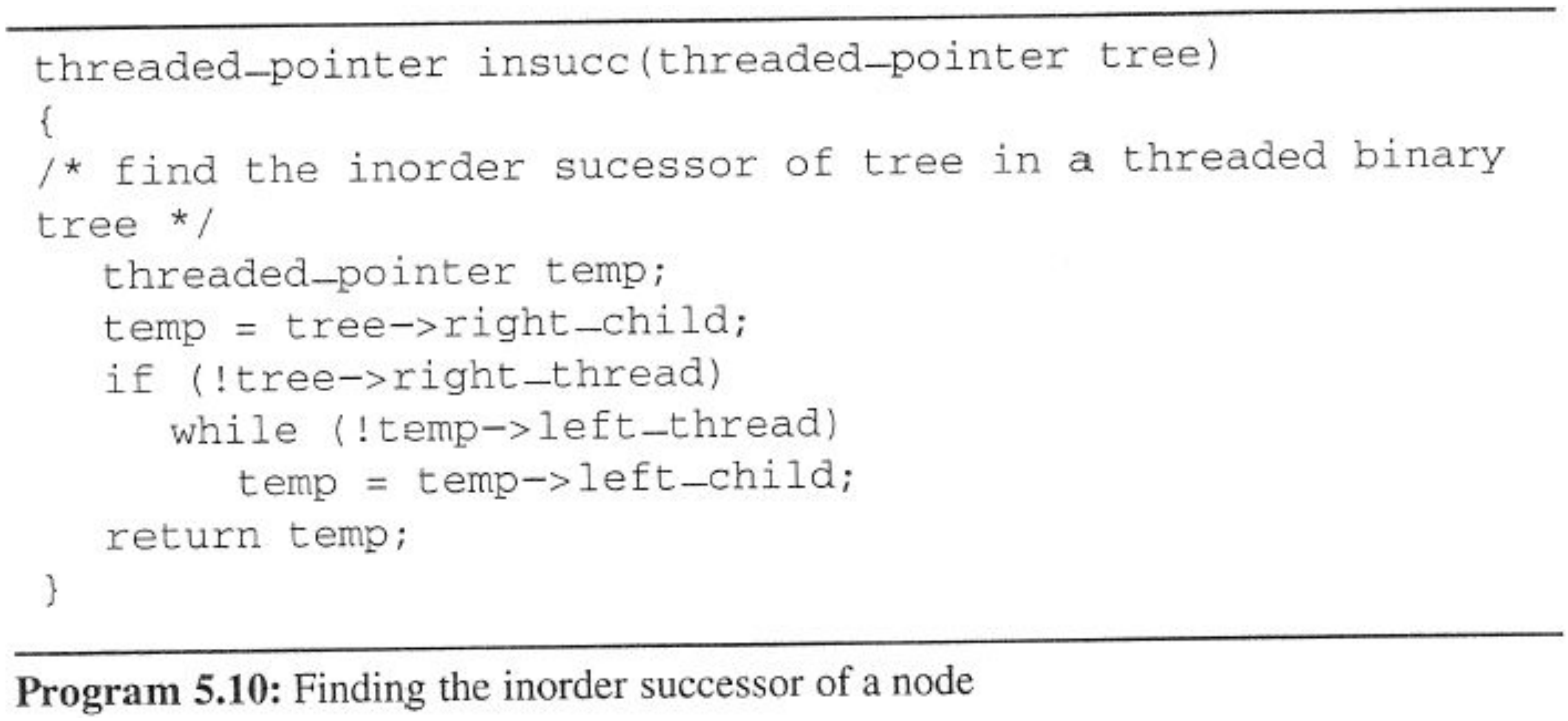
Actual tree



Inorder Traversal of a Threaded Binary Tree (1)

- Threads simplify inorder traversal algorithm
- An easy O(*n*) algorithm (Program 5.11.)
  - For any node, ptr, in a threaded binary tree
    - If ptr -> right_thread = TRUE
      - The inorder successor of ptr = ptr -> right_child
    - Else (Otherwise, ptr -> right_thread = FALSE)
      - Follow a path of left_child links from the right_child of ptr until finding a node with left_Thread = TRUE
  - Function insucc (Program 5.10.)
    - Finds the inorder successor of any node (without using a stack)

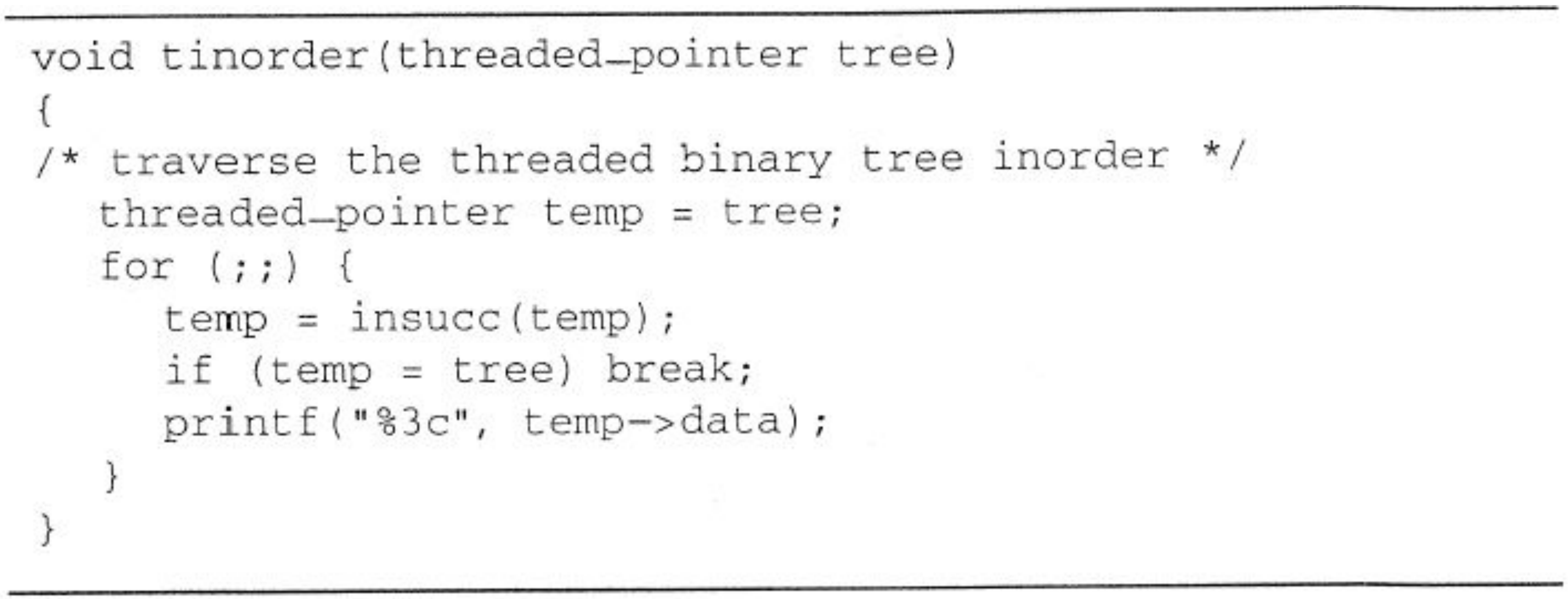
Inorder Traversal of a Threaded Binary Tree (2)



```
threaded_pointer insucc(threaded_pointer tree)
{
/* find the inorder sucessor of tree in a threaded binary
tree */
   threaded_pointer temp;
   temp = tree->right_child;
   if (!tree->right_thread)
      while (!temp->left_thread)
         temp = temp->left_child;
   return temp;
}
```

Program 5.10: Finding the inorder successor of a node

Inorder Traversal of a Threaded Binary Tree (2)



```
void tinorder(threaded_pointer tree)
{
/* traverse the threaded binary tree inorder */
   threaded_pointer temp = tree;
   for (;;) {
      temp = insucc(temp);
      if (temp = tree) break;
      printf("%3c", temp->data);
   }
}
```

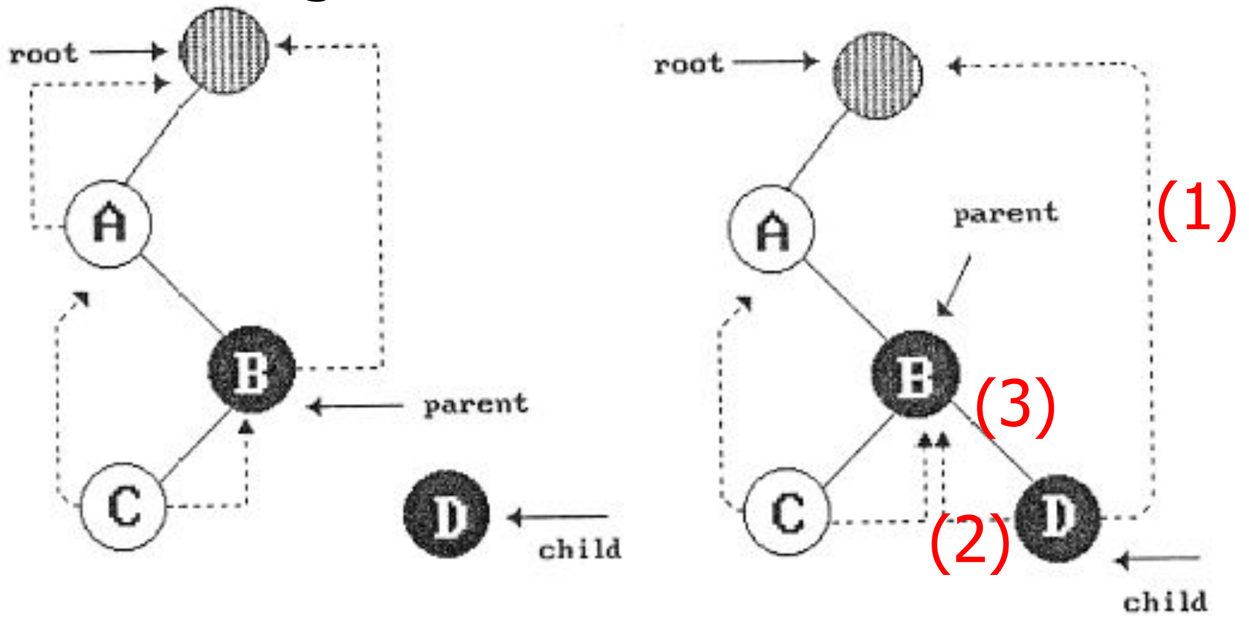
Program 5.11: Inorder traversal of a threaded binary tree

Inserting a Node into a Threaded Binary Tree

- Insert a new node as a child of a parent node
  - Insert as a left child (left as an exercise)
  - Insert as a right child (see examples 1 and 2)
- Is the original child node an empty subtree?
  - Empty child node (parent -> child_thread = TRUE)
    - See example 1
  - Non-empty child node (parent -> child_thread = FALSE)
    - See example 2

Inserting a node as the right child of the parent node (empty case)

- parent(B) -> right_thread = FALSE
- child(D) -> left_thread & right_thread = TURE
- child -> left_child = parent
- child -> right_child = parent -> right_child
- parent -> right_child = child


root
A
B
C
parent
D
child
root
A
parent
B
C
D
child
(1)
(2)
(3)